Dissertation

One-Sided Communication on a Non-Cache-Coherent Many-Core Architecture

eingereicht von

Steffen Christgau, M.Sc.

vorlegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
in der Wissenschaftsdisziplin Betriebssysteme und Verteilte Systeme

angefertigt am
Institut für Informatik und Computational Science
Professur für Betriebssysteme und Verteilte Systeme

Potsdam, den 1. Februar 2017

Supervisor
Prof. Dr. Bettina Schnor

Referees
Prof. Dr. Bettina Schnor
Prof. Dr.Wolfgang Karl
Prof. Dr.Wolfgang E. Nagel

Examining Board
Prof. Dr. Andreas Polze
Prof. Dr. Tobias Scheffer (head)
Prof. Dr.-Ing. Benno Stabernack

Faculty Delegate
Prof. Dr. Arkady Pikovsky

Christgau, Steffen
christgau@cs.uni-potsdam.de
One-Sided Communication on a Non-Cache-Coherent Many-Core Architecture
Dissertation, Institute for Computer Science
University of Potsdam, February 2017

Published online at the
Institutional Repository of the University of Potsdam:
URN urn:nbn:de:kobv:517-opus4-403100
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-403100

Abstract

Contemporary multi-core processors are parallel systems that also provide shared memory for programs running on them. Both the increasing number of cores in so-called many-core systems and the still growing computational power of the cores demand for memory systems that are able to deliver high bandwidths. Caches are essential components to satisfy this requirement. Nevertheless, hardware-based cache coherence in many-core chips faces practical limits to provide both coherence and high memory bandwidths. In addition, a shift away from global coherence can be observed. As a result, alternative architectures and suitable programming models need to be investigated.

This thesis focuses on fast communication for non-cache-coherent many-core architectures. Experiments are conducted on the Single-Chip Cloud Computer (SCC), a non-cache-coherent many-core processor with 48 mesh-connected cores. Although originally designed for message passing, the results of this thesis show that shared memory can be efficiently used for one-sided communication on this kind of architecture. One-sided communication enables data exchanges between processes where the receiver is not required to know the details of the performed communication. In the notion of the Message Passing Interface (MPI) standard, this type of communication allows to access memory of remote processes. In order to support this communication scheme on non-cache-coherent architectures, both an efficient *process synchronization* and a *communication scheme* with software-managed cache coherence are designed and investigated.

The *process synchronization* realizes the concept of the general active target synchronization scheme from the MPI standard. An existing classification of implementation approaches is extended and used to identify an appropriate class for the non-cache-coherent shared memory platform. Based on this classification, existing implementations are surveyed in order to find beneficial concepts, which are then used to design a lightweight synchronization protocol for the SCC that uses shared memory and uncached memory accesses. The proposed scheme is not prone to process skew and also enables direct communication as soon as both communication partners are ready. Experimental results show very good scaling properties and up to five times lower synchronization latency compared to a tuned message-based MPI implementation for the SCC.

For the *communication*, *SCOSCo*, a shared memory approach with software-managed cache coherence, is presented. According requirements for the coherence that fulfill

MPI's separate memory model are formulated, and a lightweight implementation exploiting SCC hard- and software features is developed. Despite a discovered malfunction in the SCC's memory subsystem, the experimental evaluation of the design reveals up to five times better bandwidths and nearly four times lower latencies in micro-benchmarks compared to the SCC-tuned but message-based MPI library. For application benchmarks, like a parallel 3D fast Fourier transform, the runtime share of communication can be reduced by a factor of up to five. In addition, this thesis postulates beneficial hardware concepts that would support software-managed coherence for one-sided communication on future non-cache-coherent architectures where coherence might be only available in local subdomains but not on a global processor level.

Zusammenfassung

Aktuelle Mehrkernprozessoren stellen parallele Systeme dar, die den darauf ausgeführten Programmen gemeinsamen Speicher zur Verfügung stellen. Sowohl die ansteigende Kernanzahlen in sogenannten Vielkernprozessoren (many-core processors) als auch die weiterhin steigende Leistungsfähigkeit der einzelnen Kerne erfordert hohe Bandbreiten, die das Speichersystem des Prozessors liefern muss. Hardware-basierte Cache-Kohärenz stößt in aktuellen Vielkernprozessoren an Grenzen des praktisch Machbaren. Dementsprechend müssen alternative Architekturen und entsprechend geeignete Programmiermodelle untersucht werden.

In dieser Arbeit wird der Single-Chip Cloud Computer (SCC), ein nicht-cachekohärenter Vielkernprozessor betrachtet, der aus 48, über ein Gitternetzwerk verbundenen Kernen besteht. Obwohl der Prozessor für nachrichten-basierte Kommunikation entwickelt worden ist, zeigen die Ergebnisse dieser Arbeit, dass einseitige Kommunikation auf Basis gemeinsamen Speichers effizient auf diesem Architekturtyp realisiert werden kann. Einseitige Kommunikation ermöglicht Datenaustausch zwischen Prozessen, bei der der Empfänger keine Details über die stattfindende Kommunikation besitzen muss. Im Sinne des MPI-Standards ist so ein Zugriff auf Speicher entfernter Prozesse möglich. Zur Umsetzung dieses Konzepts auf nicht-kohärenten Architekturen werden in dieser Arbeit sowohl eine effiziente *Prozesssynchronisation* als auch ein *Kommunikationsschema* auf Basis von software-basierter Cache-Kohärenz erarbeitet und untersucht.

Die Prozesssynchronisation setzt das Konzept der *general active target synchronization* aus dem MPI-Standard um. Ein existierendes Klassifikationsschema für dessen Implementierungen wird erweitert und zur Identifikation einer geeigneten Klasse für die nicht-kohärente Plattform des SCC verwendet. Auf Grundlage der Klassifikation werden existierende Implementierungen analysiert, daraus geeignete Konzepte extrahiert und ein leichtgewichtiges Synchronisationsprotokoll für den SCC entwickelt, das sowohl gemeinsamen Speicher als auch ungecachete Speicherzugriffe verwendet. Das vorgestellte Schema ist nicht anfällig für Verzögerungen zwischen Prozessen und erlaubt direkte Kommunikation sobald beide Kommunikationspartner dafür bereit sind. Die experimentellen Ergebnisse zeigen ein sehr gutes Skaliserungsverhalten und eine fünffach geringere Latenz für die Prozesssynchronisation im Vergleich zu einer auf Nachrichten basierenden MPI-Implementierung des SCC.

Für die Kommunikation wird mit SCOSCo ein auf gemeinsamen Speicher und software-basierter Cache-Kohärenz basierenden Konzept vorgestellt. Entsprechen-

de Anforderungen an die Kohärenz, die dem MPI-Standard entsprechen, werden aufgestellt und eine schlanke Implementierung auf Basis der Hard- und Software-Funktionalitäten des SCCs entwickelt. Trotz einer aufgedecktem Fehlfunktion im Speichersubsystem des SCC kann in den experimentellen Auswertungen von Mikrobenchmarks eine fünffach verbesserte Bandbreite und eine nahezu vierfach verringerte Latenz beobachtet werden. In Anwendungsexperimenten, wie einer dreidimensionalen schnellen Fourier-Transformation, kann der Anteil der Kommunikation an der Laufzeit um den Faktor fünf reduziert werden. In Ergänzung dazu werden in dieser Arbeit Konzepte aufgestellt, die in zukünftigen Architekturen, die Cache-Kohärenz nicht auf einer globalen Ebene des Prozessors liefern können, für die Umsetzung von software-basierter Kohärenz für einseitige Kommunikation hilfreich sind.

Acknowledgements

At this point, I would like to express my thanks to some people who supported me in the process of creating this thesis. First of all, I thank Bettina Schnor for supervising me in the past years and for the vivid and fruitful discussions we had. Second, this thesis would not have been possible without the provision of an SCC system by Intel. For their support on SCC matters, thanks go to Werner Haas and Michael Riepen, Intel engineers at the time. My colleagues from the sun deck at the Institute for Computational Science deserve credits for the nice time we had together in the past years: I want to point out Jörg Jung and Sven Schindler, who shared the office with me from time to time, but who also caused (sometimes unproductive, yet welcomed) distractions when they stayed in the room. The latter applies to Marius Lindauer as well. I do not want to forget Klemens Kittan to thank him for his friendly and infrastructure support, like shutting down and turning the SCC on again due to reoccurring, nasty and campus-wide power shutdowns. Thanks goes also to Sebastian Menski for his experimental support in the very early stages of the thesis, and to Martin Ohmann for his work on the FFT benchmark during his diploma thesis. In addition, Sven Schindler and Susi Kirschbaum have to be mentioned for their individual support in language matters.

Contents

1 Introduction

Microprocessors and the main memory attached to them are the essential building blocks of computer systems. Over the last decades, the processor performance continuously increased by several orders of magnitude [14, p. 3]. In contrast, the performance of the main memory, which provides data to the processor, could not be improved that much and therefore created a large performance gap between these two essential components. To compensate that gap, several layers of *caches* — fast intermediate memories — were added between the processor and the main memory [14, pp. 72 ff.]. They hold recently used data that is accessed by the processors and hide the large access times of the main memory. Keeping the cache utilization as high as possible is critical to fully exploit the processor's processing capabilities [1, 15].

While several techniques for improving instruction processing contributed to the performance gains of processors, the clock rate has been the main driver. However, technical limits prevent further upscaling. In fact, the frequency of microprocessors stagnates since about 15 years as shown in Figure 1.1[1]. To further increase the performance, the usage of *multi-core processors* became common at that time and they are ubiquitous by now. In this type of processors, the data path and control unit as well as caches are replicated and grouped in *cores* (see Figure 1.2). This enables parallel execution of programs on a single processor.

Besides the replication of some processor resources, other components are used by multiple cores. Usually, the cores are connected via an interconnect and share the main memory, although the memory might be distributed (see Figure 1.2). This enables programs to share data via the commonly used memory. The usage of this *shared memory* and the caches at the same time imposes an important problem. When a process accesses data, the information will be transparently saved inside the caches attached to the executing core. Thus, copies of the data from main memory are created. If more than one core accesses data in the shared memory, potential copies in the caches must be identified and an eventual modification must be handled such that

[1] Additional data compared to [14, Fig 1.11] obtained from http://ark.intel.com/products/93742

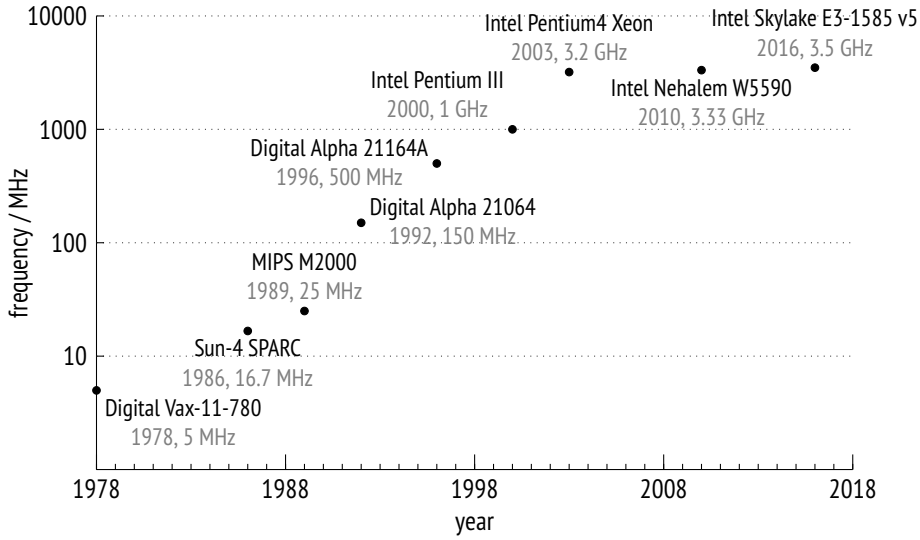


Figure 1.1: *Growth of processor frequency since 1978 (based on [14, Fig. 1.11]).*

each process observes the most recent data. This task of maintaining the correct state of cached data is known as *cache coherence*.

1.1 Motivation

Over the decades of processor design, several techniques were developed to improve the performance of a single processor core. Today's architectures are able to issue multiple memory accesses per clock cycle and also can load several machine words in parallel, e.g. to fetch data into their vector registers. If a data access causes a cache miss, i.e. the referenced item is not stored in the local caches, other caches must be checked for copies which are potentially modified. Similar, if a local write to shared data is encountered, other caches must be informed about that change. This causes traffic on the interconnect to keep the cached data coherent.

It has been shown that coherence traffic of so-called snooping protocols can exceed the capabilities of bus-based interconnects [16, 17]. A shift from pure snoop-based protocols, which require broadcasts [14, p. 364], to directory-based or mixed coherence schemes as well as different interconnects, such as point-to-point, ring, or mesh topologies, made multi-core processors less prone to contention by coherence traffic.

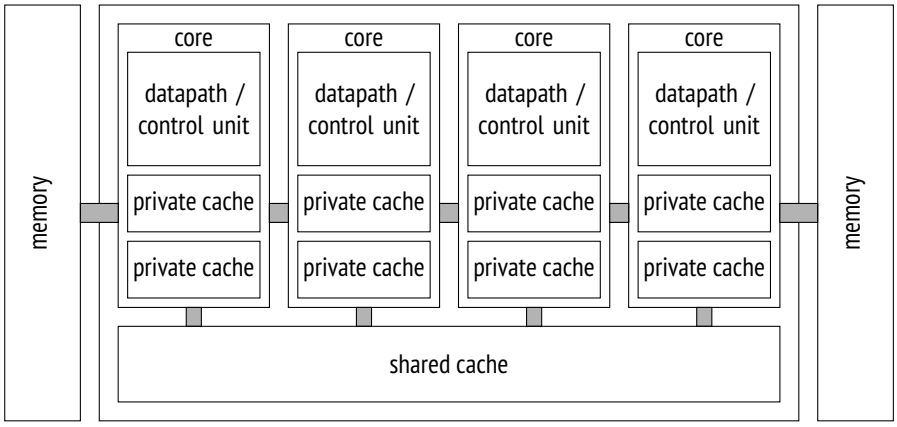


Figure 1.2: *Schematic overview of a multi-core CPU with shared last level cache.*

However, even on those architectures the implementation of coherence adds latency to memory accesses requiring a deep understanding to achieve good performance [18].

In addition to the challenge of efficient programming of large-scale coherence systems, two observations can be made.

First, advances in memory technology led to the integration of fast high-capacity on-chip memories that assist conventional main memory. This type of memory provides high bandwidth to the cores that share the processor die. However, the high bandwidth makes building a coherent multi-socket system practically impossible, because current coherent interconnects do not provide the required bandwidth to serve the coherence traffic. For the *Knights Landing* processor, a memory bandwidth of more than 400 GB/s is stated. Even if the coherence traffic is reduced, it exceeds the capacity of the QuickPath Interconnect (uni-directional bandwidth of 19.2 GB/s) which is used to provide coherence in systems with lower memory bandwidths. [19]

Second, other architectures intentionally break with a support of cache coherence. The Intel Single-Chip Cloud Computer [20] provides no coherence at all and enables the research for many-core processors without that feature. Further, HPE's *The Machine* [21] or the *EUROSERVER* project [22] are designed to consist of multi-core processors but do not provide a globally coherent system. While those two systems are still based on shared-memory, they provide coherence only inside a subdomain, or island, consisting of multi-core processors. Thus, research of processor producers as well as designs of system builders consider the absence of coherence.

As a result of these two observations, non-cache-coherent architectures based on shared memory need to be considered. Programming models that use shared memory and do not require transparent hardware-based cache coherence have to be regarded as well. One-sided communication of the *Message Passing Interface (MPI)* is such a model and is therefore focused within this thesis.

1.2 Outline

Chapter 2, provides background information which is required for the subsequent discussions. Cache coherence and its limitations are discussed. The chapter also presents the Intel *Single-Chip Cloud Computer (SCC)*, which is used as non-cache-coherent many-core platform, as well as a short overview on programming models for many-core processors. This also covers one-sided communication which is investigated within this thesis.

In Chapter 3, efficient synchronization for one-sided communication on the SCC is discussed. Therefore, an existing classification of synchronization schemes is extended. Implementations of the MPI general active target synchronization are then surveyed according to this extended categorization. The survey also discusses the suitability of the implementation for the SCC. Based on this analysis, a synchronization protocol is designed and evaluated on the many-core chip to confirm its effectiveness.

Afterwards, Chapter 4 addresses the communication on the SCC in absence of hardware-based coherence. The design of a software-based cache coherence protocol for MPI's one-sided communication that relies on shared memory is derived. The chapter presents fundamental requirements according to the MPI standard, discusses the efficient implementation on the SCC, and shows the results of the experimental evaluation. In addition, beneficial features for future architectures are proposed.

Finally, Chapter 5 summarizes and discusses the results presented in the previous chapters. With a look at future work, the thesis is concluded.

1.3 Contributions

The main contributions of this thesis are the following:

1. The concept, implementation and evaluation of a synchronization protocol for one-sided communication on a non-cache-coherent many-core architecture is presented. The concept is based on shared memory, but does not require cache-coherence. In the evaluation it is demonstrated that the implementation of the proposed concept is efficient and provides much lower latency compared to a tuned message-based solution.
2. The SCOSCo approach, a software-managed cache coherence solution for one-sided communication on non-cache-coherent many-core processors, is designed. Requirements for such an implementation with respect to the MPI standard are identified. Based on these, a suitable implementation for the Intel SCC is derived and an evaluation is provided. Both micro-benchmarks as well as a stencil application and a communication-intense three-dimensional Fourier transform confirm the efficiency of that approach with up to five times reduced communication share on the overall runtime.
3. From both contributions above, consequences for the possible future CPUs are derived. Beneficial features are identified which should be included in future systems to support one-sided communication on shared-memory system where cache coherence is not available.

In addition to these major contributions, the following minor ones evolved

1. The existing classification of implementation options for the synchronization of one-sided communication is extended by a new class. (see Section 3.2). The additional class closes the gap between literature and implementations.
2. A survey on existing MPI libraries with a focus on general active target synchronization is carried out. The implementations are classified using the extended scheme. Based on its outcome, general concepts of such implementations are identified (cf. Section 3.3).
3. As a result of the survey, a conceptual error in the stable version of the well-known MPI library Open MPI is identified. The bug has been reported to the community and was confirmed as well. See Section 3.3.3 for details.
4. Multiple implementation errors in the original MPI library for the Intel SCC are identified that prevent the usage and evaluation of MPI one-sided communication with this library or any one-sided MPI application (see Section 4.1.2).
5. A (possible) bug of the SCC's hardware is identified. The hypothesis is supported by an involved Intel engineer (cf. Section 4.4.2).

1.4 Publications

The majority of the above contributions has been reviewed and published in the following publications and was presented at the according conference workshops and symposia.

- Steffen Christgau and Bettina Schnor. „One-Sided Communication in RCK-MPI for the Single-Chip Cloud Computer“. In: *MARC Symposium*. Ed. by Eric Noulard. ONERA, The French Aerospace Lab, 2012, pp. 19–23.
- Steffen Christgau and Bettina Schnor. „Software-managed Cache Coherence for fast One-Sided Communication“. In: *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. Ed. by Pavan Balaji and Kai-Cheung Leung. ACM, 2016, pp. 69–77. DOI: 10.1145/2883404.2883409.
- Steffen Christgau and Bettina Schnor. „Synchronization of One-Sided MPI Communication on a Non-Cache Coherent Many-Core System“. In: *ARCS 2016 - 29th International Conference on Architecture of Computing Systems, Workshop Proceedings, April 4-7, 2016, Nuremberg, Friedrich-Alexander University, Erlangen-Nürnberg*. Ed. by Ana Lucia Varbanescu. VDE Verlag / IEEE Xplore, 2016.
- Steffen Christgau and Bettina Schnor. „Exploring One-Sided Communication and Synchronization on a non-Cache-Coherent Many-Core Architecture“. In: *Concurrency and Computation: Practice and Experience* 29.15 (2017). DOI: 10.1002/cpe.4113.

1.5 Scope

This thesis addresses the domain of high performance computing. As a consequence, general purpose parallel programming environments, like Thread Building Blocks, Cilk, Go Routines of the Go Language, etc. are not covered. Further, the context of this thesis are many-core processors. Although there is no commonly shared strong definition, it is assumed that many-core processors are high-end or research chips with a significantly higher number of cores than found in commodity consumer-level products. In addition, the thesis is restricted to chips with physically shared memory, and fully coherent chips or systems are generally not in its scope. Also, this thesis focuses on the design of software, especially middleware that relies on hardware features, but does not address the hardware implementation of processors.

1.6 Conventions

In this thesis the following conventions are employed.

1. All physical quantities are considered to be a product of a number and a unit. That is, a quantity divided by its unit gives the quantity's value. As a result, expressions like *"latency / µs"* are used in plots or tables to indicate that the following given numbers should be considered as µs.
2. Units of quantities are expressed with SI prefixes which are based on powers of ten. However, when a quantity is based on the unit Bytes, the prefixes k, M, G, and T have to be considered as powers of two. That is, $k = 1024, M = 1024^2$, and so forth.
3. Bytes are abbreviated as B and the small letter b denotes the unit Bit, not Byte.
4. Names of authors are displayed with small caps, such as in HOARE.
5. Source code is typeset with a mono-spaced font, like `printf`. Routines from the MPI standard are generally written in upcase letters, like `MPI_BARRIER`, matching the standard's convention. In listings of real or pseudo code, they are shown according to the appropriate language binding, like `MPI_Barrier` when `C` is used.

2 Background

In this chapter, the architectural concepts of many-core processors are briefly introduced. Also, limitations concerning the memory subsystem with a focus on caches are covered. Those motivate to focus on non-cache-coherent systems, like the Single-Chip Cloud Computer. In the progress, different programming models and their suitability for non-cache-coherent many-core processors with shared memory are discussed, especially the one-sided communication model and its realization within the Message Passing Interface standard.

2.1 Shared Memory Processors

In the domain of *high performance computing (HPC)*, multi-core systems with shared memory are essential building blocks of computer systems. They consist of multiple cores which are connected via some type of interconnect. For performance reasons, memory controllers are integrated into the processor in most cases. The design allows all cores to access the memory attached to the processor, i.e. the processor provides shared memory. As depicted in the introduction, caches are used to bridge the performance gap between the raw processor performance of the cores and the much slower main memory. Figure 1.2 illustrates an exemplary design with private and shared caches. Either type can be moved closer or farther away from the cores in other designs. VAJDA [24, pp. 9. ff] as well as RÜNGER AND RAUBER [25, p. 21] describe different design variants and the concepts in more details.

Techniques, like instruction pipelining, out-of-order execution, or superscalar designs, improved the raw performance of the individual processor cores. Nowadays, the majority processors are able to complete more than one memory operations per cycle. A core of a contemporary Intel Skylake processor can theoretically issue four memory operations per clock cycle. With a word size of 64 bits and a clock rate of 3.5 GHz (cf. Figure 1.1) this results in 112 GB/s of theoretical peak memory bandwidth per single core, not considering references to instructions. In a multi-core system with four

cores, this results in 448 GB/s of data traffic [14, p. 73]. Current DDR4-3200 memory with a frequency of 1.6 GHz delivers only 25.6 GB/s — barely enough bandwidth for a memory intensive application running on a single core.

Since a long time, however, caches exploit the fact that most memory references reoccur both in space and time. They buffer data which is likely to be reaccessed in a certain time frame as well as data that is close to the last referenced ones. The caches are faster than the main memory, but can only buffer a subset. Nevertheless, they provide the bandwidth that an application demands, as long as the principle of spatial and temporal locality is fulfilled by the application [14, p. 72 f.]. Thus, they are an essential component for contemporary processors. Efficient systems without caches seem to be hardly feasible. Details about cache design, replacement strategies, and optimizations can be found in HENNESSY AND PATTERSON [14].

In addition to single processors that house several cores, connections between those multi-core processors are possible. In general, each of the connected processor nodes provides memory to the system which is accessible to every core in the system. Since the access latency of memory in local and remote nodes differs, those designs are also denoted as *non-uniform memory access (NUMA)*. They also impose a challenge to the interconnect between the nodes because it has to sustain the bandwidth demands of the individual cores that potentially access remote memory. Note that also multi-core processors can be NUMA, depending on the actual design [25, p. 21].

2.1.1 Memory Consistency and Coherence

Even in a single multi-core processor, the presence of multiple cores and their caches imposes two different but related problems.

Memory Consistency

When processes in a multi-core system exchange data via shared memory, i.e. they perform load and store operations, the question arises when the result of those events become visible to the running processes. The model that describes this property of the memory system is the *memory consistency model* or *memory model* [26, p. 229]. It can also be regarded as set of rules which specify the allowed behavior of programs running on the multi-core processors [27, p. 21]. If these rules are followed, the

memory system ensures that programs generate correct results according to the model. The rule set generally defines, in which order memory operations are allowed to be executed.

The problem of maintaining consistency is subject to many publications reaching back to at least LAMPORT's publication on correct program execution on multi-processor computers [28]. Within his model of sequential consistency, LAMPORT defines that the execution order of memory operations has to follow the instruction order in the program. That is, no reordering of the stated operations is allowed to happen. As a result, any sequentially executed interleaving of the (different) programs that maintains the original instruction orders represents an outcome of the parallel execution. [26, p. 231]

More relaxed models loosen the ordering rule of the sequential consistency model. They allow load or store operation to be reordered. The *total store order* model, e.g., allows independent load operation to execute before stores, but it maintains the order between writes [25, p. 95 f.]. To some extend, this model applies to contemporary Intel x86 processors [29].

The *release consistency (RC)* model [30] allows the reordering of all memory operations, but only within a pair of synchronization operations, named *acquire* and *release*. A processor has to complete an acquire operation before subsequent loads or stores are performed, but the order of independent accesses is unspecified. Completion of those accesses is enforced by the release operation. No reordering of acquire and release operations is allowed within this model.

For more details, GHARACHORLOO ET AL. [30] provide an overview of consistency models for shared memory multiprocessors that is extended by the survey of MOSBERGER [31].

Cache Coherence

The discussion of memory consistency ignores the presence of caches. This is valid, since from a programmers point of view they are transparent and, in fact, consistency is also an issue without caches. However, caches are critical components and need to be considered because they are used to implement the consistency [27, p. 21]. With multiple cores and their caches, the task arises to keep the caches up to date in order to avoid inconsistencies due to actions of other cores. This is the task of *cache coherence*. Coherence protocols are used to solve this problem.

Those protocols assign states to the cache lines. The lines represent the buffered subset of the main memory. The state describes the level of sharing inside the system. Coherence protocols define automata to describe the transitions between those sharing states. The set of the modified, exclusive, shared and invalid state is used by the probably most prominent protocol, *MESI*. Inputs to the automata are memory operations observed by the caches. Those are events from the local core as well as events issued by other ones. For example, an observed load on an exclusive cache line causes a transition into the shared state. Details of those protocols can be found in Chapter 7 in the book of SORIN, HILL, AND WOOD [27].

To observe operations by other cores, two different approaches are discussed in the literature: *snooping protocols* and *directory protocols*.

For snooping protocols, each cache maintains the state of a cache line. The cache observes, or snoops, memory operations by the local and other cores to change the cache line state according to the protocol [14, p. 354 ff.]. All memory actions issued by cores to their caches are broadcast to all other caches in the system. The observed operations serve as input to the protocol's automaton and cause according changes in the individual cache line states.

In directory protocols, the state is kept in a directory and not replicated in every cache of the system. Therefore, only the directory needs to be consulted when a memory operation is issued by the cores. The directory performs further steps like fetching the data from memory, requests data from other caches, collects the responses, resolves conflicts, and replies to the requesting cache. Thus, it relies on point-to-point messages with requests and replies rather than broadcasts.

The employed directory can be centralized. To avoid a bottleneck, it can be distributed as well. In that case, a (configurable) hash function can be used to identify the responsible directory based on the memory address [14, 32, 33]. In addition, the directory keeps track which caches contain copies of a memory block. Usually, a bit vector is employed to precisely track which caches own a copy. As a result the number of additional bits per cache line in the directory scales with the number of caches (or cores) within the system [14, p. 380].

A mix of the distributed and snooping protocol variants is also possible [14, p. 363 f.]. For example, for subdomains of the system a snooping protocol can be used. In case of a miss inside that subdomain, the directory protocol is employed to contact the directory and ask for the originally requested memory block. To find copies or invalidate cached data, the protocol may send requests, i.e. it snoops, on point-to-point links and waits for replies on order complete the operation [32, 34].

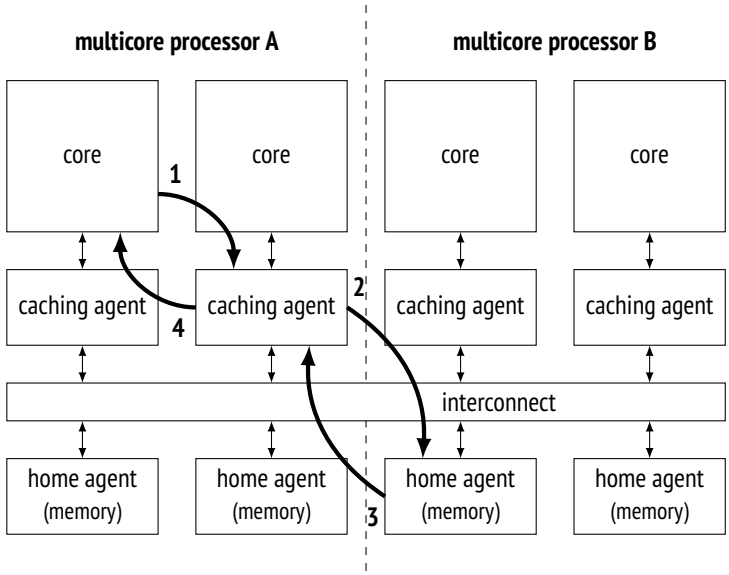


Figure 2.1: *Messages sent in a QuickPath-connected system of multi-core processors.*

Figure 2.1 illustrates the messages that are sent in a system of multiple multi-core processors with distributed shared memory and a mixed protocol, found in systems with Intel's QuickPath Interconnect technology [32, 34]. The diagram shows a message sequence that occurs in case of a local cache miss. In this case, a core sends a request to the cache controller (caching agent, CA) which is responsible for the requested memory address and is part of the last level cache in the local processor. From there, the responsible memory node (home agent, HA) that holds the directory is contacted if the cache controller could not satisfy the request from the last cache level. In the example, the memory node replies with data from memory, but it could also request data from other cache agents in case the address is shared.

2.1.2 Limitations of Cache Coherence

Both snooping and directory protocols as well as mixed variants are used in multi-core and many-core processors.

Because snooping protocols require broadcasts, they often use a bus or a ring as interconnect since these are broadcast media and, therefore, well-suited for implementing those protocols [14, p. 356]. The bus' bandwidth needs to scale with the number of

attached cores in order to provide them enough memory bandwidth. In fact, this is the limiting factor. As indicated above, a single core can easily require several GB/s of memory bandwidth. However, this exceeds the capacity of buses [14, p. 379], and such an approach will not scale for much more than eight cores [14, p. 363] or even connected multi-core processors, which has been demonstrated in experiments [16]. For that reason, such protocols are only used in systems with lower core counts or in systems with mixed protocols [25, p. 89, 32], where the last cache level of a multi-core processor is used as broadcast medium [25, p. 85].

In larger systems, like multi-core multiprocessor systems or many-cores, either mixed protocol variants or pure directory protocols are used [32, 35]. Since they rely on point-to-point messages only and do not employ broadcasts, the demands on the interconnect are lowered. Nevertheless, the increasing core count imposes new challenges to both the network performance as well as the storage capacity of the directories.

As pointed out above, a directory has to maintain a list of caches that possibly share a cache line. This implies a scaling problem because one bit per core has to be added to each cache block state. Thus, storage requirements for the directory are increased and can limit the use of directory protocols in the field of HPC when implemented in that straight-forward fashion [14, p. 379]. However, MARTIN, HILL, AND SORIN propose that the list of sharing caches can be stored in a hierarchical manner to solve this problem [36].

Besides those storage concerns, an increased core count still requires that the interconnect provides enough bandwidth to the cores [25, p. 31], even in presence of the coherence protocol messages. Compared to memory accesses in systems without coherence, the traffic per cache miss increases by about 20% [36] due to additional messages (cf. [32]).

In addition, the access latency to memory is increased. Instead of requesting memory directly, e.g. in case of a cache miss, the directory needs to be consulted. To provide coherence, this compromise has to be accepted, but its overhead might be small compared to high memory latency. In addition when the set of sharing caches is tracked precisely, the coherence traffic can be minimized and kept constant for up to 1024 cores [36].

Based on these observations, it is unlikely that cache coherence disappears completely, but its realization apparently becomes difficult. ASHBY ET AL. [37] state that according designs are *"notoriously hard to verify"*. In addition, changes in technology may require design changes. Fast and large stacked on-chip memories provide much

higher memory bandwidths than conventional external memory: The integrated on-chip memory (MCDRAM) of the 72-core *Knights Landing* offers 450 GB/s, which is five times higher than the bandwidth of the chips conventional main memory. The processor uses a 2D mesh network, capable of handling 700 GB/s of traffic between the cores. It also provides cache coherence using a distributed tag directory [33].

Connecting multiple of these processors to build more powerful and coherent system is practically impossible. Coherent interconnects like QuickPath, with a maximum bi-directional bandwidth[1] of 38.4 GB/s [34] are not powerful enough to handle the coherence traffic even if a low share of coherence traffic is assumed [19, 33]. Even if one would connect the chips it might be done only in a *non-cache-coherent (nCC)* manner[2], but this requires different programming schemes compared to multi-threading applications using shared memory.

Other systems drop the support for cache coherence by design, but still connect processors by means of shared memory. They provide coherence only within a subdomain of the whole system. The proposed *The Machine* focusses on memory-driven computing, i.e. to bring computation near the data by providing a huge pool of memory to the compute elements [38]. While the compute nodes are still cache-coherent multi-core processors, the whole system itself is nCC [21, 39] making it similar to a theoretical connection of Knights Landing processors.

A similar design is observed in the *EUROSERVER* project. Here two multi-core processors are connected with a coherent interface building a *chiplet* with attached memory. Multiple chiplets can be connected using a multilevel interconnect, but it does not provide a cache coherence protocol between the linked multiprocessor nodes. However, cached load and store operations to remote memory are possible. [22, 40]

In summary, *"on-chip cache coherence is here to stay"* according to the publication of MARTIN, HILL, AND SORIN [36]. Regarding the above examples, however, limitations of pure hardware-based coherence become visible and trends to intentionally avoid this hardware feature are coming up. Thus, the investigation of nCC systems with shared memory, especially many-cores, is critical. With the Single-Chip Cloud Computer such a system is investigated in the remainder of this thesis.

[1] assuming an operating frequency of 4.8 GHz like on Haswell-EP platforms

[2] personal conversation with Avinash Sodani, Knights Landing chief architect, on April 5 at ARCS conference 2016, Nürnberg

2.2 The Intel Single-Chip Cloud Computer

The *Single-Chip Cloud Computer (SCC)* is an experimental CPU architecture developed by Intel [20] within the *Tera-Scale research program* [41]. It can be regarded as a successor of the 80-core *Teraflops Research Chip* [42] which is also known as *Polaris* [43].

With the Polaris processor, the SCC shares a tiled architecture. All tiles are connected by an on-chip network. However, the SCC's tiles are not based on a specifically designed architecture, but on the well known x86 or IA-32 instruction set architecture. This can be considered as a consequence of the limited programming capabilities of the Polaris' cores which were programmed by hand-crafted VLIW assembly instructions, offered no I/O features and did not provide an operating system [44]. Thus programming the Polaris chip was limited to only a few researchers who were able to deal with these limitations.

The purpose of the SCC is to investigate scalable many-core processor design as well as further hard- and software-oriented research linked with this kind of architecture, such as energy management and parallel programming. A crucial aspect of the design was to drop coherence between the caches of the cores and support parallel applications with hardware message passing features. [20]

2.2.1 Architectural Overview

In contrast to its predecessor, the SCC consists not of 80 but of 24 tiles which are arranged in a 6×4 regular grid as shown in Figure 2.2. Each of those tiles contains five components: two cores with internal and external caches, a *Message Passing Buffer (MPB)*, a *Mesh Interface Unit (MIU)*, and a configuration register block located in the latter unit. A technical in-detail description of those parts can be found in [45]. Based on that specification, the following sections describe the essential aspects of these five components which are relevant for the presented work.

To connect the tiles with each other and the external memory, the SCC possesses an on-chip mesh network which is based on five-port routers. The local port of each router is connected to a tile, the other four build up the two-dimensional mesh. The network is packet-based and uses a static dimension-ordered routing [46, p. 8 f.]. That is, packets traverse in x-direction first and are routed along the y-direction to their target afterwards.

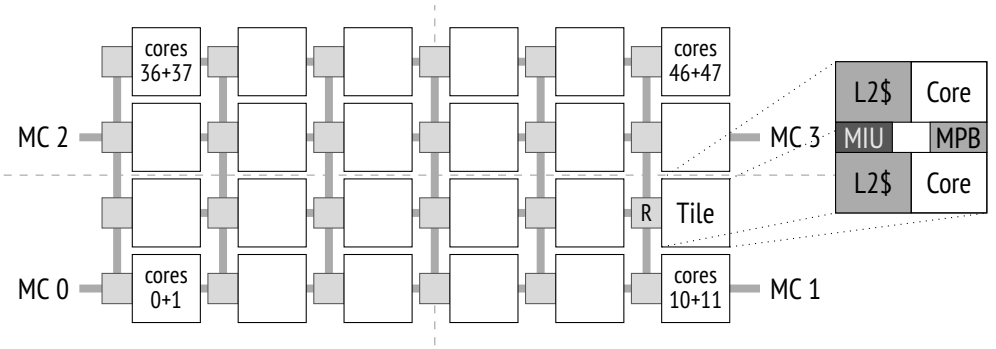


Figure 2.2: *Overview of the tiled SCC architecture.*

Further, the network attaches four DDR3 memory controllers to the chip. Six tiles with a maximum vertical distance of one and a maximum horizontal distance of two can be considered to belong to a memory controller domain (see Figure 2.2).

To access the chip, an external FPGA constitutes the system interface (SIF). An external computer is attached to the SIF-FPGA via PCI Express. This connection allows to inject and receive packets from the on-chip network. In addition, a voltage controller is integrated into the chip to enable dynamic frequency and voltage scaling. However, a discussion on energy-management is out of the scope of this work and further details are therefore omitted. Relevant research on this domain has been published in [47–49], among others.

Cores

The SCC cores are based on the 32-bit Pentium (P54C) architecture. Due the this legacy, the cores operate in-order. Instructions are executed in program order and are not dynamically reordered by the processor, e.g. to hide memory latencies as it is the case in contemporary architectures. In consequence, an SCC core stalls when a memory operation is issued but a preceding access is not yet completed.

Despite limited superscalar components in the Pentium design, the cores do not possess other features that enable parallelism at hardware level. There are neither vector units, multiple hardware threads, nor other instruction set extensions that can be found in contemporary processors. Thus, the SCC's performance is not comparable to such CPUs. However, this it not a critical aspect for the main purpose of the chip which is research.

2.2.2 Memory Subsystem

For the work presented in this thesis, the memory subsystem is essential and is therefore discussed in detail within this section. The descriptions are based on [20] and [45], if not stated otherwise.

Caches

Two 16 KB L1 caches are integrated in each core, one for instructions and one for data. Their size is doubled compared to the original Pentium architecture. A line in those caches is 32 Byte large. The L1 cache can operate in write-through or write-back mode, i.e. when cached data is written the memory will be updated immediately or when the modified line is evicted, respectively.

In addition to the two integrated L1 caches, a second level cache is externally attached to each core and closely located on the tile (see Figure 2.2). Each L2 cache has a size of 256 KB and contains both data and instructions. It can operate in write-back configuration only. The line size is equal to the L1 cache. Both cache levels to not allocate lines when a write-miss occurs. In consequence, data is only cached on read accesses.

A key point of the SCC's architecture is the omitted cache coherence. There is neither hardware-assisted coherence between the two caches on a single tile nor between any caches of the overall chip. If the coherence property is desired, it has to be managed in software.

Mesh Interface Unit, lookup tables and Address Translation

The L1 and L2 caches are connected to a cache controller via a Front Side Bus interface. In case of a cache miss, the cache controller forwards the memory access to the *Mesh Interface Unit (MIU)*. This unit transforms the information from the cache controller into a network packet.

During the transformation of the memory access, an address translation takes place. Its input is the 32-bit physical address that was handed over to the MIU by the cache controller. The output of the translation is a 36-bit *system address* along with routing

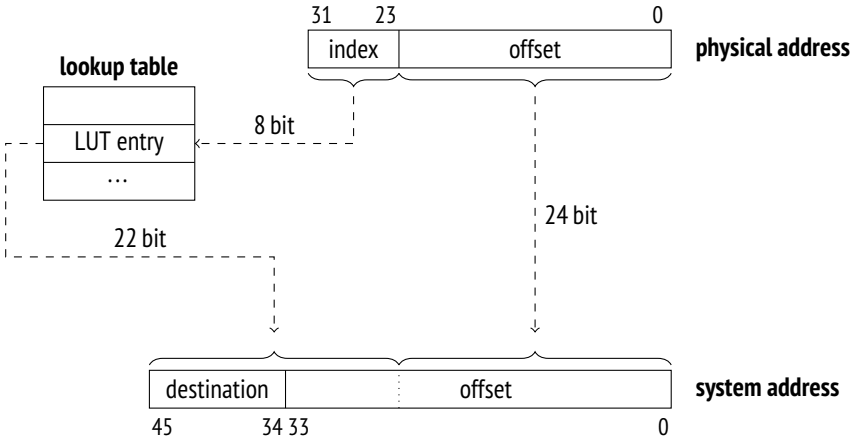


Figure 2.3: *LUT-based translation of physical to system addresses by MIU.*

information for the on-chip network. The translation is performed with the help of *lookup tables (LUTs)* as illustrated in Figure 2.3.

In the LUT-based translation procedure of the SCC, the upper eight bits from the physical address are used as index in a LUT. Each LUT entry is 22-bits large and contains network routing information such as the coordinates of the destination router/tile and the sub-target of the issued operation on that particular tile. The sub-target includes the four router ports as well as the tile's Message Passing Buffer and configuration registers that reside in the MIU. Since the DDR3 memory controllers are attached to router ports as well (e.g. on the west port of router at the lower left corner of the chip, see Figure 2.2), the external RAM is addressable too.

The location inside the given destination is specified by the 34 bits offset in the system address. It is constructed from 10 bits of LUT entry and the remaining 24 bits of the physical address. Regarding the physical addresses, the LUT mechanism consequently divides the addressable memory into $2^{24} = 16\,\text{MB}$ large pages which are identified by the data in an LUT entry.

The LUT-based address translation inside the MIU has several consequences:

1. The LUT mechanism provides access transparency. Independent of the addressed location, the required actions on software level are the same to perform a memory operation. This is even true on a machine instruction level as the translation happens outside the cores.

2. To address a specific location in the system from software, the physical address must be given. This is difficult when an operating system is used, which usually prevents direct access to a specific physical memory location when virtual address spaces are used.

3. The LUTs enable the definition of *shared memory*. If more than one LUT contains an entry with the same data, then different cores can access the same memory region. However, care has to be taken when accessing these shared memory areas due to the lack of hardware-based cache coherence.

The last consequence is further emphasized by the fact that a core's LUT resides in the configuration register block of the according tile. Accordingly, they are addressable by memory accesses and are writeable as well. The LUT configuration can be changed dynamically and memory can be easily turned into shared or private one at runtime.

The dynamic reconfiguration of the LUTs implies very little overhead since it just involves writing to the configuration registers. The written changes take effect immediately. However, effects of caching must be accounted carefully during reconfiguration since the caches operate with physical addresses. If a previously cached line is evicted after reconfiguration, it will be stored in the current destination configured by the LUT. It will not be stored in the position it was originally loaded from.

The concept of the address translation from a physical to a global (system) address is not special to the SCC. For example, the shared physical address space machine Cray T3D [50, p. 470 ff.] from 1993 used a similar scheme, where bits from the physical address are used as index for the reconfigurable *DTB annex* register set [51, p. 3-17 ff.]. Therein, the destination in the system for memory accesses is identified. Similar to the SCC, it is the software's responsibility that an access to a virtual address ends up at the right physical address and thus at the intended (remote) memory location.

Note that depending on the LUT configuration, the SCC can be classified differently. As long as no shared memory is defined, the cores operate independent of each other on isolated address spaces which makes the SCC a distributed system. That is why the SCC is also often referred to as an on-chip cluster. Contrarily, when shared memory is enabled by the LUT settings, the chip can be considered as a tightly coupled. Combining both aspects, the SCC can be considered as hybrid system [52].

2.2.3 Message Passing Buffers

In addition to the cores and the caches, a 16 KB large SRAM memory, called *Message Passing Buffer (MPB)*, is placed on each tile and is connected to the MIU as well. In total, there are 384 KB of MPB memory available on the chip. A primary advantage is its lower latency compared to the external DRAM. An MPB access on a local tile only involves the MIU and the network[3]. It is three times slower than the L2 cache (18 core clock cycles) [46, p. 9]. Accessing remote MPBs adds further network hops. However, it does not involve overhead by the memory controllers. Thus, the on-die MPBs are still faster than the external memory. Additionally, due to the LUT mechanism, DDR and MPB accesses are transparent for software and can therefore be used in the same manner. Despite its name, the MPB can be used as general purpose memory. Thus it is sometimes refererred to as local memory buffer. Since, the term MPB is more common, it used in the remainder of this thesis.

2.2.4 Memory Types

As depicted in Section 2.2.2, the physical address controls the destination of a memory access. This is because the physical address is used as input for the translation and LUT lookup, respectively. Anyhow, the physical address itself is the result of another translation process that converts a virtual address used by programs into a physical one. The page tables involved in this virtual-to-physical translation define the behavior of memory accesses in the system, i.e. the memory type.

The memory type is defined by bits in the page table entries. Generally, the page table settings affect the cache behavior for the (usually) 4 kB large memory pages. It is important to understand, that the memory type setting is independent of the destination setting. Both settings are applied in different stages of the overall translation process that converts virtual addresses into SCC system addresses. This is illustrated Figure 2.4. The following sections describe which memory types can be configured by the page tables as well as their implications.

For the discussion, three bits of the page tables entries are relevant. The first one, named PCD, controls caching. If this bit is set, the memory inside that page is not cached. If it is unset, the usage of the second important bit, the PWT bit, is useful.

[3] In principle, network accesses should not be required to access local MPBs, but a documented hardware bug prevents a network bypass which was originally intended by the designers.

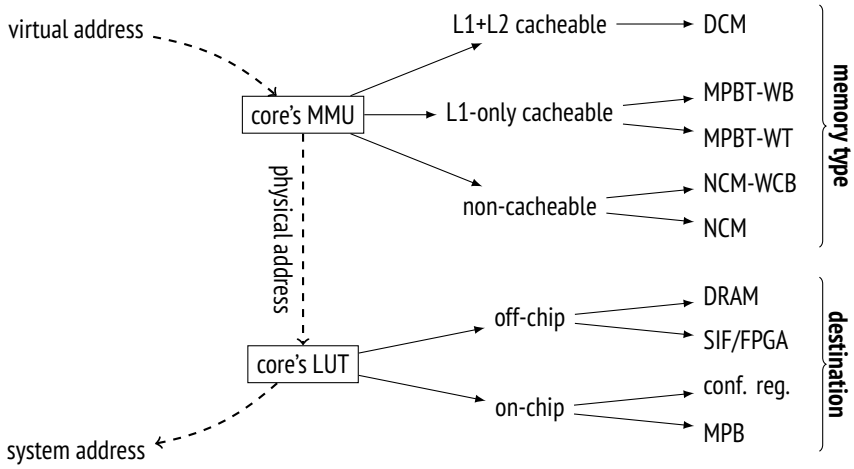


Figure 2.4: *Address translation mechanisms involved in memory accesses on the SCC.*

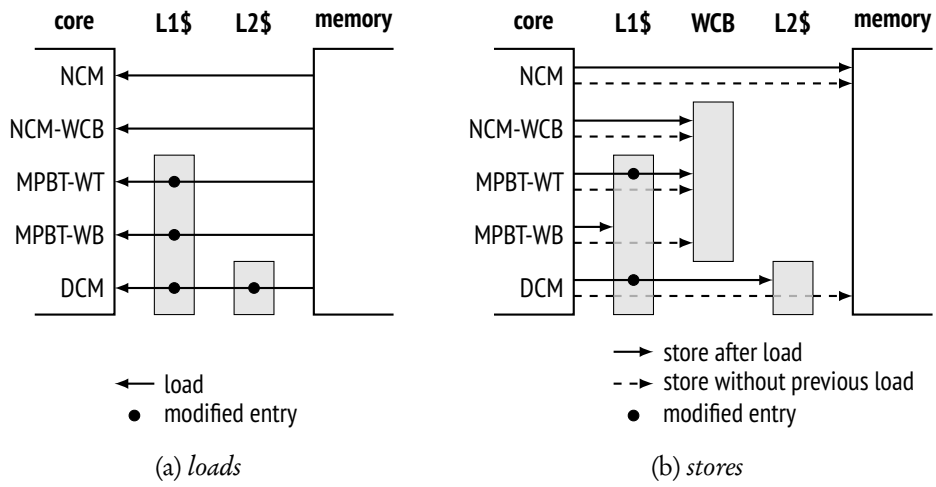
This one controls the L1 cache mode. As it abbreviation indicates, a set PWT bit configures the L1 cache as write-through.

The third important bit was originally reserved in the Pentium page tables. For the purpose of the SCC's it was redefined and serves as utility for message passing. Its designation is PMB and it disables the L2 caches for pages having the PMB bit set in the page table entry. Thus, read accesses to such memory are cached only in the L1 cache. Further, the PMB bit enables a *write combine buffer (WCB)*. This component can hold up to 32 write operations to subsequent bytes starting from an offset that is a multiple of 32. When another write operations is issued, the buffered writes are written to memory. The WCB is also flushed when it gets a write operations that does not fall in to the same 32 byte offset as the previously buffered accesses.

The combinations of those bits define different memory types which are discussed in the next paragraphs. Figure 2.4 gives an overview of the address translation processes, the discussed memory types and the possible destinations of memory accesses.

Cacheable Memory

When the page table entry's PCD bit is cleared, and the two other discussed bits are not set as well, all cache levels are activated. Consequently, a load operation fills cache lines in the L1 and L2 cache. Subsequent stores modify the cache lines and the modified



(a) *loads* (b) *stores*

Figure 2.5: *Semantics of load (a) and store (b) operations of the SCC's memory types.*

data is written back only when a cache lines is replaced. This is the usual memory that is exposed by an operating system to user space processes. Since the caches are involved with this memory type, it is called *definitely cacheable memory (DCM)*.

Combinations of DCM with the other two bits are hardly useful. PWT will cause the L1 to be write-through but since the L2 cannot be configured is such, it keeps the write-back semantics. Thus, writes will hit the L2 if the writes are issued on already cached data. Using the PMB bit disables the L2 which contradicts the DCM semantics.

Non-Cacheable Memory

The contrary memory type to DCM is the *non-cacheable memory type (NCM)*. Here, the PCD bit is set which disables all caches, regardless of the other bits' setting. Thus, a read operation will always access the addresses memory, not the cache. Similar, writes go directly the memory, if the PMB bit is not set.

If the PMB bit is set, caches are still disabled, but theWCB is activated. Thus, writes can be buffered. This can be beneficial, if performance is an issue. Since an SCC core stalls when more than one memory access is in flight, it will observe long stalls when frequent accesses to NCM are performed which involve two traversals of the on-chip

network and latencies of the memory controllers. The WCB can be used to alleviate the time the core stalls. To differentiate the NCM with activated WCB from the plain NCM memory type, it is designated as NCM-WCB.

Message Passing Buffer Memory Types

The PMB bit in the page tables activates a memory type that is unique to the SCC. As stated above, when this bit is set and caching is not completely disabled (see previous section), but only the L1 cache is activated. When an L1 cache line is filled due load operations on such memory, the line is marked by setting an additional status bit. Thus, cache lines originated from pages with set PMB bit can be identified by the cache's circuitry.

This feature is exploited by the only SCC-specific extension to the instruction set of the Pentium cores, the `CL1INVMB`[4] instruction. The operation invalidates all lines with this bit and also clears the message buffer status bit on each line.

Due to the invalidation, cached and modified data is not written back but gets lost. A flush operation for these specially marked lines is not present in the instruction set. However, the privileged `WBINV` instruction writes back all L1 cache data and also invalidates all lines in the buffer.

The feature of fast invalidation of the marked cache lines is heavily connected with the MPB and supports fast on-chip message passing. Consequently, memory with the PMB bit set is referred to as *Message Passing Buffer type (MPBT)*. A MPB can be efficiently polled with this memory type for new data using a sequence of `CL1INVMB` and read operations from the MPB. To write data in the MPB it must be ensured that it is not stuck in the L1 cache due to previous reads from the MPB. In such a case, the `CL1INVMB` instruction has to precede write operations.

Since MPBT memory is cached only in the L1 cache, the memory type has to be distinguished in MPBT-WT and MPBT-WB, depending on the cache configuration that is controlled by the PWT bit. The WT variant has the general advantage that writes to cached data are also propagated to the targeted memory.

However, the MPBT memory activates the WCB. Consequently, writes to MPBT-WT memory modify the cache (if a cache-hit occurs) and are buffered by the WCB.

[4] the mnemonic presumably stands for Cache Level 1 Invalidate Message Buffer lines

Hence, those operations materialize in the main memory only when the WCB is flushed. As illustrated in Figure 2.5b, the WCB can be thought as being placed behind the L1 cache. This also means, that writes to a cached line of MPBT-WB memory reach neither the WCB nor the addressed memory location as long as no flush of the written cache line is provoked. In absence of a cache flush based on addresses, there is no efficient way to achieve this goal.

2.2.5 Configuration and Atomic Registers

As depicted in Section 2.2.2, the MIU allows addressing of MPBs, memory, and configuration registers. This also includes the LUTs which are writeable as well and are part of the configuration register set. In addition, the core's hardware status information, thermal sensors, the tile frequency configuration, and the cache configuration are also available as memory mapped registers. The set also includes the tile ID register, which allows a tile's core to determine the tile's coordinates (x and y) and its own number z inside the tile (either `0` or 1). Thus, the global core ID can be computed.

Aside the mentioned registers, the MIU also provides a single atomic register per core that is denoted as *test-and-set register (TSR)*. Such a register can be used for synchronization purposes in software. On usual IA-32 processors, the legacy `LOCK` machine instructions prefix from the Pentium architecture is used in such cases. On bus-based architectures it locks the whole memory bus and prevented other participants (cores, e.g.) to perform memory operations. Thus, an atomic modification of a memory location is possible. In more recent architectures, the cache coherence protocol handles the `LOCK` prefix. Since cache coherence is missing on the SCC, the usage of the `LOCK` prefix is of no benefit. Even if uncached memory accesses are involved (see previous section), the implementation of a global memory lock operation seems to be challenging, not scalable, and prone to high latencies.

In contrast, the TSRs provide a minimalistic mean to implement synchronization. Initially, a value of one can be read from such a register. However, it atomically switches its value such that subsequent reads return zero. Hence, if a process reads a value of one it can be interpreted as having obtained the mutual exclusive lock. To release that lock, a write operation has to be issued to the register. The written value is ignored, but usually zero is used to indicate a release. After such a write operation, the register again returns the value of one for the very next read operation. To access the memory mapped TSRs in the MIU and obtain the described synchronization semantics, non-cached memory is the only option to do so. Cached operations would

either prevent getting the most recent data from the TSR or would most likely be caught by the cache when attempting to release the lock.

When multiple processes running on different SCC cores need to synchronize by means of a single TSR the on-chip network is automatically involved. Since this is done by a memory access which itself underlies NUMA characteristics, unfairness occurs when cores on different tiles compete for the TSR [53]. This is especially the case, when the TSR is contented, like for spin-locks.

To work around the limited number of TSRs and their mutex-only semantics, the system's FPGA-based chipset has been extended and subsequently provide 96 *atomic increment registers (AIR)*. Those offer semaphore behavior and atomically increment on reads (and return the old value), respectively decrement on writes, and can be set up with associated initialization registers [54, p. 11 ff.]. However, compared to the TSRs, the latency of the AIRs is four times higher and unfairness is again inherited from the on-chip network. Further, contention needs to be considered as every AIR access is routed through the system interface [55].

2.2.6 Software Environment

On top the described hardware features, different software packages are available for the SCC. Since the cores are based on the Pentium architecture, existing development toolchains, i.e. compilers, linkers, as well as low-level libraries can be reused. This enables compilation of most known software that supports the x86 architecture. Compatible software ranges from shells, basic command line utilities and scripting languages (Python, Perl) to database systems (MySQL) and web server software (Apache).

Linux for the SCC

To run those user-space applications an operating system is required. For this purpose, the Linux kernel was initially made SCC-compatible by Intel Labs as a customized version that was bundled along with user-land applications as *rckOS*. Subsequent work by SOBANIA, TRÖGER, AND POLZE improved the portability of the Linux kernel [56] which resulted in the creation of *sccLinux*[5]. Both versions have in common, that each core of the SCC runs a single instance of the operating system. This is mainly

[5] https://github.com/hpi-scc/sccLinux, last accessed 2016-04-25

due to the nCC architecture of the SCC. Unless uncached memory is used, the Linux kernel must be made ware of the missing cache coherence, especially for shared kernel data structures. Although there are implementations of other operating systems [57], bare-metal application frameworks [58], and hypervisors [59, 60], which enable to run a single operating system instance (including Linux) on top of all SCC cores, this work focuses on the case of 48 individual instances.

Default lookup table Configuration

To support booting and running 48 individual Linux instances, the default LUT configuration is designed such that each core (and thereby each Linux instance) gets an equal amount of private memory. That is, only a single LUT has an entry that addresses a specific portion of the system's RAM.

In case of the SCC hardware available for this work, 32 GB of RAM are available. With a system page granularity of 16 MB (cf. Section 2.2.2), each core is assigned an amount of $\lfloor 32\,\text{GB}/48/16\,\text{MB} \rfloor \times 16\,\text{MB} = 672\,\text{MB}$ RAM. This equals 42 entries in an LUT. Further, the LUT configuration ensures that the private memory of each core is located in the most closest memory controller. Consequently, each controller takes memory requests from 6 tiles or 12 cores respectively (cf. Figure 2.2 on page 17). The hardware setup ensures that all memory controllers are equipped with the same amount of RAM, i.e. 8 GB (8192 MB) per controller.

In addition to the private memory entries, the default LUT configuration includes entries to access all MPBs and configuration registers which are per-tile entities (see above) and thereby allocate $2 \times 24 = 48$ LUT entries in total. Two additional entries are reserved for the voltage and frequency controller and the management interface. A summarized layout of the default LUT configuration is shown in Table 2.1.

Legacy Shared Memory

In this setup, $256 - 42 - 48 - 2 = 164$ LUT entries are not used by private memory or other resources. In addition, the amount of private memory per memory controller ($12 \times 672\,\text{MB} = 8064\,\text{MB}$) leaves 128 MB free for further usage. The default LUT configuration defines one entry (16 MB) per controller that uses this spare area. As the according entries are replicated for all cores, this portion of the system memory is shared and is denoted as *legacy shared memory (LSM)*.

Table 2.1: *Summary of the default LUT layout for an SCCsystem with 32 GB RAM.*

Entry No.	Entries	Purpose
0 – 40	41	private memory
128 – 131	4	legacy shared memory
191 – 215	24	message passing buffers
224 – 247	24	tile configuration registers
250 – 251	2	voltage controller and management
255	1	private memory
all others	160	unused

Kernel Device Driver Support for the SCC

To access any shared memory region that is defined by the LUT (e.g. MPBs and the legacy shared memory), the according entry must be traversed during the address translation process (cf. Section 2.2.2). However, to explicitly address an LUT entry, the physical address of a memory access must be set accordingly. Modern operating systems and processors, like Linux on x86 processors, prevent such accesses with the concept of the virtual address space and memory protection schemes.

To avoid this problem, the SCC-adopted Linux kernel adds additional device drivers. Those are similar to the `mem` driver of the stock Linux kernel which enables an application with root privileges to map a specific region of the physical address space into the application's virtual address space. For establishing such a mapping, the `/dev/mem` character device is used along with the `mmap` system call.

In contrast to that device, the SCC-specific `rckmem` driver enables any non-root application to perform such a mapping. In addition, the driver allows to specify the memory type which should be setup in the according page table entries. By default, the sccLinux kernel supports the NCM, DCM and MPBT-WB memory types with similar named character devices, like `dev/rckncm`. To perform memory accesses to a special component of the SCC, like an MPB, with the appropriate memory type, an application has to `open` the character device file and must use the returned file descriptor in a subsequent call to `mmap`. As a parameter to the latter function, the desired physical address range is specified. The resulting pointer inside the process' address space can then be used for regular memory operations to access data in the desired resource.

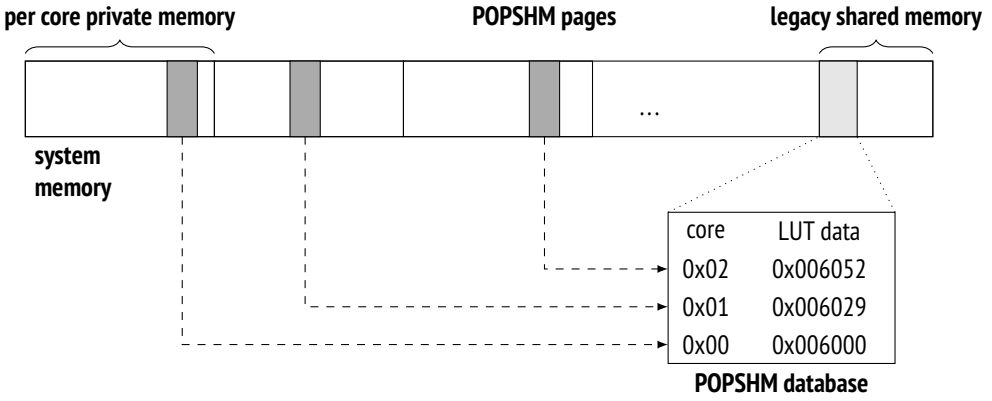


Figure 2.6: *POPSHM pages and database inside the legacy shared memory.*

Privately-Owned Public Shared Memory

Besides the support by special device drivers, the adjusted kernel allows to export a portion of the private memory to other cores. As a result, this memory can be shared between multiple cores. Therefore, this portion of exposed per-core memory is called *privately-owned public shared memory (POPSHM)*.

At boot time, the kernel reserves a fixed amount of 16 MB large pages. The amount of pages is given by a kernel command line parameter. Without further reasons, a hard upper limit of four 16 MB pages is implemented. Although reserved, the kernel does not use the pages in future and does not handle it over to applications. Hence, they can be used for other purposes.

When the reservation is established, the kernel stores the number of registered pages along with their according LUT entry data in a well-known location in the legacy shared memory. The location is called *POPSHM database*. This is illustrated in Figure 2.6. With the help of SCC-specific kernel device drivers (see previous section), an user-space application can access this database and read the LUT configuration data for the exposed POPSHM pages of each core. Since the LUTs are configurable at runtime, the information from the database can be used to enable access to the pages exposed by other cores. If all cores expose the maximum of four pages, then this feature allows to create 3 GB of shared memory, noting that the 160 unused LUT entries (see Table 2.1) are not sufficient to address all of that memory. However, the usage of this memory is safe since pages do not contain sensitive kernel or user space data, like stack or code data.

2.3 Programming Models for Many-core Processors

Several programming models and according implementations are available to be used on parallel computers, depending on the their hardware architecture. With a focus on shared-memory-based non-cache-coherent processors, employing programming models for shared memory appears to be valid. In those models, however, the whole memory is shared by default and all units of execution, e.g. threads, may access any part of the address space at any time without further actions. Implementing such a model on a non-cache-coherent system causes high performance penalties due to the management of cache coherence that needs to take care of every memory access since it might access actually shared data [61]. For that reason, shared memory programming models like OpenMP [62, 63] or threading libraries such as POSIX threads [64], are build around the assumption of hardware-based cache coherence [62]. Hence, those models are not considered any further within this thesis.

2.3.1 Message Passing Concepts

Message passing is a fundamental parallel programming paradigm that is used when processes have to exchange data. It does not rely on shared memory but rather assumes some kind of interconnect between the processes that run in parallel. Nevertheless, shared memory can be used to transport the messages.

The concept of message passing for parallel programs dates back at least to HOARE's paper on „*Communicating Sequential Processes*“ (CSP) from 1978 [23]. Here, a sequentially executed process with a given name can send data to another uniquely named process via input and output commands. Those have to match each other. A process is delayed until the matching operation in the other process is executed.

In general, message passing programming models have in common that there exist multiple separate processes. Those can execute different programs as it is proposed by HOARE [23]. In contrast, it is also common to follow the concept of *single programm, multiple data (SPMD)*. The processes execute the same program but operate an different data sets, i.e. they do not share any information.

When it comes to communication, two types of processes can be identified. A *sender* emits data or *messages*, which are transferred to the *receiver*. To address specific processes, they need to bear an *unique identifier*, like a name. In general, a *matching*

between the sent and received messages by certain criteria is demanded. Those criteria may include the identifier of the sending process, the structure of the message, or a *tag* that labels messages of a certain type. Wildcards for those criteria may also be used.

The communication itself can be performed in different ways. It can block until both processes have issued their communication calls, like it is the case in CSP [23]. A less restrictive way is to ensure that the provided data buffers can be reused after the communication operation returned. In case of a receive, this also means that a matching send operation has completed and application buffer was updated with received data. In both approaches, the communication is often denoted as *blocking* or *synchronous* since a synchronization between the processes is achieved when the message was received.

Opposite to this, *non-blocking* communication, does not enable the reuse of the buffers or requires matching operations on the sender side, respectively. This has two effects. First, an application can continue with computations while the communication system processes the issued operations. Therefore, an overlap of communication and computation is possible, which can hide the latencies of communication operations. Second, to check for the completion of message transfers, e.g. buffer updates on the receiver side, separate means for synchronization are necessary.

2.3.2 The Message Passing Interface Standard

The *Message Passing Interface (MPI)* is the most prominent example of an interface that builds upon the message passing programming model. This standard defines a set of routines to enable platform-independent *application programming interface (API)* for implementing parallel programs. The first version of the standard was released in November 1993 [65, p. 2]. Subsequently, eight versions have been released, with version 3.1 [65] from 2015 being the most current one at the time of writing.

The MPI specification is the de-facto standard for developing parallel applications in the field of HPC. Implementations like MPICH, Open MPI, and MVAPICH are prominent implementations of the standard's API.

In MPI, every process is identified by a numerical *rank*. The processes are not required to execute the same program [65, p. 20], but it is common for MPI application to follow the SPMD concept (see above).

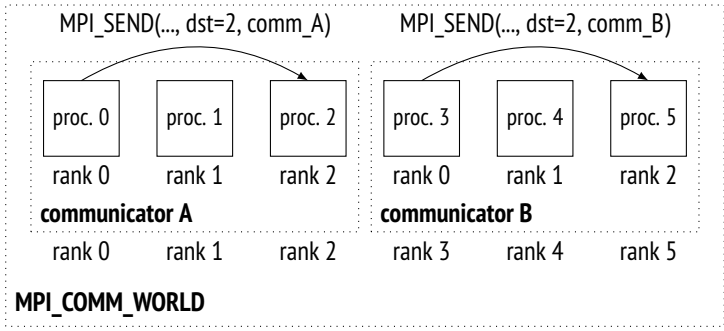


Figure 2.7: *MPI processes, communicators, and communication within*

Processes can be arranged in groups. In a group of n processes, the ranks from 0 to $n-1$ are present. To distinguish groups of the same processes, *communicators* define a communication context. Messages are always sent within a communicator [65, p. 27]. Together with the numeric rank, communicators are used to uniquely identify a process as shown in Figure 2.7. The `MPI_COMM_WORLD` communicator contains all processes. New communicators and groups can be derived from the world communicator and its group, respectively.

Within the standard, different sets of routines are defined. Point-to-point routines [65, § 3], like `MPI_SEND` or `MPI_RECV`, enable the exchange of messages similar to the CSP scheme by HOARE.

The exchange can be performed in a blocking or non-blocking fashion. In the notion of the standard, non-blocking operations, like `MPI_ISEND` and `MPI_IRECV` do not have any temporal properties, e.g. they do not have to finish work in a specific time frame. Instead, non-blocking communication functions do not allow the usage of the supplied buffers until the operation has completed. To check and wait for completion, calls like `MPI_WAIT` are available. For blocking communication calls, like `MPI_SEND` or `MPI_RECV`, it is ensured that the buffer can be reused when the function returns. In case of `MPI_RECV` this implies that a matching send operation has been issued, the message was received, and the supplied application buffer was updated with the message's content. Thus, synchronization between the sender and receiver implicitly performed.

All communication point-to-point operations have in common that they require the specification of the sender and the receiver. That is, the user needs to specify the communicator and the rank on both sides of the communication (see Figure 2.7). For that reason, this communication scheme is also known as *two-sided communication*.

In addition to the sender and receiver, the message itself needs to be identified. Thus, a *tag* is required upon message sending and reception. For receiving data, wildcards can be used for the tag and the sender. The standard formalizes rules that describe when a send and receive operation match each other based on the communicator, rank and tag. Further, ordering rules are specified which an implementation must follow to comply to the standard [65, § 3.5].

With increasing number of cores and, thus, more communication per processor, maintaining these requirements becomes a performance criticial task within the implementation of the MPI [66, 67]. Despite those issues, point-to-point messages are probably the best known parts of the MPI standard. Together with collective operations, which are operations executed by all processes of a communicator, they are integral building blocks of many parallel applications.

2.3.3 Message Passing on the Single-Chip Cloud Computer

Since the SCC was designed with support for message passing (see Section 2.2), according libraries exist which support that programming model on that chip.

RCCE

RCCE[6] is a lightweight library for implementing message-based parallel programs on the SCC following the SPMD model [68]. It is based on operations, that store data in the MPB and load data from there without participation from other processes. It assumes that all MPBs are shared. Data therein must be allocated using collective operations. This creates a symmetrical namespace. The library creates two regions within that namespace, one for internally used flags, and one for data to be exchanged. The flags are set by send operations and polled by blocking receive operations.

In addition to the basic functionality for message transfers, communicators are provided in the basic library. RCCEComm [69] and iRCCE [70] extend the library by collective operations and non-blocking operations, respectively.

While the extended functionality of RCCE enables programming of parallel application on the SCC, it has to be considered, like the chip itself, as a research vehicle. It

[6] pronounced as *"rocky"*

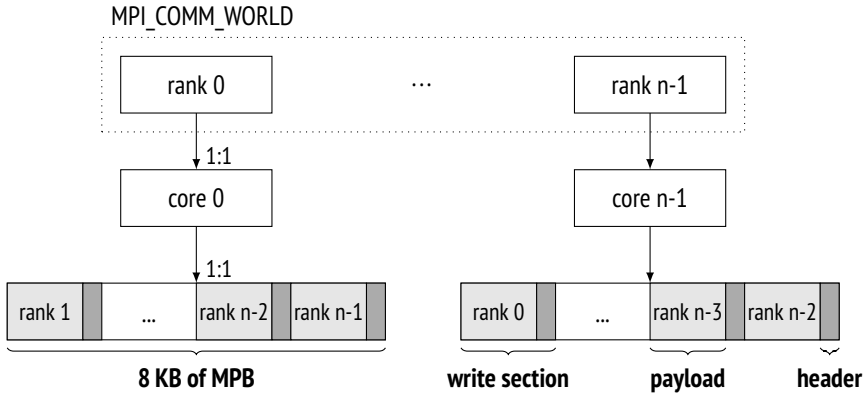


Figure 2.8: *RCKMPI's original layout for the SCC's Message Passing Buffers.*

provides a lightweight and easily extendable message passing programming environment for the processor. To analyze the behavior of real-world application, porting those to RCCE is required. Using MPI, for which a larger code base of applications exists, is therefore an alternative for investigations.

RCKMPI

With RCKMPI [71] an MPI library for the SCC exists. It is based on MPICH and, different to other attempts [68], does not rely on a TCP/IP network provided by the Linux operating system. Instead, the MPBs are directly used for transferring data between the cores.

The library allows only one MPI process to be executed on a single SCC core. Each per-tile MPB is split into two 8 KB parts that are assigned to each of the two cores. Thus, a per-core MPB is created. When n MPI processes are executed, this buffer is logically divided into $n-1$ *exclusive write sections*, dedicated to every MPI process except the local one. This is illustrated in Figure 2.8

Within every write section, metadata and payload is stored (see Figure 2.8). A sender writes the message into the payload area of the write section in the MPB of the remote receiver. Afterwards a sequence number in the metadata is increased. The receiver polls the send sequence number for changes. In case of a change, the data from the payload area is extracted and the reception is acknowledged by increasing the receive sequence number.

The sequence number based protocol is implemented in the lowest layer of MPICH library. A so-called channel device implements both the transfer of outstanding messages and the polling of the MPB to detect incoming messages to satisfy receive operations.

Besides the MPB, the MPBT-WB memory type and the `CL1INVMB` instruction are used by RCKMPI. The MPBs are mapped with the memory type in order to remove stale data from the cache using the new machine instruction. This is required to observe the sequence number changes in the metadata and to fetch the most recent data from the payload section.

In addition to the MPB-based channel device, RCKMPI also offers two other channels that use POPSHM with uncached memory accesses, and a combination of the MPB and POPSHM channels. A performance analysis by CHRISTGAU AND SCHNOR [2] showed that the pure MPB-based channel outperform the other ones in terms of bandwidth. As a consequence, they are not considered within this thesis.

2.3.4 One-sided Communication

As described above, the concept of message passing in general requires both the receiver and the sender to know when a message has to be received. Within an MPI application, this implies to know the rank and the communicator of the sender, and the tag of the expected message. This model applies well to static communication patterns. However, when it comes to dynamic patterns, expressing the data transfers with a two-sided approach becomes a complicated task even with the opportunity of using wildcards. [72, pp. 133 ff.]

Besides those issues of expressing dynamic or irregular exchange patterns, two-sided communication implies synchronization between the processes. That is, a receiver cannot continue until the according data has been emitted by the sender. Similar the sender might not continue until the receiver becomes ready to receive the data, i.e. it allowed the transfer data into its local memory. This cooperation between the processes can have an impact on performance. In MPI, frequent and implicit synchronization by communication operations might be solved with non-blocking communication [72, p. 120 f.], but it still has the problem for expressing dynamic communication patterns.

Concepts

The idea of *one-sided communication (OSC)* provides a solution for both issues. Within the concept, only one process is required to specify the parameters of the communication, e.g. the amount and destination of transferred data. In addition, OSC breaks the bonding between data transfer and synchronisation that is implicitly present in two-sided communication. [72, p. 133]

To enable one-sided data transfers, access to remote memory is enabled between processes. In contrast to shared memory programming schemes, only a portion of a process' address space is shared. Generally, the shared portions need to be explicitly specified. In actual implementation of the model, this often implies that memory needs to be registered to become shared [73, 74], either by explicit library calls [65, pp. 405 ff.] or by special variable declaration [75, 76].

The described selective sharing of data does not imply that the underlying memory is physically shared. Instead, the concept can be applied to both system with shared or distributed memory [74].

This applies well to the cluster-like default configuration of the SCC where the cores (and the processes running on them) have distinct address spaces in terms of system addresses, but where shared memory can be created by means of the LUTs. Thus, supporting the programming model of OSC is a valid choice on the SCC's architecture.

The communication itself is enabled by primitives that either fetch or replace remote memory. Additional operations might be defined to support more complex or atomic tasks [65, pp. 423 ff.]. The communication can also be hidden from the user by language elements: The usage of variables that denote shared data might be translated by a compiler into invocations of low-level communication operations [75, 76].

To ensure that communication occurs in an ordered manner, e.g. that the remote memory contains the correct data or to notify a remote process about the end of communication, synchronization is still necessary. However, as depicted above, the synchronization is separated from the communication. So, explicit operations to synchronize processes are required, but those can complete a bulk of communication operations that have been issued beforehand. This is also known as bulk synchronization [72, p. 133].

The explicit synchronization also enables implementations to ensure a coherent view on the shared memory. The according synchronization operations can be used to perform the coherence operations. Those might be assisted by hardware mechanisms, but they are not required to be so. Hence, the coherence management can be implemented in software as well.

Implementations

The OSC model can be used either directly or indirectly. Libraries like BSPlib [74], MPI [65, §11], or OpenSHMEM [77] provide direct access to memory registration and communication primitives. Those can either be used to write applications using OSC or provide libraries that also provide one-sided semantics but on a higher level. Language-based approaches like Chapel [78], Co-array Fortran [75], or Unified Parallel C (UPC) [76] aim to provide more productivity to the language user [78] by hiding the low-level mechanisms.

These languages provide a *partitioned global address space (PGAS)* that creates the illusion of shared memory although the data is actually distributed across different address spaces. However, in languages like UPC, constructs exists that expose low-level functionality to the user [79, p. 48]. In fact, PGAS languages are built on top of communication libraries. Some use specialized libraries especially designed for this type of language, like GASNet for UPC [80, 81]. However, MPI can be used as a foundation for their implementation as well [82, 83], although there have been limitation to do so in the past [84].

As apparent from above, several libraries could be considered for a discussion of the OSC programming model on the SCC. Concerning performance, different studies present different results: BAUER ET AL. [85] declare MPI as a winner over UPC and Co-Array Fortran, BURKHART ET AL. [86] present *"roughly the same [...] performance numbers"* for Chapel and MPI. Results from OHMANN reveal the same for MPI and UPC, while MALLÓN ET AL. [88] observe slower performance for UPC than for MPI due to a bad language implementation of UPC, and COARFA ET AL. [89] present better speedup values for MPI than for UPC. The reverse result is presented by BELL ET AL. [90] and MAYNARD [91] who clearly favor UPC over MPI based on the observed performance.

While those mixed observations do not allow a clear assessment of which implementation of OSC is faster than the other, it shows that MPI can compete and even out-

perform other libraries. Application developers also observe significant performance benefits when using MPI's one-sided communication with a well-tuned implementation [92]. Moreover, MPI is the dominating programming environment in the HPC domain. A survey of 28 HPC centers in Europe revealed that out of 57 scientific application none employs *"any of the PGAS family of libraries/languages (e.g. CAF, UPC, SHMEM)"* [93, p. 19]. A marginally minority or even none of the surveyed programmers use these languages, while MPI OSC is the fourth most used programming interface [93, pp. 41 f.]. Consequently, this thesis focuses on the MPI version of one-sided communication.

2.3.5 One-Sided Communication in the MPI standard

Within Chapter 11, the MPI standard defines interfaces for using one-sided communication [65]. Following the general concept, separate descriptions for communication and synchronization functions are provided to complement the two-sided communication API of the standard. In addition, the memory model, semantics, and correctness of the OSC interface are presented.

The details of these aspects are explained at the appropriate chapters within this thesis. At this point, an overview of the programming environment is given. Additional information on using the MPI's OSC interface is provided by GROPP ET AL. [72, 94] and HOEFLER ET AL. [73].

In MPI, processes enable access to their memory by opening a *window* to their address space. The creation of such a window is a collective operation within an MPI communicator, Functions that create a window are, for example, `MPI_WIN_CREATE` and `MPI_WIN_ALLOCATE`. By those functions, a process-specific amount of memory is exposed to other process as shown in Figure 2.9. In case of the latter function, the memory is allocated by the MPI library. The other function accepts a user-provided pointer to memory that should be exposed. This memory can be allocated by using `MPI_ALLOC_MEM` as shown in Listing 2.1.

The window creation functions return a *window object* on every process. It serves as a handle to the collectively created windows, which are the locally exposed memory regions [72, p. 140]. *Dynamic windows* allow to attach and detach memory to an existing window at runtime [65, pp. 410 ff.]. The standard also defines creation for shared memory windows on coherent platforms [65, pp. 407], but their discussion as well as dynamic windows is out of the thesis' scope.

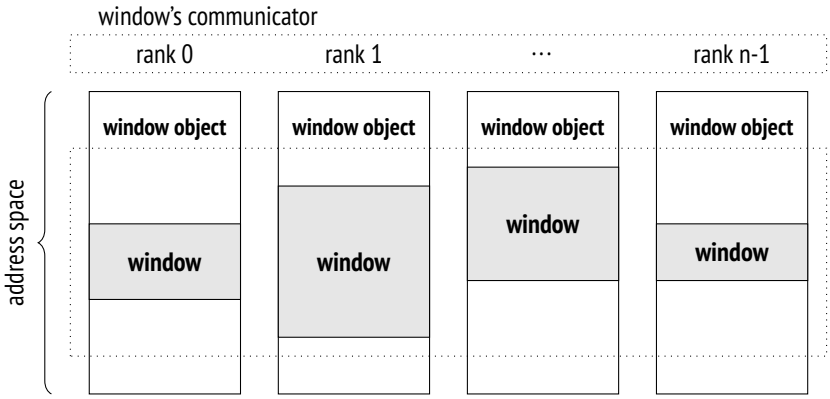


Figure 2.9: *Asymmetric allocation of windows and the window object.*

After successful creation, the windows object can be used in conjunction with a rank to perform a *remote memory access (RMA)*. The addressed process is denoted as *target*, while the process that issues the RMA operation is called *origin*. According to the programming model, only the origin has to issue a function call, like `MPI_PUT` in the example from Listing 2.1, to perform communication.

The communication operations require a buffer containing the data which is going to be communicated. According to the general style of MPI functions, the datatype of the buffer, the amount of according elements in the buffer, and the rank of the target need to be specified. Following the concept of OSC, the offset and the datatype in the remote window are specified by the origin process as well.

Primitive communication operations are fetching data from a remote window, and replacing remote data with local one. Those operations are generally referred as `GET` and `PUT`. Additional operations may allow the combination of local and remote data (`ACCUMULATE`) or perform atomic fetch-and-add or compare-and-swap tasks.

In order to communicate, the origin and target processes need to synchronize each other. Compared to two-sided communication, this is a separated task and it has to be explicitly performed. The MPI standard defines different ways to perform the synchronization at different levels of granularity. In Listing 2.1, the fence synchronization style is used.

```
1  int* buffer;
2  int buffer_size, window_size;
3  MPI_Win win_obj;
4
5  MPI_Alloc_mem(buffer_size, MPI_INFO_NULL, &buffer);
6  MPI_Win_create(buffer, window_size, ..., MPI_COMM_WORLD, &win_obj);
7
8  /* synchronize */
9  MPI_Win_fence(..., win_obj);
10
11 /* distribute local buffer */
12 for (i = 0 ... n-1) {
13     int target = choose_target(i);
14     MPI_Put(&buffer[i], 1, MPI_INT, target, offset, 1, MPI_INT, win_obj);
15 }
16
17 /* (bulk) synchronize */
18 MPI_Win_fence(..., win_obj);
```

Listing 2.1: *Example for window creation and performing RMA.*

2.3.6 Discussion

As depicted above, the one-sided communication model offers a way to implement parallel programs without the restrictions of message-passing. It provides the possibility to selectively share memory and, thus, allows modifications of the shared data without active participation of remote processes. In addition, the synchronization is decoupled from the communication which also enables the management of coherence in software, if it is not available in hardware.

Those concepts match to architectures like the SCC where the sharing of memory can be configured, but where the coherence has to be managed by software. This removes the burden from the hardware for which the transparent maintenance of coherence has been shown to be challenging in high-bandwidth systems. Consequently, the remainder of this thesis focuses on the design of efficient OSC on the non-cache-coherent architecture of the SCC. The discussion is restricted to the OSC interface of MPI, but the concepts might be applied to other — less frequently used — programming environments as well.

2.4 Related Work

As outlined in Section 2.2.2 the SCC can be considered as a distributed system because the cores operate like an on-chip cluster. With regard to the coherence management, the literature concerning distributed systems is vital. The topic is closely related to maintaining memory consistency in distributed systems, where nodes are loosely coupled but a shared memory programming environment should be provided. This section focuses SCC-specific solutions but also mentions more general ones briefly.

2.4.1 Coherence and Consistency in Distributed Systems

Previous work on distributed systems repeatedly discussed the theory and practice how to support shared memory programming for those platforms, also known as *distributed shared memory*. This comes along with questions on maintaining consistency (cf. Section 2.1.1).

The literature provides numerous work on the topic of distributed shared memory. The bibliography of ESKICIOGLU from 1996 lists over four hundred related publications [95]. A full discussion of those is hardly feasible within the scope of this thesis. Instead the main focus is on the release consistency (see p. 11) model, its application to the SCC and alternative approaches for software-based cache coherence on that particular chip.

2.4.2 Coherence via Release Consistency

There exists some research that focuses on release consistency for the SCC. ZHOU ET AL. [96] sketch a software-based approach for managing cache coherence on the SCC and on a conventional multi-core SMP machine that is based on the release consistency model. The foundation of their approach is a domain management module and a coherence policy module. Via the policy module a cache policy can be configured for a certain domain of the employed platform. By doing so, multiple application that run on different cores can use different cache policies.

Conforming to the RC model (see Section 2.1.1), a process needs to logically acquire a shared memory location before modifications, since its view to the most recent

changes of the shared data needs to be updated. After modifications have been performed, a release operation is performed which makes the changes visible to the other processes.

Additionally, the approach of ZHOU ET AL. is based on an extension to the `C++` programming language and its runtime. It adds a new keyword to the language that declares variables as shared or requires invocation of special allocation methods. Due to this language restriction, the MPI interface cannot be addressed.

The Software-Managed Coherence (SMC) library provides the implementation of the previous approach for the SCC [97]. It is based on a Linux kernel module that allows to change the cache settings for a given virtual memory range via the `mprotect` system call. The module is designed for the Linux kernel version 2.6 and required manual manipulation of the system call dispatch table. In the user space, a library implements the domain and policy controller.

The library uses the legacy shared memory (see Section 2.2.6) for the virtual shared address space. It is mapped into every SMC program and thus part of the process' virtual address space. Depending on the cache policy, the acquire and release functions perform different actions.

In the most simplest case, i.e. when caching for the virtual shared address space is disabled, the two functions do nothing. However, the default cache strategy employs a write-through configuration of the L1 cache in pair with the MPBT memory type, i.e. MPBT-WT (see Section 2.2.4). In this case, the acquire function invalidates the cache using the `CL1INVMB` instruction. The release function flushes the write combine buffer that is activated when MPBT memory is used.

All other possible hardware cache policies are supported as well. The L2 cache must be disabled to get those to work correctly because the L2 is not explictetly flushable. In addition, the write-back configuration of the L1 cache should not be used since whole cache lines are written back. If two processes update different data but on the same cache line one update gets lost when the other propagates to the main memory.

The work of [97] and ZHOU ET AL. [96] demonstrates that a software-managed cache coherence is feasible. However, the concept of a shared virtual address space that holds symmetrically allocated data is not applicable to MPI's one-sided communication. Although windows are created collectively, the size of the window memory can vary between processes as shown in Figure 2.9. Further, collective allocation does not imply that all running MPI processes have to be participants of a window creation, because

creation can happen in sub-communicators of `MPI_COMM_WORLD`. Both circumstances let address spaces become asymmetrically.

Shared Data in an ML Runtime

Using the SMC library, SIVARAMAKRISHNAN ET AL. [98] implement a runtime library for ACML, an asynchronous and concurrent version of the ML functional programming language on the SCC. They use the shared virtual memory for language objects, which can be classified as mutable and immutable. An object of either type can become shared among the ML threads if more than one thread references it. When this happens, the object is moved from a local, i.e. private, heap into a shared, i.e. public, one.

For both object types, dedicated heaps are available. A *cached shared heap* is used for immutable objects. Since those objects do not change, the authors admit they do have not to *"deal with cache coherence issues"* [98] when an object would be modified. The SMC library with the MPBT-WT memory is used to access the cached shared heap. To *"circumvent […] coherence issues"* for mutable objects, uncached memory is used. This does match well to the MPI model were windows hold arbitrary, thus writeable, data and makes to ML approach unsuitable for MPI OSC.

In the experimental results, the authors compare their partitioned heap approach with a single unified heap where all caches are disabled. The combined runtime of eight different ML benchmarks is analyzed. Unsurprisingly, a better scalability is observed for the partitioned heap where caches are enabled for immutable objects. The cause is attributed to the caching of the MPBT-WT memory type. However, only 10% of the benchmarks' memory accesses reference shared data. The remaining fraction touches the local heap only which is always cached.

2.4.3 Shared Virtual Memory

To support shared-memory-style programming in distributed systems, the illusion of shared memory needs to be provided to the application. In 1989 LI ET AL. [99] introduce the *shared virtual memory (SVM)* concept for loosely coupled systems. Therein, a mapping manager is used to provide a view to the shared data. The data is not physically shared but only virtually by the mapping manager.

LI ET AL. discuss different options how to implement the SVM approach. They end up using the page concept of operating systems and processor memory protection features. A page that belongs to shared memory is owned by one of the executing processors in the distributed system. Upon access the mapping manager transfers ownership and updates the content of the page frame with the data from the referenced memory page. Accesses to the shared memory page are detected by handling processor faults, i.e. correctable errors, inside the operating system.

Concerning the SCC, several research works rely on the concept of SVM, where the coherence of the caches has to be ensured in the presence of virtual shared memory.

MetalSVM

The Chair for Operating System of the RWTH Aachen University developed the concept of a hypervisor for the SCC named *MetalSVM* [100]. Its aim is to provide an operating system, Linux in their studies, a transparent coherence management and, thus, a cc-NUMA view on the platform. The hypervisor works on top of the hardware and underneath the targeted operating system which runs as virtualization guest and without further modification.

In the (Metal)SVM approach a memory page is owned by a single core. The owner is tracked in page frame metadata. Only the owner core can access the page and modify the data. This is ensured by the page table entries. Page faults are caught by the hypervisor of which one instance is running per core. The instances communicate via messages and interrupts that can be sent between the cores. During page fault handling, the current owner core of a page (if any) is requested to release the page by a message. The owner flushes outstanding writes and cached content to the memory, changes its local page table entry, and transfers the ownership to the requesting core. This one invalidates the cache to get fresh data. As a result, a strong consistency is implemented which prevents any core to access outdated memory. [101]

For the shared pages, MetalSVM uses the write-through cache configuration in the page tables. In addition, L2 caches are completely disabled as its software-based flush operation is considered as being too costly.

For the evaluation, the authors investigate a two-dimensional Laplace problem which is solved with a five-point stencil application, which is similar to a *cellular automaton (CA)*. Results reveal that the message-based communication via the low-level non-

blocking iRCCE library delivers the highest runtimes. Contrarily, the runtime of MetalSVM-based versions is significantly lower (30 – 50%). Anyhow, the discussion does not include a comparison with enabled L2 caches. Thus the reported absolute runtimes might be higher than they could be when caches were activated.

RockyVisor

SOBANIA ET AL. [60] also present a hypervisor for the SCC that exposes a coherent view onto the SCC. It is named *RockyVisor*. Different to MetalSVM, RockyVisor runs on top of an operating system, i.e. the sccLinux (see Section 2.2.6) which was the outcome of preceeding work [56]. The concept of a single page owner appears again in RockyVisor and so does the requirement to flush the cache when ownership is transferred. The implementation of the software-managed coherence was in progress when the article was published. However, no follow-up work is known.

Rhymes

With *Rhymes*, the authors of [102] present another shared virtual memory system for the SCC. Different to the previous SVM approaches, Rhymes is built around the Barrelfish operating system [103]. Rhymes supports two sharing modes for memory. Both modes use lock/unlock routines with IDs that needs to be wrapped around accesses on shared data.

The first mode is similar to MetalSVM and the SMC library, i.e. the MPBT-WT memory configuration is used in conjunction with cache invalidation and flush of the WCB. The authors point out that the write-through operations and the disabled L2 cache are clear drawbacks of this sharing mode.

The second mode needs to be explicitly activated via API calls. In this mode, read-only copies of shared pages are created from a so-called *golden copy* upon first read access on the shared data. The page tables are then exchanged transparently to point to the fresh but *read-only copy*. When a write is issued, the subsequently invoked fault handler creates a second but writeable *twin copy*. When the accesses to the shared data are completed (indicated by an unlock call), the twin copy is synchronized with the golden copy by comparing the data in the writeable twin copy with read-only copy. When changes are detected they are written into the golden copy.

The second mode delivers significant better performance in the conducted experiments compared to the first MetalSVM-like mode. This is attributed to the higher cache utilization which is prevented in the first mode as it uses the MPBT-WT mode. However, the results for the employed Graph500 benchmark with speedup values of up to 350 for 48 cores are questionable since the data — as stated by the authors — nearly fits completely into the L2 cache with increasing core counts. Thus, it is not clear whether the performance gain is due to the new share mode or a better cache utilization. Furthermore, no comment is made on the baseline version. Thus, the overhead compared to a pure sequential version without SVM is not discussed.

Ryhmes' second mode can be applied to MPI's window concept quite well. Before accesses to a remote process are performed, a Ryhmes lock is acquired on the window object and the second mode is activated. The modifications to the twin copy are applied to the master copy, i.e. the remote window memory, when the access epoch ends. However, for large windows, the overhead of synchronizing the twin copy with the master copy might degrade the performance.

Saches

Finally, another implementation of a software-based shared virtual memory for the SCC is presented by KIM ET AL. [104]. It enables multithreaded applications to be started on the chip which appears as a single multicore CPU to those programs. Shared memory is created via configuration of the LUTs. Shared data needs to be accessed through so-called *saches* which provide a copy of the shared data and consistency operations. Those update the main memory and the sache when required for the data that is accessed through these objects.

As a consequence, every load and store operation to shared data needs to be preceded or followed by sache consistency operations. The API offers optimizations for array accesses which update the local data en block and not element-wise. Anyhow, existing applications need heavy rewriting to add the required consistency operations.

Nevertheless, the authors do not attempt to ensure cache coherence between the cores that access shared data. Instead, the shared data is accessed with uncached memory accesses. The local sache appears to be cached, but this is not clearly stated in the literature. On the other hand, disabling the caches appears to be adverse concerning performance.

Summary

In summary, shared virtual memory requires to keep track of current page owner and data transfers to update the local copy of shared data. Most implementations require support by the operating system or hypervisors (see above). As shown in the literature, SVM can serve as base for providing a shared memory view on the SCC. Though, using a software layer for providing a shared memory view on the system in presence of physically shared memory is questionable. In addition, an SVM implementation like MetalSVM would apply to the whole software stack. For MPI's one-sided communication, the coherence needs only to be managed for the window which resides in shared memory, because the remaining parts of a process' address space are kept private. For that reason, SVM approaches appear to be overhead compared to a dedicated solution that manages the coherence of the shared memory only.

2.4.4 Object-based approaches

In addition to the virtual shared memory that is — to the most extend — transparent for applications, there are object-based solutions to provide coherence on the SCC. In those cases, the usage of an object-oriented programming style is required to ensure cache coherence. The concept ensures consistency of shared data on object granularity but neither for the whole address space nor an address range.

MESH

MESH, a flexible and efficient sharing framework for many cores, is presented by PRESCHER ET AL. [52]. It provides a replication mechanism for PGAS-like programming environments where objects can be stored in a global address space. Multiple memory locations can hold replicas of these objects, e.g. read-only objects, to avoid memory controller congestion (see Section 3.4.5). The framework is written in `C++` and is designed for the use in that language.

To access a remote object, an access object needs to be created. The access object takes care of the cache coherence. Upon (automatic) destruction of the access object, issued writes to the underlying object are propagated to the replica (in RAM). It does so by flushing the according L1 and L2 cache lines with the help of the Linux kernel. The

results show that a flush of single modified cache line takes 580 cycles, not including the transition into kernel space. It has to be noted that the employed SCC system was running at higher frequencies: 800 MHz for the cores and 1.6 GHz for the mesh network which is nearly the doubled frequency of the default setting used within this thesis.

The approach of MESH resembles the release consistency model. The creation of an access object matches to the acquire operation while the destruction of the access object matches to the release. However, MESH is limited to the C++ programming language. A language-agnostic approach would require explicit calls for the release/acquire operations as they are provided by MPI's synchronization calls. Therein, the access object is constituted by the window object.

Despite these similarities, the replica concept does not apply well to MPI windows. Compared to generally small-sized C++ objects, MPI windows might have large sizes. Maintaining replicas can therefore cause high runtimes due to memory transfers. In addition, windows are generally not read-only which is one of the motivations for using replicas. Further, the flush operations during the release-like destruction of the access object causes is likely to create large amount of memory accesses. The performance of cache line flushes for a whole window can be significant.

Moreover, the release operation of multiple origin processes after the synchronization that concludes communication would cause race conditions. The performed flush would cause the whole window content to be written back in main memory. Since every origin process performs this operation, data that has been written into main memory by one process will be overwritten by a subsequent origin that completes its access epoch.

MESH and Cache Coherence

Following the work-in-progress publication, ROTTA ET AL. [100] present a software-level cache coherence for the MESH framework with a focus on parallel graph applications. The authors employ the POPSHM library (cf. Section 2.2.6) to allocate shared data. In addition the DCM, NCM and MPBT-WT memory types are used to map the shared memory. Inside this region, shared objects are created that represent elements of the graph.

Access to the shared object is only possible via access proxies (see access object from

MESH). Those take care of coherence operations. When an access object for read or write accesses is created, cache invalidation is performed. When a write access has ended and the access object is destroyed, the cached content is written back to memory. However, an application must follow a multiple reader/single writer pattern which does not apply to MPI one-sided communication were multiple origins can perform `PUT` operations at the same time.

Conceptually, the approach from [100] still has the drawback of using access objects for shared data which is hardly compatible to MPI.

2.4.5 Software-Based Cache Coherence

The management of cache coherence by software has been discussed in previous research. TARTALJA AND MILUTINOVIC [105] present a selection of papers dealing with the topic. They differentiate between static and dynamic strategies, with the first ones being applied at compile time and the last ones being applied at runtime of the program. The authors also point out, that software-based coherence schemes can reduce the complexity of the hardware, can compete with hardware-based approaches, and are scalable as well [105, p. 1]. Thus, those schemes need to be considered as well.

Already in 1987, CHEONG AND VEIDENBAUM [106] use a FORTRAN compiler to detect `DOALL` loops in numerical applications which can be executed in parallel on a nCC shared memory multiprocessor system. To manage cache coherence, the authors extend the compiler and inject invalidation statements before and after the loop body. Thereby, the latest computational results from before and after the loop can be observed by participating processors.

The authors use a write-through cache policy as it avoids to keep track of which cache line needs to be written back to memory. The approach of CHEONG AND VEIDENBAUM requires a compiler that is able to detect parallelism and inject coherence management code. In case of MPI the compiler needs to be aware of the synchronization methods. However, most, if not all, MPI implementations rely on an existing compilers and provide library functions, but do not provide compiler extensions. Thus, this approach is not followed within this thesis.

In subsequent work CHEONG AND VEIDENBAUM, enhance their compiler approach to avoid *"indiscriminiate invalidation"* of the whole cache [107]. The approach is

still compiler-based and injects invalidations instructions. The extension is based on tracking the state of variables within the scope of a subroutine. If variables are only read but not written (by any processor) no cache invalidation is required.

ASHBY, DIAZ, AND CINTRA advance the idea of a more selective software-based cache invalidation [37]. Their concept relies on release consistency. Cache flushes and invalidations are explicitly inserted in the release and acquire operations, respectively. Locks serve as synchronization devices that provide release and acquire options.

In addition, the authors propose hardware extensions which support the invalidation to affect only the memory location that were modified by other processes and belong to a shared memory region. The hardware extension makes use of signatures based on Bloom filters to keep track of changes by other processes. In combination with the cache line tags a lock identifier is created using the filter. The identifier is associated with the lock data structure. Upon acquire, all cache lines are invalidated whose tag-based identifier is found in the Bloom-filter-based lock signature, i.e. those cache lines are invalidated which belong to the set of changed lines associated with the lock. For the release operation, an explicit write-back operation is proposed. [37]

The evaluation using a simulation of up to 32 UltraSPARC processors shows that a full invalidation scheme *"performs surprisingly well"*. The proposed software-based and hardware-assisted solutions offer only slight performance improvements of about 5% with respect to the runtime of a full invalidation solution. In some exceptions, the runtime could be nearly halved. In addition, the software-based approach performs nearly as the hardware-based MSI cache coherence protocol would do and both are close to an ideal baseline.

The results from [37] — as well as the others — show that software-based coherence can compete with hardware-based approaches. In addition, the release consistency-based approach could be applied to MPI one-sided communication and the hardware-assisted selective invalidation also matches the proposed SCOSCo approach. However, the results were obtained using simulation only and were not applied to MPI's OSC programming model. The SCC, in contrast, allows an experimental evaluation and application to the Message Passing Interface which was not addressed in detail in all of the known publications. This motivates further investigations of the SCOSCo idea.

Other software-based approaches often require special hardware support that assists the coherence management. The dynamic schemes of SMITH [108] and CHERITON ET AL. [109] are examples. Due to the lack of systems that support these techniques, they

are not considered further within this thesis. However, the technique of SMITH [108] that uses invalidation of translation lookaside buffers upon synchronization events might be applied to MPI's one-sided communication as well if a processor would support that feature.

Summary

The results from the related work essentially reveal two aspects.

First, software-managed coherence has been widely discussed in the past. Different implementations, that provide cache coherent environments and can compete with solutions which are implemented purely in hardware, have been discussed in the past. Thus, considering software-based cache coherence in the advent of nCC systems (see Section 2.1.2) is a valid approach.

On the other hand, there is no software solution for cache coherence that specifically addresses the one-sided communication model. While more general schemes apply to a wider range of applications, including OSC, they cannot take advantage from the explicit synchronization mechanisms that is mandatory within this communication model.

2.5 Conclusion

As discussed in this chapter, cache coherence in shared memory systems that is purely implemented in hardware faces practical challenges. While hardware based solutions to the coherence problem are unlikely to vanish, supporting interconnected multi-core processors that provide shared memory but no cache coherence becomes relevant in the foreseeable future. It is therefore critical to consider concepts that are based on software, as it has been done in the past.

In addition, one-sided communication was pointed out as being a programming model that fits to non-cache-coherent systems, including many-cores with shared memory. On the one hand, selective sharing of memory enables the accesses to remote memory on which OSC relies on. On the other hand, the necessary synchronization provides means for maintaining the cache coherence.

For the above reasons, the remainder of the thesis focuses on the efficient support for one-sided communication on the non-cache-coherent Single-Chip Cloud Computer. As with one-sided communication itself, the discussion is split into synchronization and communication. The latter includes the discussion on maintaining cache coherence for the programming model in middleware. In the following two chapters, the efficient design for both facets of OSC on the SCC are presented.

3 Synchronization for MPI One-Sided Communication

In the previous chapter, one-sided communication of the Message Passing Interface was identified as an adequate programming model for non-cache-coherent many-core processors. One of the key aspects for OSC is the separation of communication and synchronization.

This chapter sheds light on the details of the synchronization aspect. The according application programming interface of MPI is illustrated in Section 3.1. A classification of different implementation options is provided in the subsequent section. The existing classification from the literature is extended to account more recently published work and to compensate the disadvantages of the known classes. The resulting classification scheme is used to identify appropriate synchronization mechanisms for the SCC. For this purpose, real-world implementations of the OSC synchronization API are surveyed in Section 3.3 and classified according to the previously defined scheme. Based on the outcome, beneficial concepts for nCC systems are identified and a synchronization scheme for the SCC is developed in the subsequent section, followed by an experimental evaluation of its implementation in Section 3.5.

The discussion of the implementation approach and its evaluation was published in the Workshop Proceedings of the ARCS 2016 conference [3] and was presented in a talk of the 12th PASA workshop on site.

3.1 Background: MPI Process Synchronization

The MPI standard requires that one-sided communication happens only after the processes have agreed on the communication [65, § 11.5] (cf. Section 2.3.5). The motivation for this is twofold. On the one hand, origin processes (those which perform the communication) need to know when they are allowed to communicate

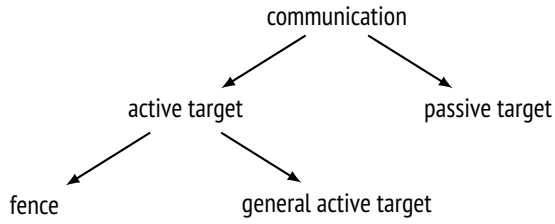


Figure 3.1: *Synchronization styles for MPI one-sided communication.*

with a target process (the destination of the communication operation). For instance, this is required to avoid accidentally overwritten data at the target process. In other words, the targets have to signal their readiness for communication. Thus, the targets participate actively in the synchronization. In this scenario, the second motivation arises. The targets have to be informed about the end of communication, i.e. when no more communication is performed and the window memory that was subject to modifications is available for further usage by the target.

As indicated in Section 2.3.5, the MPI standard defines different synchronization styles that address the described scenarios [65, §11.5]. Depending on whether the target processes are involved in the synchronization, the standard differentiates between *active* and *passive target communication*.

Although the target is never involved in data transfers from the perspective of the MPI API level, this classification takes the required synchronization calls into account. That is, in active target communication a target invokes synchronization methods whereas in the passive class it does not. Figure 3.1 illustrates the hierarchy of the different synchronization styles using the standard's notation. The main principle of OSC, i.e. the communication parameters are only provided by the origin, is still valid within this classification.

Within the active target communication class, further distinction is made depending on how the synchronization is performed. On the contrary, such a differentiation is not made for passive target communication. The next subsections present key aspects of the synchronization API and the differences between the synchronization styles.

Independent of the synchronization style, all of the synchronization API calls require a window object and a so-called assertion as parameter. The window object is essential as the participating processes synchronize on this parameter. Consequently, communication is coordinated on a per-window basis, not necessarily in a global manner.

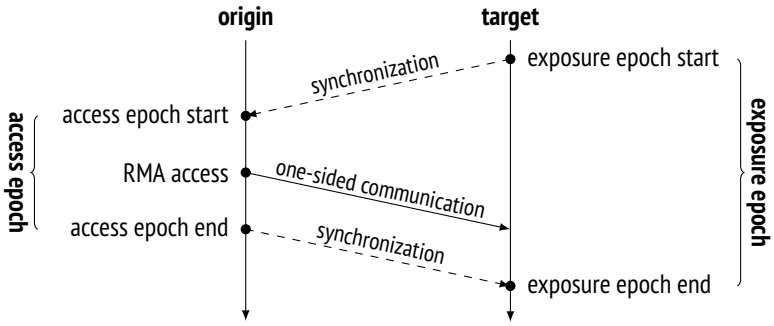


Figure 3.2: *Active target synchronization in MPI OSC programs.*

The assertion parameter may contain hints to the MPI implementation that are provided by an application and describe the application's behavior. Given such a contract between the application and the middleware, the MPI library may take optimized code paths. However, this argument is primary used for optimizations and usually set to zero, meaning no assertion is made.

3.1.1 Synchronization Epochs

All synchronization styles employ calls to start and finish a sequence of code in which the communication can occur. These sequence is called *epoch*. Depending on which type of process (origin or target) is considered, one can further distinguish between *access epochs* and *exposure epochs*.

As illustrated in Figure 3.2, access epochs are found in origin processes. Within an access epoch, communication calls, also referred to as RMA accesses, can be performed. Those calls operate on the exposed memory of target processes. In case of active target communication (see above), the targets participate in the communication by opening an exposure epoch. Within that time frame, they allow to be used as a destination for communication operations. Since passive targets do not participate in the communication at all, the concept of an exposure epoch does not apply. [65, § 11.5]

For active target communication, there must always be a match between access and exposure epochs. That is, an origin cannot communicate to a target if that target does not have an according exposure epoch. However, the MPI standard does not impose restrictions on the temporal order. For example, an origin does not have to

wait for the target exposure epoch at the beginning of its access epoch. In contrast, it is the implementation's choice when the synchronization is actually performed [65, p. 439].

Concerning the communication inside an access epoch, the standard requires that those operations are *non-blocking*. Within the standard's semantic this means the API user is not allowed to access the provided buffer for other purposes. This does not imply the immediate return of the communication operation. Rather, it means that the reuse of the supplied buffers is only allowed after *completion* of these operations. Local completion of RMA operations is ensured by closing an access epoch. Completion on the target side is guaranteed by closing the exposure epoch. [65, pp. 11, 417]

3.1.2 Fence Synchronization

The first synchronization style to be discussed is the *fence synchronization*. It falls into the category of active target synchronization, i.e. both target and origin processes have to call according methods. In particular, the `MPI_WIN_FENCE` has to be called by both sides. The call takes only two parameters: the assertion and the window object, as mentioned above.

The `MPI_WIN_FENCE` routine has to be invoked collectively among all processes that created the window, i.e. by all processes in the communicator which was used during the collective creation of the window object (see Section 2.3.5). As its name suggests, the actual implementation often *"entails a barrier synchronization"* [65, p. 441]. Communication operations (like `PUT` and `GET`) to any process of the window's communicator can be issued between two fence calls. Listing 3.1 illustrates the usage of fence synchronization.

The role of target and origin process is not fixed when the synchronization is carried out. From the perspective of an application, it is not required to specify which role a process will take in the subsequent communication phase. This makes fence synchronization appropriate for global yet changing communication patterns where every process (of the window's communicator) can be chosen as a target for communication and where this decision is made at runtime. [65, p. 437]

In actual implementations, the collective nature of the fence operation is exploited. MPI libraries make use of other collective operations like barriers or reductions. Optimizations of those fundamental operations have been widely discussed in the

```
/* open access/exposure epoch */
MPI_Win_fence(0, win);

if (process_needs_to_communicate()) {
    /* application logic dynamically picks a target process */
    target_rank = choose_target_process(window_communicator);

    /* this process becomes origin, process target_rank becomes target */
    MPI_Put(..., target_rank, ..., win);
}

/* close opened epoch */
MPI_Win_Fence(0, win);
```

Listing 3.1: *Example for usage of MPI's fence synchronization.*

literature, ranging from new algorithms to optimizations for certain architectures [110–114]. This also includes discussions of those operations for the SCC [53, 115–117]. Consequently, the fence synchronization style is not discussed in more depth within this thesis.

3.1.3 General Active Target Synchronization

Different to fence synchronization, the *general active target synchronization (GATS)* allows to synchronize a subset of the processes in the window's communicator. This is advantageous if an application can determine which processes are going to communicate with each other. With GATS it is also possible to specify the roles a process is going to take (origin and/or target).

In detail, the API of GATS consists of four methods that can be divided into two groups, one for target and one for origin processes. Each group is constituted of one routine that opens an epoch (exposure for target, access for origins) and one for closing that epoch, namely:

`MPI_WIN_START` and `MPI_WIN_COMPLETE` open and close an access epoch at the origin process. Similar to fence, the non-blocking communication operations can only be issued between the invocation of these two synchronization methods. The `MPI_WIN_COMPLETE` call ensures that all of the non-blocking communication

```
1 /* origin process(es) */
2 MPI_Win_start(start_group, 0, win);
3
4 /* access epoch */
5 if (need_to_communicate(target_rank)) {
6     MPI_Put(..., target_rank, ... win);
7 }
8
9 MPI_Win_complete(win);
```

```
/* target process(es) */
MPI_Win_post(post_group, 0, win);

/* exposure epoch */

/* no OSC communication operations */


MPI_Win_wait(win);
```

Listing 3.2: *Example of MPI general active target synchronization for origin (left) and target (right) processes.*

calls have finished at the origin such that local buffers are ready for reuse after communication. [65, p. 441 f.]

`MPI_WIN_POST` and `MPI_WIN_WAIT` are used for opening and closing an exposure epoch at the target processes. While a call to `MPI_WIN_POST` can be considered as an indication for allowing communication, `MPI_WIN_WAIT` waits until all `MPI_WIN_COMPLETE` calls have been issued, i.e. all access epochs came to an end. [65, p. 443 f.]

Due to the names of the four methods, the GATS synchronization is also often referred to as PSCW (Post-Start-Complete-Wait) synchronization.

Beside the assertion and the window parameter (see above), the two routines that open epochs, `MPI_WIN_START` and `MPI_WIN_POST`, take a group argument as illustrated in Listing 3.2. This argument is a handle to an `MPI_GROUP` object that represents an ordered set (or list) of processes [65, § 6.2.1]. For `MPI_WIN_START`, the *start group* (G_s) contains the processes to which the origin process can communicate within the following access epoch. Vice versa, the group given to `MPI_WIN_POST`, the *post group* (G_p), holds the ranks of origins that are allowed to communicate with the calling target process. [65, § 11.5.2]

The given groups on both sides have to match each other. In a correct MPI program, all target processes in an origin's start group must have that particular origin process in their post group [65, p. 441 f.]. Listing 3.2 illustrates the invocation of the synchronization methods with the two group parameters. Given a single origin process with rank `0` and two targets with rank 1 and 2, the start group has to contain the ranks 1 and 2. The post group of the two targets must have only one element, rank `0`.

The reason for requiring the described matching is to ensure a correct synchronization between the origin and target processes while the API is still convenient to use. Correctness is required to avoid deadlocks as illustrated in the following example: If not all target processes given from the origin's start group would issue their `MPI_WIN_POST` operation, the origin would wait infinitely until the targets become ready. Therefore, the origin will never perform the notification at the end of its access epoch. Consequently, those targets that actually synchronized correctly would wait in their `MPI_WIN_WAIT` calls for the completion of the origins access epoch, which will never happen.

The convenience aspect arises from the fact that the start group must be considered as a list of possible rather than definitive target processes with which the origin will communicate. As indicated in Listing 3.2, the origin can decide within the access epoch if it actually needs to communicate with a certain target. This is similar to fence synchronization (see Listing 3.1). The main difference is that the set of possible targets is generally limited by the specified start group. However, at the end of a GATS access epoch all targets that started an exposure epoch have to be notified to let these epochs end. To prevent infinite waiting, this must done in any case with all processes from the start group, i.e. even if no communication was performed with a particular target. Otherwise, not all exposure epochs would come to an end. The MPI API facilitates this with the pair of `MPI_WIN_START` and `MPI_WIN_COMPLETE`. By using these calls, there is no need for manually synchronizing with each of the targets.

The laborious construction of the post and start group are not discussed here. This *"somewhat cumbersome"* [118] step does not influence the synchronization process, although it is required to construct the groups beforehand. In general, they have to be derived from an existing group which itself must be queried from a communicator. For details, refer to [65, § 6.3.2].

3.1.4 Passive Target Synchronization

As explained in the introduction of Section 3.1, only the origin processes participate in the communication and synchronization in case of the passive target communication. Targets do nothing from the perspective of the MPI API. However, the targets are still involved in the window creation, i.e. they expose a portion of their local memory for accesses by other processes, including their own. These accesses still need to be synchronize. For that purpose, the MPI standard uses the concepts of locks known from shared memory programming. [65, p. 438]

```
1 if (need_to_communicate(target_rank)) {
2     MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target_rank, 0, win);
3
4     /* access epoch (no exposure epoch on the target) */
5     MPI_Put(..., target_rank, ..., win);
6
7     MPI_Win_unlock(target_rank, win);
8 }
```

Listing 3.3: *Example of passive target synchronization in MPI using an exclusive lock.*

To access a window of a target process, an origin has to acquire a lock first. The lock is specific to a window and a target. The two parameters need to be provided for the synchronization method `MPI_WIN_LOCK`. As a result, the passive target synchronization basically affects only the single targeted process. After acquisition of a lock, the access epoch is opened. When all the intended accesses have been performed, the lock has to be released with `MPI_WIN_UNLOCK` as shown in Listing 3.3. This operation also closes the access epoch.

For the synchronization of the accesses, the MPI standard defines two lock types: shared and exclusive locks [65, p. 445]. The latter type is beneficial to synchronize write accesses with reads or other write accesses to the same location as only one process can hold an exclusive lock. In contrast, the shared type can be acquired by every process as long as no exclusive lock is set at the specified process. This is useful, e.g., for concurrent read accesses which do not interfere.

In addition to `MPI_WIN_LOCK`, the function `MPI_WIN_LOCK_ALL` can be used to acquire a shared lock on all processes that belong to the window [65, p. 446]. This can be useful in a phase of communication where a process performs a lot of communication operations to a high number of processes. To avoid frequently repeated calls to lock/unlock a call to `MPI_WIN_LOCK_ALL` might be beneficial.

It has to be noted that if a process needs to access local memory that belongs to a passively-synchronized window it must participate in the synchronization as well. Otherwise local accesses (by conventional memory accesses, i.e. load and stores) will interfere with RMA operations. Depending on the type of local access, a shared or exclusive lock has to be acquired before any access. [65, p. 447]

The passive target synchronization enables implementations of PGAS runtimes of

programming languages [83, 119], which has been greatly supported by the MPI-3 specification. In addition, PGAS oriented libraries, like GlobalArrays which is used in computational chemistry [82], or distributed data structures, like hash maps trees and linked lists [72, § 6] are supported by MPI's passive communication API. However, further discussion is out of this thesis' scope. A discussion of an efficient implementation for that synchronization style might be subject to future work. GERSTENBERGER ET AL. [120] present an approach that appears to be a good candidate for an implementation on the SCC. The following discussion focuses on GATS.

3.2 Classification of Implementation Methods

As outlined in Section 3.1.1, the MPI standard gives an implementation much freedom to realize the synchronization methods that were discussed in the previous section. For example, `MPI_WIN_START` is not required to wait for all `MPI_WIN_POST` operations (see Section 3.1.3), but an implementation can choose to support this blocking behavior [65, p. 439]. On the path to an implementation of the synchronization primitives for the Intel SCC, it is beneficial to categorize existing implementations. This facilitates the assessment of existing libraries based on their suitability for the nCC many-core chip.

GROPP AND THAKUR [121] provide such a classification scheme for implementation options of the MPI OSC synchronization. The two classes pf implementation methods defined in their scheme are *deferred* and *immediate*.

3.2.1 Deferred Method

For deferred synchronization, the execution of methods that open an access epochs is delayed until the end of an access epoch. The same applies to the communication calls. Their non-blocking characteristic supports the deferred scheme, since an application should not touch the provided buffers until the access epoch has ended. As a result of the deferred synchronization, the actual data transfers are neither performed nor initiated until the access epoch has ended. Only after that the synchronization is executed and the communication can proceed.

A primary downside of this approach is that optimizations, especially the overlap of communication and computation which can hide communication latencies, are not

Table 3.1: *Classification of synchronization for MPI one-sided communication.*

Class	Epoch Start	Communication	Overlap
deferred	non-blocking	delayed to epoch's end	not possible
immediate	blocking	prompt	possible
trigger-only	non-blocking	prompt	possible

possible. On the other hand, the deferral enables an MPI implementation to merge and optimize multiple of the communication calls and consequently reduce the latency for the data transfers. In addition, the deferred synchronization makes an application less prone to process skew. That way, a delayed execution of a target's synchronization method (`MPI_WIN_FENCE` or `MPI_WIN_POST`) does not cause a delay on the origin side when it enters the access epoch (by calling `MPI_WIN_FENCE` or `MPI_WIN_START`). The same applies to passive target communication. Here, an origin process that holds an exclusive lock does not cause another origin's access epoch to start later.

3.2.2 Immediate Method

Opposite to the previous category, the starting synchronization calls (e.g., `MPI_WIN_POST` and `MPI_WIN_START`) of the immediate class perform the synchronization immediately when they are invoked. Usually, this leads to blocking implementations in which the targets wait for the origins to become ready for the communication. This step can be based on barriers or similar collective operations. For platforms with special support for these tasks (like the IBM Blue Gene architecture [122]), the immediate synchronization can be implemented efficiently.

A drawback of this synchronization scheme is its vulnerability to process skew (see above). A late target process causes an origin to be delayed in the start of its access epoch until the target performs it synchronization. On the other hand, this immediate synchronization is also advantageous because origin and target are ready for communication after their epochs have been started. Therefore, communication can be executed on invocation.

This also offers the possibility to overlap communication and computation. In addition, GROPP AND THAKUR point out that this method is beneficial for systems that support true one-sided communication, e.g. shared memory systems, where communication can be performed upon invocation [121].

3.2.3 Trigger-Only Method

In addition to the classification from [121], a third class that combines the advantages from deferred and immediate synchronization (see Table 3.1) can be identified. In the *trigger-only* variant, the starting synchronization calls initiate synchronization operations but do not block to wait for their completion. This is similar to the deferred class. However, in the trigger-only scheme, it is a communication call (`PUT` or `GET`, e.g.) that checks if its target has synchronized, not the ending synchronization call (`MPI_WIN_COMPLETE`, e.g.). If the target is not yet ready, the communication call blocks until the particular target process transitions into a synchronized state.

One of the beneficial aspects of the trigger-only method is that an origin waits for a target process only when it is actually required, i.e. when communication should be performed. This is different to a blocking/immediate synchronization that waits for all processes (given in the start group, e.g.) even if no communication will be performed with some of them inside the access epoch. This makes the trigger-only variant less prone to process skew than an immediate implementation. In addition, after the synchronization with a particular target is completed, all subsequent communication with that process can be performed promptly which enables overlap of communication and computation.

3.2.4 Discussion

Of the discussed synchronization methods, the immediate and trigger-only type are the most suitable ones for an implementation on the Intel SCC. Both ensure that target processes are ready for communication when it is invoked. Further, the SCC provides the possibility to use shared memory based communication (see Section 2.2.2). As pointed out in the discussed literature, the immediate method is well-suited for such a case.

Because the trigger-only method combines the possibility of performing communication upon invocation with a relaxed start of the synchronization epoch, a trigger-only method should be favored for an implementation. In contrast, the deferred method should not be considered as a design approach due to the lack of possible overlap and native support of one-sided communication by using shared memory on the SCC.

3.3 Survey of Synchronization Implementations

For an efficient synchronization scheme on the SCC, the trigger-only and (with restrictions) the immediate methods were identified as eligible classes in the previous section. The following section surveys the implementation of synchronization routines in existing MPI libraries. The aim of this step is to classify these implementations according to the schema presented in the previous section and thereby evaluate their suitability for the SCC. In addition, design concepts that might be useful on the SCC will be identified in this study. From the results, conclusions for the actual implementation of the synchronization protocol will be drawn.

3.3.1 MPICH

MPICH [123, 124] is considered the reference implementation of the MPI standard. One of its goals is easy portability. A layered software architecture that abstracts communication devices supports this aim.

The implementation of the routines defined by the Message Passing Interface is realized on top of the *Abstract Device Interface* (ADI) as illustrated in Figure 3.3. This represents the first layer of platform abstraction. To port MPICH to a new hardware architecture, the ADI layer can be ported to that platform. However, this task is cumbersome, as the ADI consists of many functions that basically map one-by-one to MPI functions that deal with communication.

To ease the portability of MPICH, the *Channel Device version 3* (CH3 device) implements the ADI layer. It breaks the ADI's functionality down into sending and receiving of messages. The simplified platform-specific implementation of send and receive operations is up to so-called channels. The focus of the following discussion lies on the implementation of the synchronization function inside the CH3 device because it is the device that is most commonly used. The analysis is based on MPICH version 3.1.3 and was done by a source code review of the files `ch3u_rma_sync.c` and `ch3u_rma_ops.c` in the `src/mpid/ch3/src` directory of the source code package.

The general scheme of the synchronization methods inside the CH3 device follows the deferred method presented in Section 3.2.1. Since one of the goals of MPICH is easy portability, the CH3 device does not assume the existence of any special hardware features that enable fast and truly one-sided data transfers. It therefore

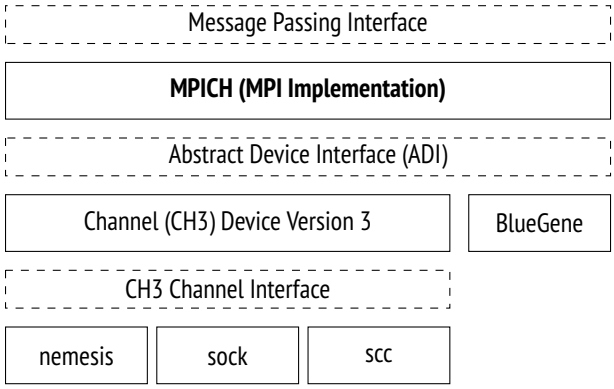


Figure 3.3: *Layered software architecture of MPICH.*

employs an optimized message-based synchronization scheme that aims to minimize the incorporated overhead as described in [125] and [126]. As a consequence of the message-based abstraction that is employed by the CH3 channel, the one-sided communication calls are also implemented with the help of messages.

GATS Synchronization

The GATS (or PSCW) synchronization in MPICH follows a deferred approach that enqueues all communication and processes the queue at the end of an access epoch, i.e. in `MPI_WIN_COMPLETE`. The queued operations are only performed after the target process has signalled its readiness for RMA operations.

To signal the origin the target's readiness, an empty point-to-point message, i.e. a control message handled internally by the CH3 device, is sent during a call to `MPI_WIN_POST`. Due to the employed deferred synchronization method, the `MPI_WIN_START` routine does not wait until that control message has arrived from all processes but does nothing except for saving the start group argument for later usage.

For the termination of the synchronization epoch, an integer variable is used by the targets. It contains the number of origin processes that have finished their RMA operations on the given target process. Therefore, the counter is designated as *completion counter*. Its initial value is set to the number of processes in the post group (see Figure 3.4). The `MPI_WIN_WAIT` progresses the reception of interal MPICH messages until the counter reaches zero, meaning that all origins completed their access epoch.

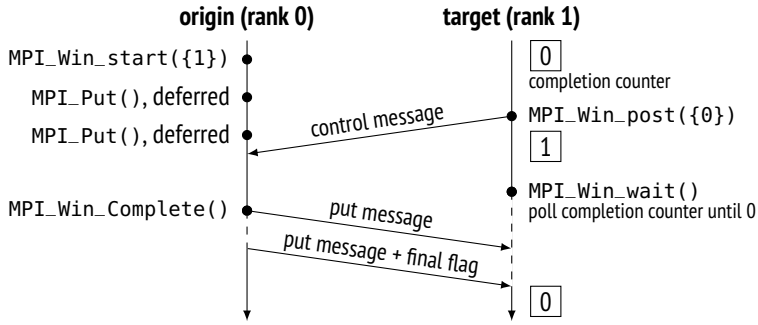


Figure 3.4: *Sequence diagram for GATS synchronization of MPICH.*

To get the notification about the epoch's completion, both the message-based implementation and the deferred approach of the synchronization are exploited. The deferral is achieved by storing the parameters of all communication operations in a queue. The queue is processed in the ending `MPI_WIN_COMPLETE`. Here, the origin first has to wait for all targets to become ready for communication, i.e. it has to check that all of the empty control messages sent during `POST` have arrived. After that, the RMA operations are performed by sending out corresponding internal messages. Those emulate the one-sided operations.

When the final communication message is transmitted, a flag inside the internal message header is set to signal the end of the access epoch to the target process. Upon reception of that final message by the target, the flag is evaluated which results in a decrement of the completion counter. If the counter reaches zero, all access epochs that addressed the local process' window have ended and the `MPI_WIN_WAIT` routine returns. This process is illustrated in Figure 3.4.

Discussion

The approach of MPICH's CH3 device employs a deferred scheme. As discussed in Section 3.2.4, there is no benefit of using such a scheme on the SCC that supports RMA operations via creation of shared memory. The authors of MPICH are aware of the drawbacks but point out that the message-based implementation covers the *"general case"*.[1] Optimizations for *remote direct memory access (RDMA)*-capable networks are in discussion at the time of writing [127]. However, the concept of completion counters

[1] https://wiki.mpich.org/mpich/index.php/RMA_Design last accessed 2016-06-27

is a space-efficient way to implement synchronization at the end of access and exposure epochs.

3.3.2 MVAPICH

MVAPICH [128] is a derivative of MPICH and is primarily developed as high quality MPI implementation for high performance networks like InfiniBand which provide RDMA support in contemporary high performance compute clusters. The implementation has been subject to numerous research activities[2] including the optimization of OSC.

GATS Synchronization for Shared Memory Systems

LAI, SUR, AND PANDA [129] put their focus on OSC for shared memory systems like contemporary multi-core and multi-socket systems. Those provide the opportunity to implement MPI's RMA features in a truly one-sided fashion. With the help of kernel-based technologies and hardware assistance the message transfer is achieved across process boundaries without requiring a shared-memory based window.

Concerning synchronization, only the general active target aspect is discussed by LAI, SUR, AND PANDA [129]. In preceding work, SANTHANARAMAN ET AL. point out that the deferred concept taken by MPICH *"provides no scope [for] overlap"* [110] and thus an immediate approach was chosen. Complementary, the authors of the follow-up publication [110] neglect both a deferred and immediate method in advance of a trigger-only (see Section 3.2.3) variant. Herein, bit-vectors are exploited to perform both the start/post and complete/wait operations.

Each process owns two bit vectors of the window's communicator size, one for use at the beginning of an epoch, one for its end. Each entry is dedicated to the process whose rank in the window communicator matches the bit position. Both vectors reside in shared memory. Upon `MPI_WIN_START` the origin process returns immediately. The publication does not explicitly state when the vector is checked for a target to have issued the `POST` operation. However, Figure 2(b) in [129] leads to the conclusion that the vector is checked when an MPI RMA operation is performed as those *"are not deferred"* [129, § 4.2].

[2] http://mvapich.cse.ohio-state.edu/publications/ lists several hundred publications

The targets set their according bit when MPI_WIN_POST is called. Although it is not clarified how the bits are set, it is very likely to be done with atomic bit operations available in the Intel64 instruction set involved in the experimental evaluation.[3]

The completion of an epoch is realized with the second bit vector. Analogous to the beginning of an epoch, the origins set their bit at the target side to indicate completion of an access epoch within MPI_WIN_COMPLETE. On the target side, when MPI_WIN_WAIT is invoked, the bit vector is polled until all origin processes have set their according bit. It is not motivated why completion counters are not employed.

The presented synchronization method is reused in the succeeding publication of POTLURI ET AL. [130]. The main difference to its precursor is the use of shared memory not only for synchronization data but also for the window's data. If created with MPI_ALLOC_MEM, a windows' memory is backed by shared memory. This reliefs the implementation from the usage of kernel assisted zero-copy data transfer across process boundaries and enables direct memory access by an origin process.

Regarding the evaluation of the synchronization method, both publications ([130] and [129]) do not specifically address the synchronization in the presented experiments. Although both discussions involve comparison with stock MPI implementations, the employed OSU micro benchmarks [131] always include communication. Due to changes in both synchronization and data transfer methods (compared to stock implementation) the presented results do not allow an assessment of the synchronization method and their benefit for an applications performance.

Discussion

The shared memory concept proposed by POTLURI ET AL. [130] is a good candidate for an implementation on the SCC since shared memory can be easily created with the help of LUTs. Further, the concept of bit vectors does not rely on messages and thus promises low overhead.

In addition, the proposed protocol falls into the trigger-only class and is therefore beneficial (see Section 3.2.4). However, the reason for employing a bit vector at the end of the synchronization epoch is not clear and nCC systems are not addressed in the discussed literature.

[3] the LOCK prefix in pair with a bitwise OR or BTS (bitset) instruction are possible for an implementation

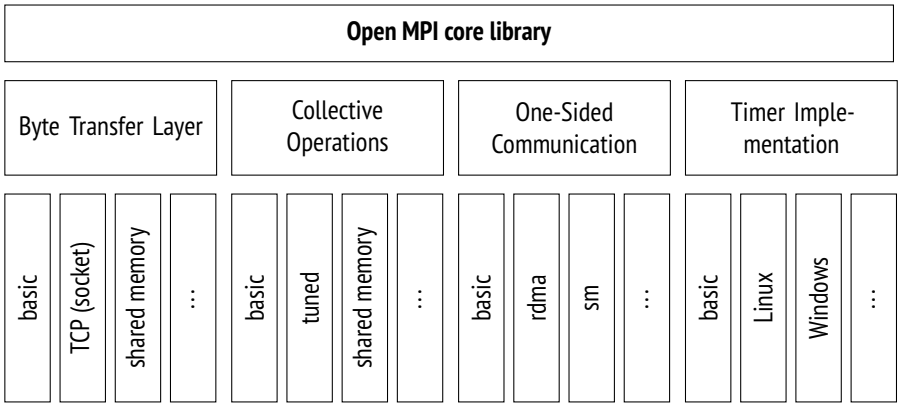


Figure 3.5: *Software architecture of Open MPI (based on [132, Fig. 15.2]).*

3.3.3 Open MPI

Aside of MPICH and its derivate MVAPICH, *Open MPI* is one of the best known MPI implementations. It differs architecturally from MPICH. Instead of a fixed device implementation compiled into the library, Open MPI's functionality is split and implemented in frameworks where plugins, called *components*, provide different realizations of the frameworks functionalites [132]. Among others, prominent examples for frameworks include collective operations, one-sided communication, or low-level routines, like timer implementations. Figure 3.5 illustrates the component-based architecture of Open MPI.

Which component should be used at runtime can be chosen before program execution without the need for recompilation of the library. The OSC framework allows the selection of the *pt2pt*, *sm* or *rdma* component that provide implementations for systems with shared memory and RDMA, respectively. In each of the components, the synchronization methods are implemented differently.[4]

GATS Synchronization

BARRETT ET AL. [133] state that the synchronization mechanisms in Open MPI are *"similar to the design used by MPICH2"*. This is still true for GATS in the pt2pt

[4] https://github.com/open-mpi/ompi-release/tree/v2.x/ompi/mca/osc/, last accessed 2016-06-21

component[5] in the latest stable version at the time of writing (version 2.0.1). The concepts of control messages and completions counters are employed as in MPICH.

During `MPI_WIN_POST`, a control message is sent to the origin. Upon arrival, the origin checks if the post message was expected. If not, the post operation is appended to a list of unexpected posts. Such pending posts are evaluated during `MPI_WIN_START`. If the post message is expected (or upon its reception by the progress engine), a counter is increased. This counter was decremented by the number of processes in the start group within the `MPI_WIN_START` method.

Nonetheless, the counter is not polled in the `MPI_WIN_START` routine for reaching zero which would indicate completion of the post operations at all target processes. Instead, this poll operation is performed in `MPI_WIN_COMPLETE`. As a consequence, communication calls are postponed until the end of an epoch. Therefore, the synchronization method incorporated in Open MPI can be classified as deferred.

The completion of an access epoch is realized by the very same means as in MPICH, i.e. by using completion counters. During `MPI_WIN_POST`, a target's completion counter is set to the negative number of origin processes specified in the method call's group argument. In `MPI_WIN_COMPLETE`, a message is issued signalling the end of the access epoch. A notable difference to MPICH is that there is nothing like a final indicator which is piggy-backed by a communication operation. Instead, a separate control message is required for notification. Nevertheless, upon reception the completion counter at the target side is increased by one. `MPI_WIN_WAIT` and `MPI_WIN_TEST` test the counter for being zero in a blocking and non-blocking fashion respectively.

While the pt2pt component works with the deferred method, the rdma component uses an immediate approach. Instead of messages, it is based on RDMA transactions that modify synchronization data structures. For `MPI_WIN_POST`, instead of a control message, the target directly modifies an array that indicates post operations of targets [134]. This array is polled in `MPI_WIN_START` until all targets have started their access epoch, which is similar to MVAPICH's shared memory approach (see above).

For the synchronization at the epoch end, the concept of completion counters is used again. Similar to the `POST`/`START` operations, `MPI_WIN_COMPLETE` does not rely on messages but increments a per-target completion counter directly via atomic RDMA operations. Accordingly, `MPI_WIN_WAIT` polls the counter until it reaches zero.

[5] file `rdma/pt2pt/osc_pt2pt_active_target.c`

As described in Section 3.2.2, the employed immediate approach is advantageous for systems that natively support RDMA. Thus, the choice of the synchronization method for the rdma component is valid. However, this behavior changed between version 1.8 and 2.0 of Open MPI. Before the newer version, the pt2pt module was named rdma but used the deferred message-based method described above.[6]

GATS Synchronization for Shared Memory Systems

Although the rdma component employs a straight-forward protocol, the sm (shared memory) component of Open MPI's OSC framework used an even simpler one up to version 1.8 (and all sub-versions, starting from version 1.7.5). It is automatically chosen if the window was created using shared memory as backing store of the windows data and all processes in the window's communicator are executed on the same shared memory system.

The simplified protocol extends the usage of completion counters. These are still used for signalling the end of an access epoch to the targets. In addition, the concept of completion counters is also used for the beginning of the access and exposure epochs. Instead of using control messages and a list of received `POST` control messages (see pt2pt component) or a directly modified array that indicates post operations (see rdma component), the sm component uses a per-origin counter for `START`/`POST` operations.

The counter is initialized to zero at window creation. At the beginning of an exposure epoch it is increased atomically by every target process when they perform their post operation. The origin process waits until the counter reaches the number of processes in the start group. When the origin completes its access epoch, the counter is reset to zero. This algorithm is illustrated in Figure 3.6.

Despite its simplicity, this immediate protocol is actually error-prone. A target process from a future epoch that does not belong to the start group can modify the counter during its post operation as if it were a member of the current start group. Different to the control messages in the former approaches, there can be no check if the process belongs to the start group or not. This is due to the usage of counter residing in shared memory. Say, for some reason, process skew occurs and the second target process in Figure 3.6 (right most time line) can issue its post operation earlier than the expected first target.

[6] https://github.com/open-mpi/ompi-release/commit/4fd518e last accessed 2016-06-21

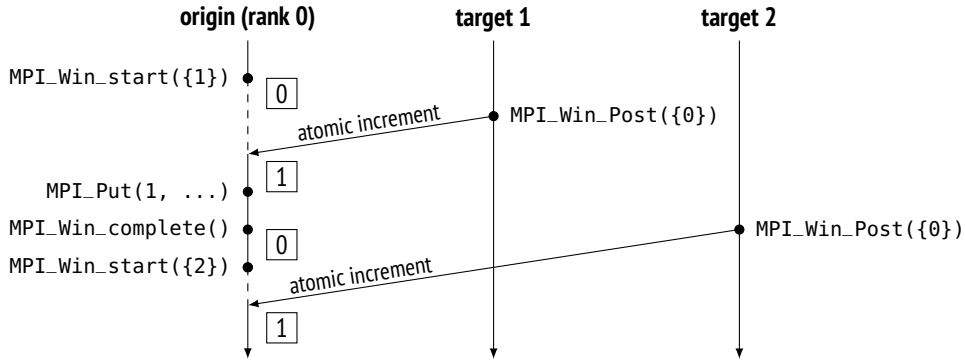


Figure 3.6: *Erroneous shared memory synchronization scheme of Open MPI.*

The origin process subsequently assumes that a member of the current start group posted and continues with communication to a non-synchronized process as soon as the counter reaches zero. This is likely to result in data corruption. Even deadlocks are possible as the completion counter of the unsynchronized target process will be incremented in the described situation. However, the result of this operation will disappear when the target initializes the completion counter in the `MPI_WIN_POST` operation. In the subsequent `MPI_WIN_WAIT`, the target will wait for an increase of its completion counter which will never happen.

This bug was present in Open MPI releases since 2014 until September 2015. It was reported to the maintainers of Open MPI in the progress of this thesis[7] and fixed in version 2.0.[8] Following a proposal of this thesis' author, the new implementation uses the concept of bit vectors from (see previous section) for the start of the synchronization. However, the implementation lies in the immediate category, not the trigger-only. Nevertheless, completion counters are still in use to synchronize the processes at the end of the epochs.

Discussion

The Open MPI implementation of the synchronization methods does not provide new concepts that are usable. The idea to use counters for beginning of the GATS epochs was proven to be error-prone. The concepts known from MPICH and MVAPICH are employed for the GATS synchronization. Only small differences in the

[7] https://www.open-mpi.org/community/lists/users/2015/09/27622.php, last accessed 2016-08-03
[8] https://github.com/open-mpi/ompi/pull/925, last accessed 2016-06-21

implementation can be observed. One significant exception from MPICH is the usage of immediate methods for shared memory and RDMA-capable systems.

3.3.4 FoMPI

GERSTENBERGER ET AL. present an implementation for both MPI active and passive target synchronization for Cray's massively parallel XC system [120, § 2.3]. The MPI implementation for this machine is called *FoMPI*.

GATS Synchronization

FoMPI's implementation of the GATS routines is immediate as well. For the beginning of an epoch a shared list which is called *match list* is used. The list is of fixed size and is associated to an MPI window. The capacity of the list is not stated in the literature. Rather, it is assumed that the number of neighbor processes that will enter the list *"is known to the implementation"* [120, § 2.3]. It is likely that the list size is either equal to the number of processes in the window object or a fixed number that fits for the employed benchmarks.

Within `MPI_WIN_POST`, a target process puts an entry in the list that identifies itself, i.e. its rank inside the communicator associated to the window. The origin process polls the list in `MPI_WIN_START` until all processes from the start group (the parameter associated given to the routine) made an entry in the match list. Thus, `MPI_WIN_START` is blocking and the whole scheme falls into the immediate class.

According to the publication, a target process uses a ring buffer to obtain an entry in the match list. The buffer contains indices of free elements in the match list. A head and a tail pointer are associated to the ring buffer. Upon free space acquisition, a target process reads and increases the head pointer atomically from remote memory which is supported by the XC's hardware. Further, the tail pointer is fetched. If both pointers differ, the ring buffer is not depleted. In that case, the element the head pointer points to contains the index of a free item in the match list in which the target process can make an entry. The free space management protocol is shown in Figure 3.7.

At the end of a GATS epoch, previously occupied items in the match list are freed, i.e. their occupied indices are stored back into the free space ring buffer. However, the

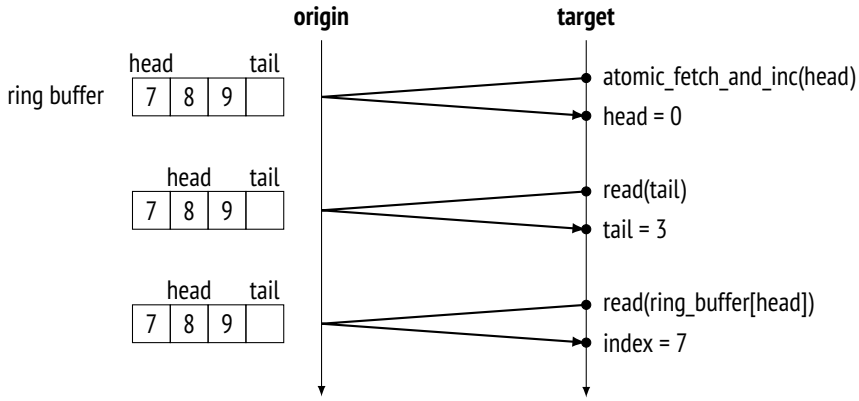


Figure 3.7: *Free space ring buffer protocol of FoMPI (based on [120, Fig. 2c]).*

description in [120] makes no statement about which process removes the items from the list.

To signal the end of an access epoch, the concept of completion counters is used again, as it is in MPICH. Although the counter is manipulated via atomic RDMA operations, the target polls the counter until it reaches zero during `MPI_WIN_WAIT` (see Figure 3.8).

Although the description in [120] is quite precise, it leaves the question how large the match list actually is. The published source code [135] of both the oldest (published before [120]) and the latest (published half a year after the research paper) version reveals discrepancies compared to the description given in [120]. In summary, no free space protocol is present and a different match list design is used.

The actual implementation uses a dynamically allocated array of the window communicator's size. That is, if the application does not create a special communicator before window creation the list of 64-bit integer values is as large as the `MPI_COMM_WORLD` communicator. Further, the list is created at every process and is remotely accessible by all other processes. In contrast to the paper's solution, the implemented approach contradicts the *"highly-scalable"* objective of [120] since for GATS synchronization only a subset of all processes has to synchronize and not all processes. However, if memory is not a matter of concern, a match list of 329 KiB per window might be acceptable.[9] In addition, a direct modification of an element in the match vector

[9] Piz Daint, the XC30 system in question, has 32 GB of RAM per each of its 5272 eight-core CPUs. http://www.cscs.ch/computers/piz_daint_piz_dora/index.html, lass accessed 2016-06-24

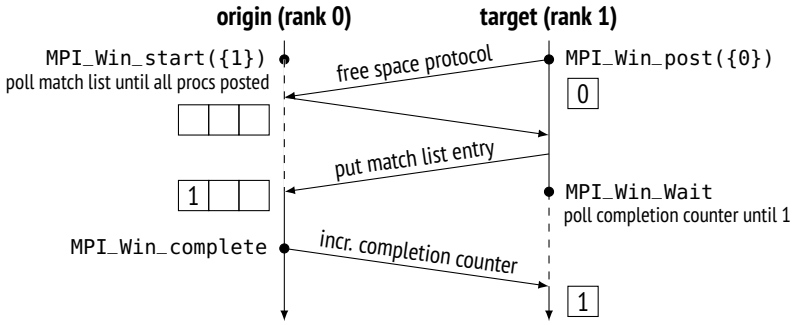


Figure 3.8: *Overview of FoMPI's general active target synchronization as described in the literature (based on [120, Fig. 2d]).*

appears to be faster than involving the free space protocol, which is described in the literature.

Contrarily to the differences in the literature and the actual realization of the start of an GATS epoch, finishing the epoch by means of a completion counter is in fact implemented as described: The origin increments the counter while the targets wait for the counter to reach the number of processes given in `MPI_WIN_POST`.

Discussion

Concerning the evaluation of FoMPI's GATS implementation, both approaches (from the literature and the source code) fall into the immediate method category introduced in [121]. Partially, new concepts are used by FoMPI, but the key concepts are already known from previously considered implementations.

The match vector resembles the bit vectors used by MVAPICH's shared memory synchronization (see Section 3.3.2). However, due to different architectures they are accessed differently, i.e. with XC30 RDMA accesses in FoMPI and conventional load/store operations in MVAPICH, and at a other granularity (bits vs. XC30 machine words)

In contrast, the free space protocol to obtain a free position in the match list is a new concept. However, its purpose and benefits remain unclear in the presence of the differently implemented source code. Furthermore, the protocol appears to be slower than a direct modification of the match list based on ranks (see above).

```
/* origin process */

/* put and notify target */
handle = NEON_Put(..., flag = 0);

/* wait for operation to complete */
NEON_Wait(handle);
```

```
/* target process */

/* expose memory to set of processes */
handle = NEON_Post(buffer, ranks, ...);

/* wait for all rank notifications */
NEON_Wait(handle);
```

Listing 3.4: *Pseudo code for NEON API usage.*

3.3.5 NEON

Although *NEON* [136] is not an MPI implementation, it aims to provide an enhanced synchronization scheme for one-sided communication. Consequently, it is discussed within this thesis as well.

The key idea of NEON with respect to the synchronization of processes is to separate the notification of targets about the end of an access epoch waiting for the local completion of RMA operations. That way, the notification can be send as early as possible, i.e. when the last RMA access (like `PUT`) to a certain target is issued, and not as late as with MPI's `MPI_WIN_COMPLETE` call. By doing so, the synchronization, i.e. notification, can be overlapped with computation. In case of a capable network, the notification message can be transferred along with the RMA operation which also saves latency. [136, p. 85 ff.]

Synchronization Scheme

In essence, a NEON application has to follow the code paths sketched in Listing 3.4. Different to the MPI's GATS routine, there is nothing like an `MPI_WIN_START` or `MPI_WIN_COMPLETE` call in sense of remote process notification in NEON. For the latter purpose, the notification is done in the communication calls with a flag that must be provided by the application and indicates the end of an access. To ensure local completion of communication operations, a `NEON_Wait` has to be called for all issued operations using the returned wait objects. A call similar to `MPI_WIN_START` is omitted in the NEON API by design as its task is said to be hidden in the middleware [136, p. 91].

The implementation of the API supports two transports: streaming sockets (TCP) and InfiniBand via queue pairs. The following considerations are based on the study of NEON's source code and the presentation from [136].

For InfiniBand, synchronization at the beginning of an epoch (`NEON_Post`) is based on messages [136, § 5.4.2] as illustrated in Figure 3.9. A special control message is sent via InfiniBand's send queue by the target. Upon reception by the origin's NEON progress engine, the sending target process is internally marked as having posted its buffer. Thus, it is ready for following RDMA operations. The target's control message also contains an unique identifier for the post operation. This identifier is later used to signal the end of an access epoch.

During communication routines, the according target process and its remote buffer is checked for readiness with the help of the internal flag set by the `POST` operation. If it was not set, the operation is deferred (by means of a queue) until the control message (see above) signals the target's readiness. If that message has already arrived, the communication is initiated immediately using InfiniBand's RDMA capabilities.

A classification according to the scheme from Section 3.2 is difficult since NEON is not an MPI implementation and an equivalent of `MPI_WIN_START` is intentionally not present (see above). The deferral of communication complies to the deferred method. However, as soon as the targets have synchronized, communication is actually performed. This matches, to some extend, the trigger-only class. In contrast, the communication calls do not wait for the target to have synchronized as they would do for trigger-only. Consequently, NEON can not be clearly classified within the existing MPI-focused scheme.

To notify a target about the end of an origin's access epoch, the application has to provide a flag with value `0` to the communication call. This marks the origin's communication as the final one to the given target within its access epoch. When the NEON middleware on the origin side encounters the zero flag, it issues an InfiniBand immediate write along with the actual RDMA write operation (write with immediate). The immediate value contains the numeric identifier that was transfered during `NEON_Post` (see above). When the InfiniBand hardware completes the RMA operation it notifies the origin via its completion queue. The origin then marks the according NEON handle of the RMA operation as finished.

On the target side, the handle identifier received from the last communication call is used to mark the access of the origin process as finished. As soon as all final messages from the origin processes, to which the buffer was announced during `NEON_Post`, have

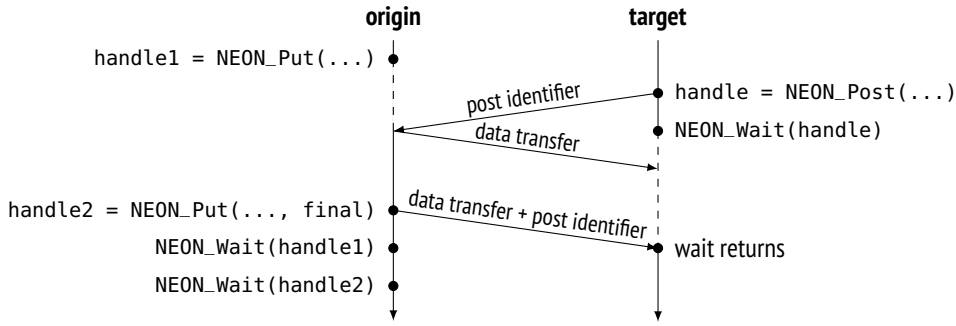


Figure 3.9: *Synchronization scheme of NEON.*

arrived, the corresponding post handle is marked as finished. During the `NEON_Wait` call, the state of the given handle is polled until it reaches the finished state. If required, the progress engine is polled until the wait condition is fulfilled.

It has to be noted, that NEON's InfiniBand implementation is not capable of posting a buffer to multiple processes, although the API states to do so (see Listing 3.4 and [136, p. 107]). Similar, the test/wait routines on target side only wait for one remote process to finish its accesses, not all. Thus, no conclusions for a GATS implementation can be drawn from NEON. Further, the NEON API only supports `PUT` operations, as `GET` is considered to be too expensive in terms of communication and is consequently not implemented. [136, p. 110 ff.] This might be due to the fact that a read operation with immediate write is not available in InfiniBand.

In case of TCP as implementation target, the message-based approach from the InfiniBand implementation is used as well. Although, an additional thread is used to enable communication without involving the application [136, § 5.3.3]. Further, the flag indicating the final communication is stored inside an internal message header [136, § 5.3.2.3] that precedes all internal NEON messages.

Discussion

The NEON design for synchronization provides a concept different than the MPI API. As a result, the implementation also differs from the previously considered MPI libraries.

The most important difference of NEON is the removal of explicit synchronization at an access epoch start. Nevertheless, the design decision has drawbacks that affects an application's implementation.

Target processes in NEON behave very similar to the ones in MPI. They issue `POST` and `WAIT` operations. The important differences are found in the origins. Here, NEON requires every origin that was specified during a target's post operation to actually perform communication with that particular target. It is required as the completion of the access epoch is only signalled by communication methods (see above). This is different to MPI where an explicit call to `MPI_WIN_COMPLETE` on the origin side signals completion to all processes that posted buffers to the origin.

Considering Listing 3.2 on page 58, a process in MPI can dynamically decide if it communicates to a target or not. In such a scenario, a NEON version of the program has to manage the flag parameter that signals the final communication to every individual target. This can be a hard or tedious task. In addition, it implies sending (empty) messages from the application if no communication to a certain target was performed by the application because it is required to notify the target to release it from a wait state. Given these restrictions, NEON is not suited for applications with dynamic communication patterns. An example of this application type are codes that simulate moving particles and perform load balancing when particles cross the border of a process' local compute domain. Depending on the actual trajectory of the particles, communication has to be performed — or not. `pCRASH` [137] is a representative of this class.

Furthermore, the API is not designed to perform the synchronization at an epoch end inside the middleware as there is — by design — no other way of telling the library the definitive end of an access. NEON therefore requires an application (on the origin side) to know which processes *will* issue a `POST` operation. If the communication pattern is not static, determining which processes need to be notified becomes difficult (see above). If a target process is not informed about the completion (because the origin was not aware of its `POST` operation), it will wait infinitely.

Ignoring these drawbacks that are experienced by the application, NEON's latency-saving idea of piggy-backing the final flag , i.e. the notification for a finished access, within communication calls is not applicable for RMA operations on shared memory systems, like the SCC. There is no way of transferring additional information with an RMA access, i.e. memory transfer. Consequently, an additional transfer (or message) needs to be started to send the notification. This adds latency.

Besides these API's drawbacks, the implementation concept is different to all the previously considered MPI libraries. Numeric identifiers for post operations are sent from targets to origins to inform them about the synchronization step. The handles are sent back and processed by the middleware to indicate the completion of an access epoch. In that sense this matches the concept of control messages from MPICH (see Section 3.3.1), but instead of counting the number of arrived post and complete notifications, the handles are used. Although this appears to be different from the MPI implementation, it is essentially a control message but with different internal data. Nevertheless, concepts like completion counters and lists/vectors that record post operations are not employed by NEON.

Finally, NEON's essential concept of early notification to the targets, which was one of the primary motivation for this work, has been discussed in the context of the MPI API. It is considered to be a useful extension to the existing API [138].

3.3.6 Summary

The preceding discussion of the active target synchronization in multiple MPI implementations reveals different but similar approaches in the actual realization of the MPI process synchronization API.

Table 3.2 summarizes the discussed implementations. A significant result is that all implementations for special hardware systems, i.e. shared memory and RDMA-capable systems, effectively adopt the recommendation from [121] and use an immediate or trigger-only method to allow an efficient implementation of one-sided communication (cf. Section 3.2.2).

For these non-deferred implementations, the beginning of an epoch at a target is signalled to origins either by explicit messages or by the usage of shared memory which is modified using atomic operations. Origin processes check for those messages or a specific state in the shared memory to start their access epoch. When the following communication actually happens is again specific to the implementation and the classification of the synchronization as discussed in Section 3.2

The end of a GATS epoch is basically the reverse to the beginning of an epoch: the origin notifies about the epoch completion using messages or shared media and the target polls for the according messages or shared media state. A common mean for this operation are completion counters, with MVAPICH — for unknown reasons — being an exception as it uses bit vectors.

Table 3.2: *Overview of the presented MPI general active target synchronization protocols.*

Implementation	Method	Means employed for Synchronization
MPICH	deferred	counter manipulating control messages for start (explicit) and completion (piggy-backed)
MVAPICH (sm)	trigger-only	bit vectors for start and completion, both residing in shared memory
Open MPI (pt2pt)	deferred	explicit control messages manipulating counters for start and completion
Open MPI (rdma)	immediate	RDMA-based manipulation of post vector and completion counters
Open MPI (sm)	immediate	atomic counters in shared memory for start and completion (erroneous) / bit vectors for start and completion counters
FoMPI (publ.)	immediate	targets acquire and set entry in match list during post; atomic RDMA to completion counters
FoMPI (source)	immediate	fixed size array of ranks that posted; completion counters
NEON	(deferred)	identifier of post operations handed over to origins and back

An exception in the above discussion is the NEON library which passes handle identifiers between target and origin processes. Nevertheless, the API as well as its implementation must be considered experimental since the synchronization of multiple origins is not implemented and therefore not evaluated. Further it implies difficulties for writing applications with dynamic communication patterns, as illustrated in Section 3.3.5.

3.4 Synchronization for the SCC

Based on the survey and its result from the previous section, an optimized implementation for MPI process synchronization is developed for the SCC

In the outcome of the conducted survey it has been identified that for shared memory systems, the usage of an immediate synchronization is beneficial and often applied

in implementations. Since the SCC can be also considered as a system with shared memory support (cf. Section 2.2.2), the usage of an immediate approach is expected to be beneficial. Nevertheless, the trigger-only class would be less prone to process skew as discussed in Section 3.2.3. Consequently, an implementation that uses a trigger-only method is designed in the following.

3.4.1 Analysis of RCKMPI's Implementation

In addition to the previously presented implementations, the existing MPI implementation for the SCC, RCKMPI (see Section 2.3.3), is examined in the following to identify its workings and possible drawbacks concerning the synchronization for one-sided communication.

Essentially, RCKMPI uses an deferred approach. This is due to its heritage from MPICH (see Section 3.3.1). That shortcoming emphasizes the need for a different implementation for the SCC as the deferred synchronization has been identified for being suboptimal for shared memory systems.

Ensuring Functionality

Besides the deferred synchronization, a further drawback inherited from MPICH is the usage of control messages for synchronization. As a result, messages for this task need to be assembled, copied into the remote Message Passing Buffer, and need to be received and processed by the according destination process. For this case, several functional requirements of MPICH's CH3 device layer (see Section 3.3.1) to an CH3 channel need to be considered. However, the original implementation of RCKMPI was not programmed to support one-sided communication. Most of the required handling of the CH3 control messages for OSC was not realized correctly [4].

As a result, programs using the MPI OSC API were hardly working when the required erroneous implementation was employed. In the progress of this thesis, the RCKMPI was fixed to provide a working message-based OSC API. In addition, the SCC-specific CH3 channel implementation was migrated to a more recent version of MPICH. This was necessary as MPICH2 1.2.1p, on which RCKMPI was originally implemented, contained bugs in the one-sided API on its own. Consequently, it was ultimately upgraded to MPICH 3.1.3. The channel device itself was streamlined: The additional

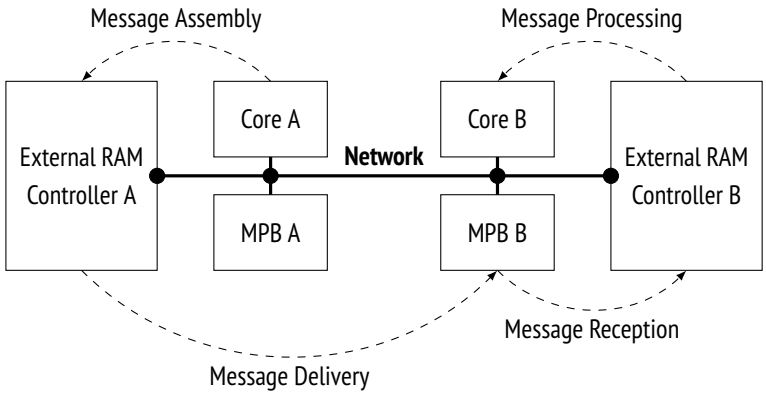


Figure 3.10: *Memory transfers for RCKMPI's synchronization messages.*

CH3 channels that employ high latencies [2] by using uncached memory (see page 35) were removed.

For the remainder of this thesis, RCKMPI refers to the MPICH 3.1.3 derivate for the SCC that uses the purely MPB-based CH3 channel and was fixed to get a working MPI OSC implementation [2]. If the original version which relied on MPICH 1.2.1 is referred to, it is explicitly stated.

Performance Considerations

Although implementation errors have been corrected in RCKMPI, the library still suffers from the problems inherited from MPICH. A complete GATS cycle between a single origin and a single target process costs at least two message transfers (one for `POST`, one for `COMPLETE`) with the overhead described above.

While this drawback can be slightly compensated by the piggyback delivery of the final synchronization message (see Section 3.3.1), the general disadvantage of message-based synchronization still applies. Figure 3.10 illustrates which memory transfers are required to assemble and deliver a message on the sender side and receive and process it on the destination side. Note that the message transfers require data to be sent over the on-chip network which may have large distances.

In case of RCKMPI it means that a message header for the internal control messages needs to be constructed, filled with meaningful data, and copied into a remote MPB.

Then, the content of the receive buffer needs to be copied into the RAM of the receiving process. Finally, it is processed by the internal MPICH progress engine.

In addition to the copy overhead of this scheme, the amount of data that is transferred to signal both a `POST` and a `COMPLETE` operation (see Section 3.1.3) is large compared to the transferred information. Even an empty control message of MPICH CH3 device with a special tag requires 32 Byte of internal header information. Even though this fits into a single cache line of the SCC's core, the amount of data to signal the mentioned operations is higher than required. At minimum it must include an identifier of the operation (`POST` or `COMPLETE`), the window, and the sending process (rank).

3.4.2 Related Work

Other researchers investigated how to optimize synchronization on the SCC and other nCC system as well. Thus an analysis of according publications is vital for this thesis.

In [115] and [55], REBLE ET AL. present the design of a hypervisor that provides a shared memory view onto the SCC hardware. It allows to run a single virtualized Linux instance like on a conventional cache-coherent multi-core processor. The hypervisor uses an ownership concept where every page frame of memory is owned by a single core. The ownership can be transferred to another core if it accesses the frame. However, the transfer needs to be synchronized between all cores.

The authors describe a hardware-based implementation of a barrier synchronization which exploits the test-and-set register of the SCC (see Section 2.2.5). It is depicted that contention on these registers has negative impact on the performance of the barrier implementation. An exponential back-off method is used to lift the contention [115]. However, for small numbers of cores that participate in the synchronization, the contention is not visible. Starting from 32 cores, the back-off method pays off. In addition, this implementation outperforms the RCCE library barrier implementation that needs 60 µs to synchronize all 48 cores where the TSR-based back-off approach performs this operation in about 25 µs.

KOHLER AND RADETZKI [116] tune the collective operations of the RCCE library. Among others, a reduce-scatter operation is improved which is the base for MPICH's, MVAPICH's, and Open MPI fence operation. In general, optimizations like usage of non-blocking send and receive operations, minimization of list-keeping overhead for

managing non-blocking sends, and balancing of the data to be processed are included in the presented library. In addition, the memory copy overhead that is caused by message reception and processing (cf. Figure 3.10) is eliminated for data that is only combined with local information and forwarded afterwards but not used locally (reduce operations, e.g.). Nevertheless, the approach still relies on message transfers.

The results indicate significant performance benefit compared to both RCKMPI, RCCE and even iRCCE, a non-blocking extension of RCCE [70]. The optimized reduce-scatter operation requires about 200 µs for a data size of several 100 elements. This is one third of the runtime required by the initially published RCKMPI library for the same task. Additionally, the performance of an Monte Carlo application was significantly reduced by applying the mentioned techniques. While those address collective operations only, these can be the base for an implementation of MPI's synchronization methods as observed in the survey of Section 3.3.

Different to the previous work, AL-KHALISSI ET AL. [117] focus solely on barrier synchronization for a possible OpenMP runtime. The authors take the findings from REBLE ET AL. [115] (see above) into account and avoid contention on the SCC's atomic hardware registers and the MPB by using a *"master-slave barrier"* and omitting centralized data structures. The authors present barrier algorithms that differ in the location of the allocated memory and usage of interrupts to release slaves from wait states.

The results in [117] show that the barrier latency can be slightly reduced by using flags that are stored in memory nearby the master. Slaves use remote put operations to indicate their participation in the barrier and the master polls its local memory to detect the changed flags. In the experiments, the MPB turns out to be the flag location that provides the lowest latency. The results also show that using interrupts instead of polling does not yield a better performance for the synchronization. Contention on the interrupt controller hardware is identified as reason for this.

A further optimization of the identified best barrier implementation is also presented. The authors omit the usage of the MPBT memory type that requires manual cache management, i.e. cache invalidation via the `CL1INVD` instruction and enforcing flushes of the write combine buffer (cf. Section 2.2.4). Instead, the authors use uncached memory accesses with the NCM memory type. As a result of this optimization, the time for a barrier is nearly halved, down to 18 µs for an barrier involving all 48 SCC cores. This is a lower value than REBLE ET AL. [115] reported. Follow-up work presented in [139] show additional algorithms that, among others, exploit unused LUT entries (cf. Section 2.2.6) but do not provide additional performance advantages.

Besides the SCC-specific optimization of barrier and collective operations, there exist publications that deal with the implementation and evaluation of these two aspects on other hardware architectures. However, most of them are not applicable to the SCC due to their hardware-specifics [111–114]. While TRÄFF ET AL. [113] deal with the implementation of the one-sided MPI routines on the non-cache coherent NEC SX-5 machine, the synchronization is based on messages (likely due to MPICH heritage) and therefore provides no new insights. However, the MPI/SX implementation follows a trigger-only approach since the *"communication calls [...] block until the target window has become exposed"* [113].

In conclusion, the literature presents large amount of research on barrier implementations and collective operations. The discussed research provides a solid base for the implementation of fence synchronization with the help of the mentioned techniques and hints to optimize synchronization in general. However, the general active target synchronization is generally not extensively discussed. Additionally, there are no attempts found in the literature to optimize this aspect of MPI on an nCC hardware architecture like the SCC. Therefore, a protocol for general active target synchronization is designed in the following section and provides one of the main contributions of this work.

3.4.3 Design Overview

The classification from Section 3.2 and the results of the conducted survey (see Section 3.3.6) revealed that immediate or trigger-only should be used within an implementation for an architecture that supports shared memory or RDMA transfers respectively. Since the SCC can be considered as shared memory system, those two classes are suited for this processor as they allow direct memory transfers.

The trigger-only variant should be favored as it is less prone to process skew (cf. p. 63). As a result, the synchronization scheme of MVAPICH for shared memory, that is also employed by FoMPI and MPI/SX (see above) at a conceptional level, is used as foundation for the design of the synchronization protocol.

The concept of a bit vector is used for synchronization at the beginning of access and exposure epoch. Different to the MVAPICH solution, the vector for signalling completion of an access epoch is omitted. Instead, the concept of completion counters is adapted to the SCC. The counters employ a both space and computational efficient way to check for the completion of access epochs.

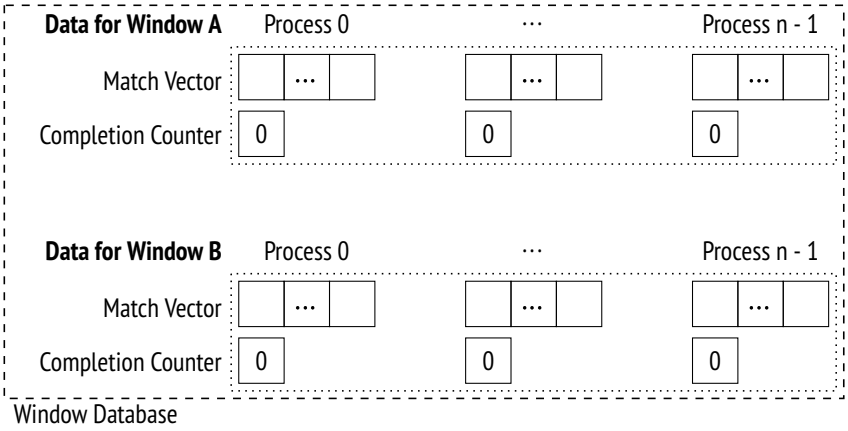


Figure 3.11: *Per-process/window data structures for GATS on the SCC.*

The bit vector and the completion counter can be stored inside a shared memory location of the SCC, although it is an nCC architecture. The usage of shared memory allows direct modification of the synchronization data structures and avoids the use of messages (see Section 3.4.1). It is shown in the following that implementing the scheme of LAI ET AL. [129] on the non-cache-coherent SCC is both feasible and efficient although the architectural properties needs to be considered carefully.

In the subsequent sections, both the concept and the actual implementation details for the SCC are discussed. First, the data structures are presented. Related to them is a detailed discussion on their physical placement. Finally, the required operations of MPI's GATS synchronization are discussed in more detail.

3.4.4 Data Structures

As outlined in the previous section, bit vectors and completion counters are the essential data structures used for the synchronization. The data structures are created on per-window basis, i.e. they are not shared among multiple windows. This is because the is the sole object that is used for synchronization in the MPI OSC API (see Section 3.1).

One completion counter and one bit vector are allocated per window and process as shown in Figure 3.11. It is required to allocate both data structures per process as it is unknown if a process becomes origin or target or even both.

The bit vector that is used at the beginning of the GATS procedure is called *match vector*. Similar to the match list introduced by GERSTENBERGER ET AL. [120], it is used to detect the `POST` operations that match the `START` call on the origin. The vector contains at least as many bits as processes in the window's communicator. The vector is logically owned by origin processes, as these are required to know when a target has synchronized. The bit at position x inside the vector is dedicated to the process with rank x inside the window communicator. If that bit is set, it indicates that the process with rank x has entered its exposure epoch.

Opposite to the match vector, the completion counters are logically owned by target processes since those processes need to know when access epochs are completed. The counter itself is an unsigned integer variable. If the counter has a value of zero, no accesses on the local window are performed. A non-zero value means that origin processes are still accessing the window. On the SCC, it has a size of eight bits.

3.4.5 Window Database

The synchronization concept requires that the aforementioned data structures can be manipulated directly to avoid message passing and processing. Therefore, they have to be allocated in shared memory or at least must be created such that all processes can access the data structures. Concerning the SCC, a process can reconfigure the LUT of its core to access a remote vector or counter.

One could argue that when using conventional DCM memory (by using `malloc/calloc`) the system address of the allocated data structures can be determined with the help of the modified Linux kernel. The determined address could then be exchanged and used by other cores to manipulate the LUT entries for further access. However, this approach is not suitable for an efficient implementation as local memory loads on the synchronization data are cached and there is no mechanism with low overhead for invalidation or flush for DCM to make local changes visible to remote processes.

As a result, the data structures need to be allocated with a different memory type such as NCM or MPBT variants (see Section 2.2.4) to facilitate management of the cache coherence. ROTTA [140] has shown that uncached memory access for small data structures expose lower latencies than cached memory accesses. AL-KHALISSI ET AL. [117] confirm these results. The usage of those memory types is only possible with the help of the according device drivers that allow to specify a physical address which is mapped into the virtual address space (cf. Section 2.2.6).

This functionality of the device driver is useful when it is combined with the legacy shared memory (cf. Section 2.2.6). The LSM has a fixed, thus well-known, address range and is shared among all cores in the default LUT configuration. Since it is mostly free (except for the POPSHM database, see page 29) it can be used for placing the synchronization data structures inside that memory. The region in which this data is placed is designated as *window database*.

Centralized Window Database

In a first approach, the 16 MB LSM in one of the four memory controllers (cf. Figure 2.2) is used to host the window database. The choice of the controller is arbitrary. By using this approach, a centralized database is created in which all match vectors and completion counters are placed.

For the allocation of the synchronization data, a simple "allocate and forget" allocator is used: At the beginning of the window database, an integer is stored that contains the beginning of the unused memory region in the database. Upon allocation, the offset is increased by the amount of reserved memory. However, the free operation does not change the offset and thus leaks memory.

While this approach is simple, other researchers observed that contention can occur on centralized, i.e. shared, synchronization data structures (see Section 3.4.2). To analyze the performance impact of the proposed centralized window database, an exemplary application is studied along with a prototype implementation of the synchronization that is based on the FoMPI (see Section 3.3.4) scheme from the literature [120].

Different to the literature, the free space protocol was omitted and replaced by linear scan for finding free entries in the match list. However, with the match list and the completion counters, similar data structures are used as in the proposed design.

The application that uses the GATS methods is a CA with a nine-point stencil (Moore neighborhood) that operates on a field of 1600 lines each containing 8192 elements of 8-bit integers. One-dimensional domain decomposition is employed such that every MPI process calculates equally sized portions of the compute domain. Excess lines are distributed evenly among processes to avoid load imbalances. The chosen domain dimensions ensure that the data does not fit into the cache which avoids super-linear speedup. The update of the whole compute domain is performed 50 times.

Because of the stencil, the domain update requires halo/ghost zone exchange between two neighboring processes. One-sided communication is used with GATS to perform this exchange. The exchange is performed as early as possible and overlapped with computation. That is, the boundaries to be exchanged are computed first and communicated concurrently with the computation of the remaining inner field.

However, to make the impact of synchronization and the centralized window database visible, the communication operations (i.e. PUTs) were removed for the measurements. The synchronization operations remained active for the purpose of this analysis. While the computational results of the application are invalid due to the missing communication, the performance gives insights in the impact of the synchronization.

Figure 3.12 shows strong scaling performance of the application in terms of speedup relative to the sequential run. The runtime represents the time required to compute the 50 iterations (see above). Overhead, for example from the initialization phase, is omitted from timing measurement. Each of the presented runtimes is the median of three gathered samples which did not vary by more than 5%. In addition, the memory controller that hosts the window database was varied. Further, an application version that does not use synchronization (and communication) was measured. It serves as an embarrassingly parallel baseline for comparing the performance with synchronization enabled. The mapping of the MPI processes to the cores was chosen such that the memory controllers domains are filled first (see Figure 2.2 on page 17).

The obtained results are manifold. First, the memory-bound application without synchronization scales linearly and is close to a parallel efficiency (speedup over number of parallel processes) of one. This proves, that the SCC's hardware, especially the memory subsystem do not present a bottleneck for the application. This confirms the results of VAN TOL ET AL. [141] who showed that all cores cannot saturate the system's memory controllers when a memory bound application is executed.

Second, the application performance is significantly affected when synchronization is active. Independent of the memory controller that hosts the window database, the speedup drops, especially for high core counts, down to about eight. This reveals a contention on the memory subsystem, since the synchronization basically adds memory accesses to the program.

Finally, the contention seems to appear at the memory controller or the on-chip network, but not at the cores as the performance drop appears at different core counts when the controller is changed. A reason for this might be the contention of the responsible memory controller or its attached router. This is because the controller

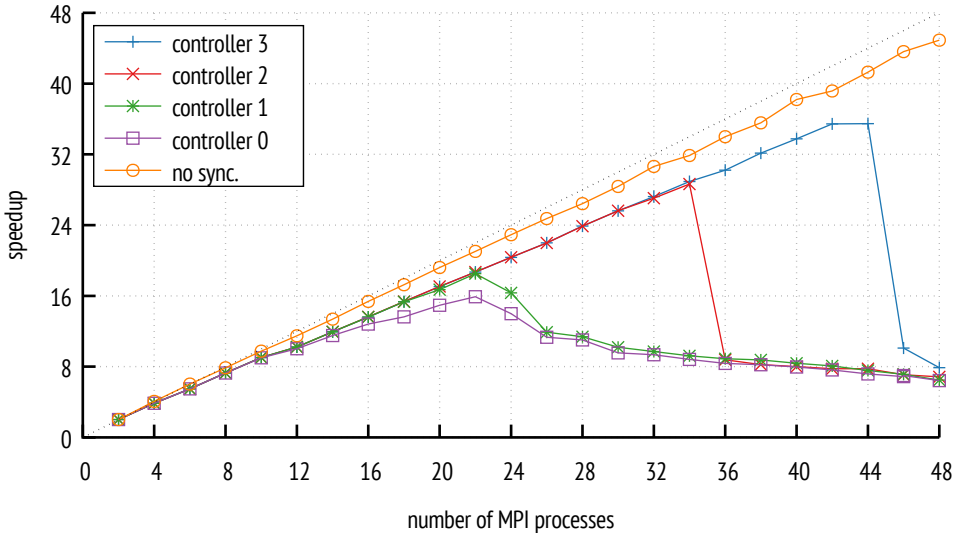


Figure 3.12: *Stencil application speedup with centralized window database.*

not only has to serve the accesses to the synchronization data structures but also the accesses from the memory-bound application. In combination, both types of access saturate the controller.

Also the locks that are used by the implementation to achieve mutual exclusion might be a reason for this, similar to the observations of [115]. An in-depth analysis of this behavior is out of the thesis' scope. However, the outcome of the experiment is clear: The centralized window database is not an option for the implementation of GATS.

Distributed Window Database

The preliminary results in the previous section indicate that a centralized window database is not suitable for the SCC or NUMA systems in general. The approach to put all synchronization data in the same memory location is straight-forward for an implementation. Nevertheless, it does not match the per-process nature of the data (cf. Figure 3.11). As a consequence, the centralized approach is dropped and a *distributed window database* is created instead.

Different to the centralized version, not a single but all four memory controllers serve as a storage for the synchronization data. Only the cores that belong to the

controller's domain (cf. Section 2.2.1) allocate storage in its memory. By doing so, the bit vectors and completion counters get closer to the core that actually owns the data. This has been found to be a beneficial approach in the literature (cf. Section 3.4.2 and [117]). Since the data is allocated in shared memory, it remains accessible by all cores.

The results of other research groups also revealed that the MPB [117] or the LUT entries [139] can serve as a location for the synchronization data and provide lower access latencies (see Section 3.4.2). Anyhow, these locations are not usable in the context of an MPI implementation on the SCC.

First, the MPB is a scarce resource on the processor. It is used by RCKMPI as transport medium. Thus, using the MPB for additional purposes requires changes in the MPI implementation and implies reducing the MPB memory size and therefore the performance of point-to-point operations.

Second, the LUT entries play a crucial role for defining shared memory. Consequently, wasting these entries for other purposes is not useful. In order to use the LUT entries for multiple purposes (define shared, store synchronization data), a resource management could be employed, but this is out of scope of this thesis. Thus the synchronization data is kept in the main memory.

To evaluate the performance of the distributed window database, the same methodology from the previous section is employed. The prototype implementation of the synchronization scheme that is based on FoMPI [120] was changed accordingly to follow the distributed database concept. Thus, the match list and the completion counter are distributed across the four memory controllers. Again, the performance is compared with a baseline application version that does not use synchronization. The speedup relative to the sequential version is presented for the strong scaling scenario in Figure 3.13.

The numbers confirm that the distributed approach does not lead to a performance drop, i.e. reduced speedup with low efficiency, for high core counts. The application now scales linearly for all numbers of started MPI processes. However, a slightly reduced performance compared to the version without synchronization can be observed. The reason is that synchronization adds serial overhead to the runtime which in turn reduces the efficiency. Nevertheless, the results motivate to follow the concept of a distributed database. It is therefore employed as synchronization data storage in the following.

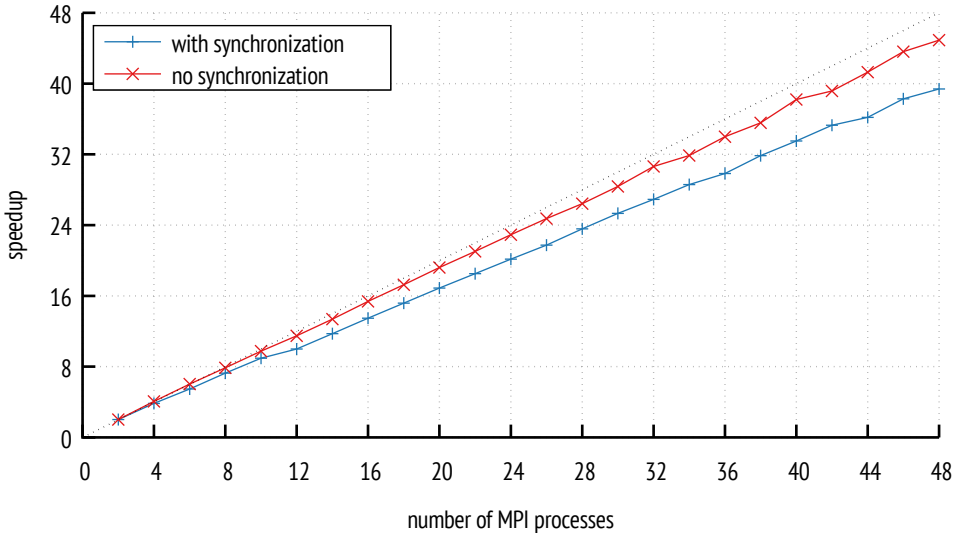


Figure 3.13: *Stencil application performance with distributed window database.*

3.4.6 Window Creation

The previous analysis revealed that the location for the synchronization data should be distributed. The per-process allocation of the match vector and completion counter in the distributed window database is performed during the collective window creation that precedes any OSC operation.

Independent of the actual creation call (such as `MPI_WIN_CREATE` or `MPI_WIN_ALLOCATE`), the implementation of this operation is comprised of two steps: allocation and address exchange of the allocated data. During allocation, memory for the match vector and the completion counter is reserved in the distributed window database. The obtained offsets in the database memory are made available in the exchange phase. The address exchange is required to enable all processes to access the created data.

Since window creation is a collective operation, so is the allocation of the synchronization data inside the shared window database. Therefore, the allocation itself must be synchronized among the participating processes that access the same portion of the distributed window database.

For the prototype implementation on the SCC, this implies that the allocation offset in the window database must be protected against concurrent accesses. This is achieved

by using the mutex-like test-and-set registers (see Section 2.2.5). The TSR of the core with an even ID that is next to the memory controller of the distributed database is acquired upon allocation and released afterwards. In the outcome of the allocation process, every process has allocated a chunk of memory and an offset in the window database's memory that addresses this chunk.

In the exchange phase, the obtained offsets are exchanged by a collective all-to-all operation. Thereby all processes know the offsets of the synchronization data structures of all window processes. Since the data resides in the LSM, which is shared by default, access to the synchronization objects is enabled. The conversion from the allocation offset that was exchanged during window creation to the system memory address is straight-forward. By adding the allocation offset to the start of a core's nearest LSM, the pointer to the data is obtained. As a result, every window process is able to address the match vector and completion counter after window creation.

A drawback of this approach is that the offsets/pointers of the allocated synchronization data is duplicated at every of the n processes of the window. Thus, the totally required memory scales with n^2. With a large many-core CPU in mind, i.e. the number of cores reaches the order of hundreds or thousands, this leads to significant memory overhead when a lot of processes participate in window creation. Ultimately, this hinders scalability in terms of the MPI implementation memory usage [142]. A solution for this issue would be to place the obtained offsets in the database as well. However, this aspect and further analysis of contention or scalability is not part of the discussion within this thesis.

3.4.7 Start and Post Operations

After discussing the storage prerequisites and the construction of the synchronization data, the next two sections describe the actual implementation of the shared-memory-based synchronization on the SCC. The discussion is separated into the operations at the beginning and the end of the epochs. Figure 3.14 illustrates the coarse steps of the implementation when two target processes synchronize with one origin. Details of the sketched steps are provided in the following.

The `MPI_WIN_START` and `MPI_WIN_POST` function operate primarily on the match vector that was allocated during window creation (see above). The pseudo-code for the two operations is shown in Algorithm 1.

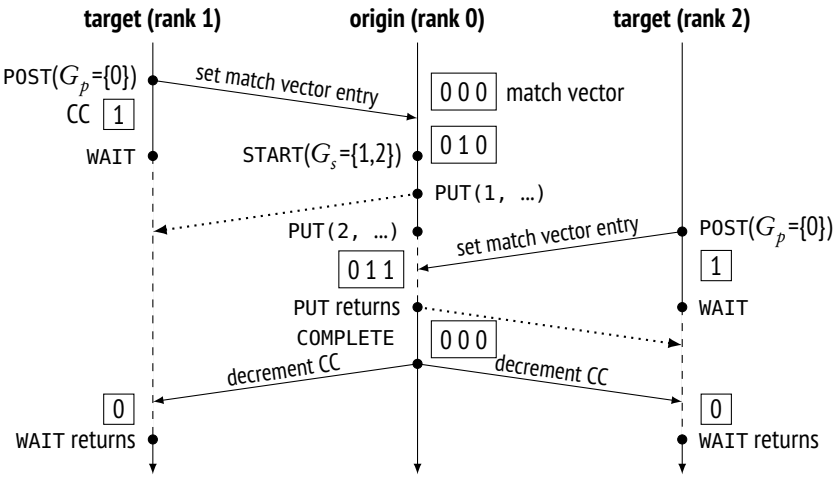


Figure 3.14: *Sequence diagram for the SCC implementation of GATS.*

When an `MPI_WIN_POST` operation is issued by a target process, it first initializes its completion counter (abbreviated as CC in the figure) to the number of processes in the post group G_p which contains all origin processes. Subsequently, the target iterates through the group and notifies the origins about its readiness for communication.

In the implementation, the group member ranks are translated into the IDs of the corresponding cores. To notify the origins, the TSR of the origin's core is locked to prevent concurrent modifications of the match vector. Subsequently, the byte containing the bit of the (target) process is read, updated, and written back using uncached memory.

Cached memory is unsuitable for this use-case since it would require an explicit write-back of the cache line that contains the loaded byte. Such an operation is not supported by the SCC's cores. Only a write-back including an invalidation of the whole cache is available in the instruction set. However, it is far slower than using uncached memory. In addition, the match vector is not accessed by the targets for any other purpose which makes caching unnecessary anyway.

On the origin side, the `MPI_WIN_START` function performs only bookkeeping operations for the implementation. In particular, the currently used synchronization type is stored in the window data structure since this is required by the upper layers of MPICH. Additionally, the ranks from the start group are translated to core IDs and stored in an array. These IDs are used later on to notify the targets about the

Algorithm 1 Pseudocode for `MPI_WIN_START` and `MPI_WIN_POST`

```
function MPI_WIN_START(G_s: Group)
    start_ranks ← core ID's of procs ∈ G_s
end function

function MPI_WIN_POST(G_p: Group)
    completion_counter ← |G_p|
    for all origins ∈ G_p do
        core ← CORE_OF_PROC(origin)
        LOCK_TSR(core)
        match_vector[core][local_rank] ← 1
        UNLOCK_TSR(core)
    end for
end function
```

completion. They have to be saved in `MPI_WIN_START` because this is the only function in the MPI OSC API that accepts the start group as parameter.

3.4.8 Polling the Match Vector

Since the synchronization ought to follow the trigger-only synchronization (see Section 3.4.3), the match vector is not polled in the `MPI_WIN_START`. Instead, it is checked in the communication calls. There, the target's bit is polled until it has been set.

Polling is performed with uncached memory since cached reads would prevent the origin to observe a post operation. In addition, caching of the match vector is dangerous in any case.

Suppose a match vector has been loaded into the origin's cache and that the origin can communicate with all targets of the current access epoch, i.e. all targets of the current access epoch have synchronized. Independent from this, another target of a subsequent access epoch (but of the same window) performs its `MPI_WIN_POST` call and modifies the match vector in main memory. Due to the missing coherence, the origin's cached copy remains unchanged, and becomes stale. In case the origin's cache evicts the line (due memory accesses by computation or due to a context switch and operations of another process) which contains the outdated match vector, it overrides the manipulated match vector in the main memory.

This would cause a deadlock. The origin would check for a POST operation that actually took place but its effect was destroyed by the cache line eviction. Consequently, this leads to an infinite loop at the origin. Therefore, uncached read accesses are the most useful variant in this case since they ensure the observation and correct manipulation of the bit vector.

However, to speed up polls of already synchronized targets, a local and cacheable copy of the match vector is used. Inside a communication call the cached vector copy is checked first. If there was no POST operation, the uncached match vector is polled and upon detection of the target's post operation, the cached copy is updated.

3.4.9 Complete and Wait Operations

At the end of an access epoch, i.e. in `MPI_WIN_COMPLETE`, each origin resets the match vector entries corresponding to the targets in start group G_s. This applies both to the uncached vector in main memory and the locally cached copy (see previous subsection).

Subsequently, it decrements the targets' completion counters. However, this notification at the end of the access epoch can only be performed after an origin has successfully started its matching exposure epoch. Only then, the target's completion counter is in a valid state and can be modified when the origin completes. Thus, an origin first ensures that every targets in G_s has synchronized (see Algorithm 2).

The origin then iterates over all targets and decrements their completion counter. Similar to the targets' accesses on the match vector, the completion counter is accessed with uncached memory (Algorithm 2). To make the decrement atomic, the TSR of the target's core are used again.

On the target's side, the `WAIT` call polls the completion counter with uncached memory as well to observe the decrements made by the origins. Polling is performed until the counter reaches zero (see Algorithm 2).

Algorithm 2 Pseudocode for `MPI_WIN_COMPLETE` and `MPI_WIN_WAIT`

```
function MPI_WIN_COMPLETE
    for all targets ∈ start_ranks do                    ▷ see MPI_WIN_START
        repeat                                          ▷ busy wait for all targets
        until match_vector[local_core][target] == 1
        match_vector[local_core][target] ← 0
    end for

    for all targets ∈ start_ranks do                    ▷ notify all targets
        core ← CORE_OF_PROC(target)
        LOCK_TSR(core)
        CC[core] ← CC[core] − 1                         ▷ decrement remote CC
        UNLOCK_TSR(core)
    end for
end function

function MPI_WIN_WAIT
    repeat
    until CC == 0                                       ▷ poll local CC with uncached reads
end function
```

3.4.10 Summary

Putting the SCC-specific details aside, the previous subsections presented a shared-memory-based synchronization scheme for an non-cache-coherent many-core chip. It employs bit vectors and completion counters which are concurrently allocated during window creation. To load balance memory accesses, the data is distributed among the memory subsystem to keep the data structures close to their respective owner. For remote manipulation of the data, uncached accesses are employed to avoid more expensive cache flushes of data that is accesses once per synchronization cycle only. However, concurrent accesses to the data structures need to be synchronized by other means (like locks or atomic modifications). Uncached accesses are employed for local polling of the synchronization data to observe the modifications by remote sites. Thus, the approach avoids unnecessary caching of the synchronization data for non-cache-coherent systems.

3.5 Experimental Evaluation

In the following, the implementation of the concept from the previous section is evaluated. This includes both tests on the functionality as well as an experimental investigation on the implementation's performance.

For the implementation of the synchronization approach, RCKMPI based on MPICH 3.1.3 was used (see Section 3.4.1). To override the message-based CH3 synchronization functions, according function pointers in the internal MPICH window object were modified during window creation. The default window creation functions were overridden by setting function pointers inside the `MPIDI_CH3_Win_fns_init` function that is supplied by MPICH's CH3 layer (cf. Figure 3.3 on page 65) especially for that purpose.

It has to be noticed that the changes in the synchronization were implemented alongside with the communication from the next chapter. This is due to the tight bonds between synchronization and communication in MPICH's message-based CH3 implementation. As a result, solely switching away from a message-based synchronization would also require changes to the communication part of the CH3 layer. For efficiency reasons, this was not performed but the communication was reimplemented without the CH3 legacy. However, for the sake of the presentation, the following discussion deals only with the synchronization aspect of the implementation. Care is taken to not include actual one-sided communication in the evaluation.

3.5.1 Environment

All the experiments were conducted on the SCC system that is lent by Intel to the Operating Systems and Distributed Systems research group at the University of Potsdam. It is equipped with 32 GB of RAM. The default frequency settings for the system are used. That is, cores operate at 533 MHz whereas the on-chip network routers and memory controllers run at 800 MHz. While higher frequency settings are configurable, instabilities ranging from spontaneous OS crashes to non-booting cores were observed with these higher frequencies. Therefore, these were not considered.

The employed software packages are listed in Table 3.3. The MPI library was compiled with optimization enabled (Level 2), strict compilation mode, and without `C++` and Fortran support as there was no need for these language bindings. For `MPI_Wtime`,

Table 3.3: *Employed software components for the experimental evaluation.*

Component	Software Package	Version
SIF FPGA bitstream	Intel sccKit	1.4.2.2
operating system	Linux with sccLinux patches	3.1.4
user land	BuildRoot-based with glibc and BusyBox	2011.11
cross compiler	GNU Compiler Collection (C Compiler)	4.4.6
MPI library	MPICH with RCKMPI CH3 channel device	3.1.3

the library was configured to use the `RDTSC` machine instruction. Thus, it provides accurate timings on the SCC. On all experiments, care was taken to provide a low-noise environment. Hence, all unnecessary processes on the SCC were terminated. Those include the syslog processes, mouse emulation, and SCC-specific CPU usage reporting tool. Especially the last one is critical as is periodically performs file system operations as well as memory accesses inside the LSM.

3.5.2 Functional Tests

To verify the functionality of the new implementation for GATS, the MPICH test suite was used. It contains twelve test cases that include the implemented MPI synchronization scheme. Except for `nullpscw`, all of them also use communication. In case the employed communication is supported by the implementation (see next chapter), the tests were included into the test set. Finally, these are (with number of used processes according to the MPICH test suite): `manyrma2` (2, with `-put` and `-pscw` arguments), `nullpscw` (7), `pscw_ordering` (4), `test2_am` (2), `test3_am` (2), `transpose3` (2), and `wintest` (2). Thus, seven out of twelve tests are used for testing.

The tests were executed with the `runtests` Perl script from the MPICH test suite. In the outcome, all of the named test cases completed successfully. This indicates that the implementation and the underlying algorithm of the synchronization fulfill the requirements of MPICH and thus those of MPI, although the number of tests is comparable small.

3.5.3 Benchmark Methodology

Besides their functionality, the performance of synchronization in general and GATS in particular is critical for MPI OSC applications. Therefore, both the performance and scalability of the according MPI synchronization routines `MPI_Post`, `MPI_Start`, `MPI_Complete`, and `MPI_Wait` is evaluated. In addition, a comparison with RCKMPI's message-based implementation of GATS is performed to assess the performance. The latency of each of the operations is chosen as the performance metric. The number of processes that synchronize with a target is selected as the main factor in the following experiments.

With a benchmark that uses both communication and synchronization of the MPI OSC API meaningless results will be generated if two different implementations are compared. For example, with MPICH's/RCKMPI's original deferred synchronization, the final synchronization call actually performs the queued communication operations. In such a case, the true costs, i.e. the required time, of process synchronization are hidden. Therefore, no communication should be performed by the benchmark.

Considering existing benchmarks, the *OSU Micro-benchmarks* [131] basically focus on the performance of communication operations, like `PUT` and `GET`, in terms of latency and bandwidth. The benchmarks generally measure the time required to perform the communication including the synchronization, but the suite does not include benchmarks that solely analyze the performance of any MPI synchronization scheme. The same applies to the Intel MPI Benchmarks[10] (IMB). As a result, both are unsuited for the experiments.

Different to the pre-defined micro-benchmarks of the OSU suite, *SKaMPI* [143] allows to create MPI benchmark scripts that are interpreted upon execution. Those can include calls to predefined measurement routines that benchmark some of MPI's functions. Concerning the API's OSC fraction, all synchronization routines can be measured. However, the environment in which the routine is measured is defined in the according built-in function of the benchmark. For `MPI_WIN_POST` this means that the post group parameter, the behavior of the origins, and a barrier used for synchronization are predefined inside the SKaMPI source code. In addition, each time when a synchronization method like `MPI_WIN_POST` is measured, a window is created and destroyed which does not match an application's behavior.

[10] https://software.intel.com/en-us/articles/intel-mpi-benchmarks/, last accessed 2016-07-25

```
1  for (i = 0; i < NUM_ITERATIONS; i++) {
2      if (comm_rank == 0) {
3          ts[i] = time_of(MPI_Win_start(start_group /* = rank 1...n-2 */, 0, win));
4          /* access epoch (nop) */
5          tc[i] = time_of(MPI_Win_complete(win));
6      } else {
7          tp[i] = time_of(MPI_Win_post(post_group /* = rank 0 */, 0, win));
8          /* exposure epoch (nop) */
9          tw[i] = time_of(MPI_Win_wait(win));
10     }
11 }
```

Listing 3.5: *Pseudocode of the GATS microbenchmark.*

As a consequence from the above, a simple benchmark that involves only synchronization and no communication (unlike OSU or IMB) is created. Different to SKaMPI, a single window is created once at the beginning of the benchmark. The started processes synchronize on that single window repeatedly in a tight loop. By doing so, the existence of contention effects like those observed by REBLE ET AL. [115] can be examined.

The benchmark records the time each of the four GATS API calls requires to complete its operation. For timekeeping, `MPI_Wtime` is employed. The gathered timings are recorded for 1001 iterations (arbitrary choice) to get statistical valuable data. Pseudo-C code of the benchmark is shown in Listing 3.5. The real code was compiled with optimization enabled (`-O2`).

During one iteration, a single origin process, which is always the process with MPI world rank 0, starts its access epoch using `MPI_WIN_START` to $k = n - 1$ target processes, where n denotes the number of started MPI processes. On the other hand, the target processes start their exposure epoch with `MPI_WIN_POST`. They expose the window only to the single origin process. Immediately after starting their epochs, the processes close them by calling `MPI_WIN_COMPLETE` and `MPI_WIN_WAIT` respectively.

A barrier at the beginning of the loop's body (as it would be used by SKaMPI) is intentionally omitted for two reasons. First, all processes are synchronized after one iteration of the loop. Thus, invoking the barrier is redundant. Second, using a barrier affects the measurement results as it is unclear (without further knowledge of the implementation) which process leaves the barrier at which point in time. For example, targets may leave the barrier earlier than the single origin process, which causes process

skew and thereby a delay of MPI_WIN_START. As a result, the timings for MPI_WIN_WAIT (MPI_WIN_POST does not wait for START) are affected. Without a barrier this negative effect is avoided.

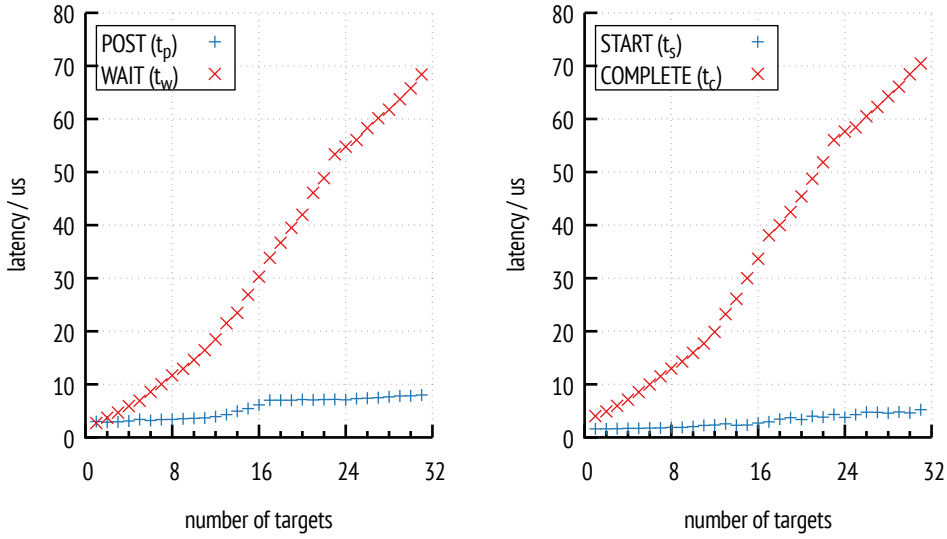
3.5.4 Scaling

With the described benchmark, a strong scaling analysis of the GATS calls latency is performed first. Starting from two processes, the number of MPI processes was increased up to a number of 32. The processes were mapped to the core with the corresponding MPI world rank (Figure 2.2). Consequently, the number of targets k was varied between one and 31 while keeping the number of origins at one for all experiments. More targets were not started, because using nearly all of the 48 possible processes on the SCC questions the usage of GATS, which is more fine-grained than fence synchronization (see Section 3.1).

In the following, the presented numbers are the medians of the recorded timings. In case of the targets, all 1001 samples $t_{p,i}$ and $t_{w,i}$ from all k target processes are collected and the medians t_p and t_w are obtained from the $k \times 1001$ samples. Besides outliers caused from process scheduling of the OS, only little deviation from the reported values was observed: For all measurements, the first and third quartile of the four measured timings never deviated by more than 5% for origin and 10% for targets from the reported median. Figure 3.15 shows the obtained latencies of the GATS calls split between the targets (a) and the single origin (b).

The presented results show a nearly constant runtime for the MPI_WIN_START and MPI_WIN_POST operations. However, for an increasing number of targets, the latency of MPI_WIN_POST increases as well, starting from 3 µs for a single target to 8 µs for all 31 targets. The reason for this increase can be attributed to the increasing distance of the targets to the origin core and therefore its test-and-set register which must be acquired in order to modify the match vector. Thus, it requires more hops on the on-chip network to read (and modify) the TSR resulting in higher access times that has been observed by REBLE ET AL. [115].

Moreover, the more processes participate in the synchronization, the more they compete for acquiring the TSR of the origin's core. REBLE ET AL. [55] observed contention on the TSR that caused exponential increasing latency when more than 24 cores compete for a spin-lock. TSRs are used for the GATS implementation as well. Thus, the impact of TSR acquisition is analyzed more deeply.



(a) *target processes* (b) *origin process*

Figure 3.15: *Scaling of GATS functions on the SCC.*

To analyze the TSR acquisition impact, a new metric is introduced. The metric counts the number of tight loops required to successfully acquire the TSR during `MPI_WIN_POST` (see Algorithm 1 on page 96). Using the same setup for gathering the timing data, the average number of loops to acquire the lock is measured per target process. The first row in Table 3.4 shows the median number of the per-process average loop count among the given number of target processes. The second row lists the total time required to acquire the TSR, modify the match vector, and release the TSR. Those steps constitute the essential steps for synchronization. The total time to modify the match vector is obtained in the same way as the number of acquisition loops, i.e. the median of the average per-process timings is gathered. To allow a better comparison with the literature, the number of targets of is increased to 47, meaning that all cores (including the single origin process) were in use.

The data reveals, that both the number of acquisition loops and the time required to modify the bit vector, i.e. to notify the origin, expose no scalability issues. Opposite to the observations in REBLE ET AL. [115], both measures do not increase exponentially but roughly linearly. In addition, the observed overall latency of the `POST` operation is still far below the latency of the simple and contention-prone spin-lock implementation shown in the publication of REBLE ET AL. Furthermore, `MPI_WIN_POST` is faster than the optimized tournament spin-lock implementation from [115] that avoids

Table 3.4: *Selected metrics of the* `POST` *and* `START` *operations.*

	Number of Targets							
Metric	1	2	4	8	16	24	31	47
`POST`: number of acquisition loops	0	1	1	7	26	27	33	69
`POST`: vector modification time/µs	0.8	1.1	1.1	2.1	4.9	5.7	6.7	12.5
rank conversion time/µs	1.3	1.3	1.4	1.6	2.5	3.6	4.1	5.1
time for `MPI_WIN_START` t_s/µs	1.7	2.0	2.3	2.1	2.8	4.3	5.3	5.8

contention and takes around 20 µs for 24 and 48 cores respectively. SCOSCo's POST operation requires only 5.7 µs and 12.5 µs (see Table 3.4) to modify the match vector which includes lock acquisition and release.

These results lead to the conclusion that the implementation of the POST operation does not suffer from contention on the TSRs. This can be explained by the synchronization that follows in the `COMPLETE`/`WAIT` function call pair because this step prevents the measurement loop of the micro-benchmark from being a tight one. Each MPI process is delayed by that synchronization and thus breaks the tight loop. Consequently, in real applications, where access and exposure epochs are unlikely to be empty (as in the benchmark), there will be no tight loop as well and thus no contention on the TSR that is caused by the MPI implementation.

In case of `MPI_WIN_START`, there is again a very slight increase in latency from 1.7 µs to 5.8 µs (see Table 3.4). Nevertheless, there is no operation concerning the actual synchronization as the implementation follows the trigger-only concept (see Algorithm 1). The increase in latency is due to the conversion from ranks in the start group to ranks of MPI's world communicator which is required to derive the core number in later RMA accesses and in the `MPI_WIN_COMPLETE` call. Since the number of targets increases, the conversion needs more time as shown in the last row of Table 3.4. The data also reveals that the time for the rank conversion contributes about 80% to the time required for the `MPI_WIN_START` call.

Comparing the latency of the `POST` operation with the one of the rank conversion shows that both are in the same order of magnitude. Further, the time for the match vector modification is even smaller for target counts up to four. This emphasizes the lightweight implementation of the `POST` operation as it is comparable fast as a conversion of numeric identifiers.

Different to the two epoch starting routines, `MPI_WIN_COMPLETE`'s and `MPI_WIN_WAIT`'s runtime clearly exhibit linear scaling (see Figure 3.15). Concerning the `MPI_WIN_COMPLETE` call, this has two reasons. First, the origin needs to notify all targets which is done in a loop and thus causes linear scaling behavior. To do so, it has to wait for all targets to be synchronized (cf. Algorithm 2 on page 98) which also scales linearly. As a result, the behavior of origin affects the `MPI_WIN_WAIT` on the target side which shows linear scaling as well.

3.5.5 Comparison with MPICH/RCKMPI

Next, the performance of the presented approach is compared against MPICH's message-based synchronization. RCKMPI is chosen as reference implementation, i.e. as performance baseline, since it is already tuned to the SCC and uses the fast on-chip MPBs to transfer messages including those for GATS (cf. Section 3.4.1).

The micro-benchmark and the methodology from the previous sections were reused. However, the recorded median times of the origin, i.e. t_s and t_c, were summed, giving the total time t_o required to perform the GATS synchronization on the origin side. The same was done for the recorded median of the target times (t_p and t_w) leading to t_t. Figure 3.16 shows the obtained times t_o and t_t for both RCKMPI and the optimized shared-memory synchronization protocol for different number of targets.

The results reveal that despite RCKMPI's usage of the fast on-chip MPB, the performance of the message-based synchronization from MPICH delivers latencies approximately five times higher (e.g., 10 targets: 17.9 µs vs. 89.2 µs) than the presented optimizations. Since both implementations share the same library infrastructure of MPICH, they also experience the same software layer overheads. Hence, the difference in the aggregated latency are caused by the implementation of the synchronization itself. As a result, the higher latency of RCKMPI can be attributed to the messaging overhead illustrated in Section 3.4.1.

The significant differences between RCKMPI and the developed synchronization scheme underline that even in presence of a tuned implementation for message transfer, a shared memory approach is more appropriate on the SCC. Bearing in mind, this is possible without hardware support for cache coherence which is in fact not needed in the presented approach as it uses uncached memory accesses.

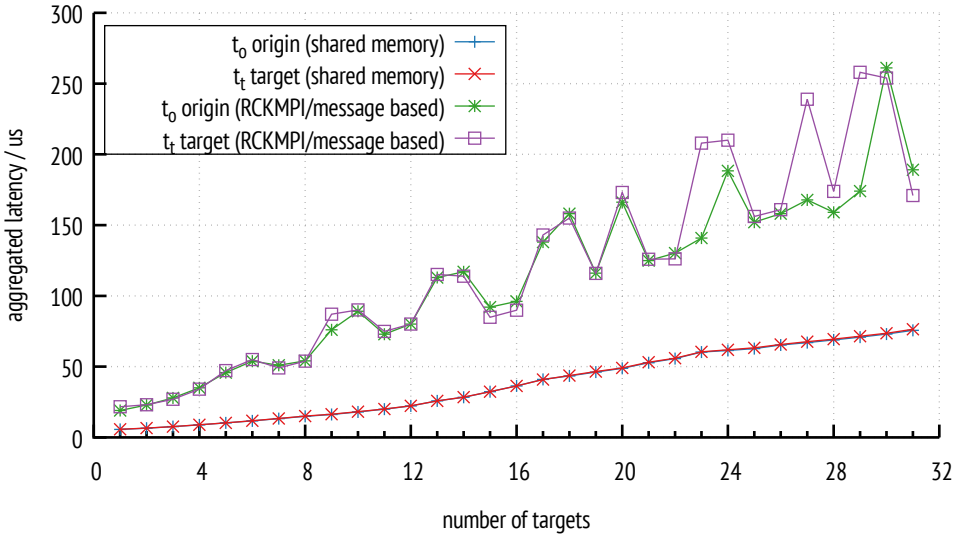


Figure 3.16: *Performance comparison for general active target synchronization.*

3.6 Summary

This chapter presented the synchronization aspect of MPI's one-sided communication API, especially its implementation on the non-cache-coherent Single-Chip Cloud Computer. Based on the literature, an existing classification scheme for implementations has been extended. The newly defined trigger-only class has been identified as most beneficial one for the shared-memory system, even in the absence of cache coherence. It enables data transfers as early as possible and thereby overlap of communication and computation. Additionally, it is more tolerant of process skew than an immediate synchronization style.

Existing MPI libraries and their implementation of OSC synchronization were studied and classified according to the scheme. As a result of the survey, a severe implementation error has been identified in Open MPI, one of the best known MPI implementations. Common concepts for implementing general active target synchronization have been identified. Among them are completion counters which are used to identify the and access epochs and vectors/lists that are used to detect active exposure epochs of remote processes. Those concepts are shared by well-established MPI implementations. However, none of them provides an implementation for a non-cache-coherent shared memory system. This gap is closed by the presented synchronization scheme.

Although originated from a cache-coherent architecture, it was shown that it can be adopted to a non-coherent architecture.

Uncached memory accesses are used to poll and modify the synchronization data structures. Caching has been identified as unnecessary mean for the task of synchronization in case for nCC systems. Despite the uncached memory accesses, the shared memory approach clearly outperforms the message-based implementation on the SCC which was designed with support for message passing. That way, the feasibility and efficiency of a synchronization scheme for non-cache-coherent shared memory system has been demonstrated.

With this, one of the two building blocks of MPI one-sided communication has been investigated. In the next chapter, the communication itself is discussed.

4 Software-Managed Cache Coherence for MPI One-Sided Communication

In the previous chapter the first of the two aspects of MPI one-sided communication, namely the synchronization, was discussed. An efficient scheme that does not rely on cache coherence was designed and implemented for the Intel Single-Chip Cloud Computer. Now, this chapter deals with the second aspect, the communication.

The focus of the discussion is on the design of an efficient communication scheme for the SCC. Although the SCC has been designed with message passing in mind, a closer look at both the MPI OSC programming model and the SCC hardware reveals that an implementation based on shared memory is more suitable for the architecture than a message-based. However, the SCC does not support cache coherence from the hardware side. Hence, the primary challenge is to manage this aspect within the context of one-sided communication in software. The development of a solution to this problem is the second major contribution of this thesis.

In the following section, a brief introduction into the communication part of the MPI OSC API is given along with an analysis of the existing implementation of one-sided communication on the SCC. From these findings, the *software-managed cache coherence for one-sided communication (SCOSCo)* is developed in Section 4.2. In Section 4.3 the implementation of the developed SCOSCo approach is presented. Further, its performance is evaluated using micro-benchmarks and applications in Section 4.4 which also reveals a hardware malfunction of the SCC. Conclusions for the design of future hardware systems and possible optimization are drawn in Section 4.5 and 4.6 which are followed by the conclusion.

The contributions presented in this chapter have been published in the Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores of the PPoPP conference [5] and of the 6th Many-Core Research Community Symposium [4]. They have been presented in a talk during at the workshop and the symposium, respectively.

4.1 Background

The next sections provide an overview of one-sided communication from the perspective of both the MPI standard and its implementation for the SCC, in particular RCKMPI.

4.1.1 MPI One-Sided Communication

As pointed out in Section 2.3.5, MPI defines one-sided communication as an alternative scheme to the point-to-point communication. Essentially, communication and synchronization are split from each other. While the synchronization was widely discussed in the previous chapter, the communication is in the focus of the following discussion. Here, the term *communication* refers to the operations that are listed in Section 11.3 of the MPI 3.1 standard [65], e.g. `MPI_PUT`, `MPI_GET`, and `MPI_ACCUMULATE`. Different to this, *one-sided communication* names the overall concept including the synchronization.

Communication Operations

Operations defined by the MPI standard can be divided into two classes. Within the first class, methods that move data between a target and origin process are defined. The second class provides operations to combine data at the target side with data from the origin. Some of those operations also return the targets data before the combination was performed. All of the operations have in common that they have to be non-blocking. There is no guarantee that the operation is finished after the function has returned [65, §11.3]. Consequently, the application-provided buffers are not available until the operation has been completed, i.e. synchronization was performed. A summary of the communication operations is illustrated in Figure 4.1.

The data movement operations include `PUT` and `GET`. The two methods simply move data of given size and MPI data type between the origin and the target. `PUT` replaces data in the targets window with the data provided by the origin. Vice versa, `GET` fetches the addressed data from the target's window into the origin's private memory. Note that for any communication operation, the data on the origin side do not have to reside in a window.

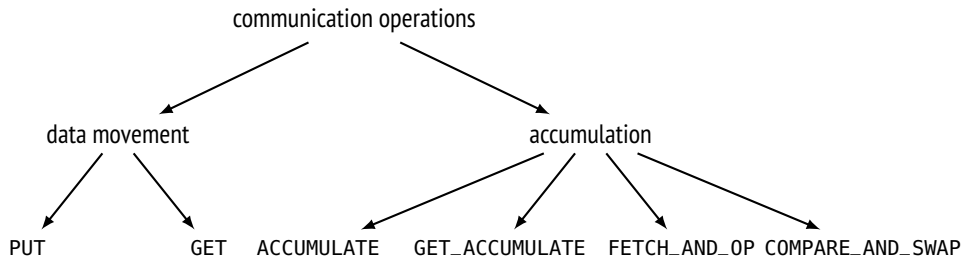


Figure 4.1: *Communication operations for MPI one-sided communication.*

To facilitate modifications, `ACCUMULATE` is provided by the standard and is a presentative of the second class of communication operations which is denoted as accumulate function [65, §11.3.4]. Instead of replacing the data in a target's window via `PUT` it combines the remote and locally provided data with predefined operations, like addition, multiplication etc. As a further extension `GET_ACCUMULATE` retrieves the data from the target's window before the subsequent modification.

`FETCH_AND_OP` is similar to `GET_ACCUMULATE` but restricts the usage to one element of data of a single data type. Due to these restrictions, the implementations can benefit from better support of the underlying hardware if it provides according routines to fetch and manipulate machine words.

Finally, `COMPARE_AND_SWAP` can be used to atomically set a value in the targets window if it matches a compare value provided by the origin. If both compare value and the addressed remote value are equal, the target data is atomically replaced by a third value. In any case, the original value in the targets window is returned to the origin. The semantics of `COMPARE_AND_SWAP` is similar to, e.g., the `CMPXCHG` machine instruction of the IA-32 architecture that can be used as foundation for synchronization data structures, such as mutexes or semaphores. However, the MPI call restricts the usage of datatypes to integers.

Except for the `FETCH_AND_OP` method, all of the above calls have a request-based counterpart. That is, the operation can be issued and returns immediately, like `MPI_ISEND` does. Similar to this non-blocking communication call, the request-based counter parts return a handle that can be used in combination with the `MPI_WAIT` or `MPI_TEST` function to check for their completion. That way, the completion of a certain OSC operation can be ensured. However, the usage of those request based operations is restricted to passive target communication (see Section 3.1) which are not discussed in this thesis.

Memory Models

The aforementioned communication operations manipulate data of a target process's memory that is exposed by the means of a window. However, when the hardware is considered, the question arises which part of the memory subsystem is affected by these operations and when operations become visible at either side of the communication. To abstract from hardware, the MPI standard assumes that data of a window can exist in private and public memory at the same time. The data that resides there is called *private copy* and *public copy* accordingly. [65, §11.4]

The public memory is globally accessible memory. Shared memory or memory regions registered to InfiniBand can be considered as an example of this class. Every process that has the required information, i.e. the address of the shared memory or the InfiniBand remote key respectively, can access those regions.

Moreover, systems mostly have faster but local memory that is not accessible globally but can be part of the memory hierarchy. Especially, caches inside the memory hierarchy can be considered as an example of such private memories where a private copy of a window data can be stored.

In addition to these types of memory, the standard further defines two models which describe if the two copies are affected by communication operations. By definition, the operations manipulate on the public copy since this is accessible for the other processes. Thus, the remaining question is how changes to the public copy propagate to the private copy.

Until version 3.0 of the standard, the Message Passing Interface defined only a single model, namely the *separate memory model*. Within this model, the two copies are considered to be logically separate. That is, the private copy is not automatically updated when the public copy gets modified. Vice versa, when the private copy is accessed, the changes it will not become visible until according action is taken.

Starting with version 3.0, the standard added a *unified memory model*. This one is the opposite of the separate model. When either the private or the public copy gets modified, the other is updated automatically. To provide an example, if a `PUT` operation modifies the public copy of a window, the private copy will be updated as well. A subsequent load operation will — opposite to the separate window model — retrieve the recently modified data. Figure 4.2 illustrates the differences. The motivation for the unified memory model in MPI was to support shared memory systems which

(a) *Unified memory model.* (b) *Separate memory model.*

Figure 4.2: *MPI's separate and unified memory model (based on [73]).*

provide transparent, i.e. hardware-based, cache coherence and can be considered as the most prominent building block in contemporary HPC environments.

Concerning the SCC, only the separate memory model can be applied. Due to the missing cache coherence, the caches of the SCC's cores are not synchronized with any other memory component of the system. If a modification occurs in either of the components it will not become visible in the other one. Consequently, only the separate window model is considered for the SCC in the remainder of this thesis.

Semantics

The communication operations of the MPI one-sided communication API are required to be non-blocking. This also applies to the calls that do not return a request handle. Due to their non-blocking nature, the completion of the issued communication operations must be ensured differently. In addition, the MPI standard defines restrictions for performing communication operations in order to be correct. [65, § 11.7]

For those calls (see above) that do not return a handle, their completion cannot be ensured until the access epoch ends. That is, a final synchronization method needs to be issued which closes the access epoch (e.g. `MPI_WIN_COMPLETE`). After that, only the local completion at the origin is ensured, so local buffers can be re-used. For the target, a method which closes the exposure epoch (e.g. `MPI_WIN_WAIT`) must be called to complete the operation and thus make changes visible.

Concerning the update of the private and public copy when one of them was modified, the standard defines the following semantics for the separate window model that

applies to the SCC: Local store operations that modify the private copy become visible in the public copy only when the exposure epoch starts, e.g. `MPI_WIN_POST` is called. Similar, changes by `MPI_PUT`s or accumulate calls apply to the public copy. They become visible in private copies only after the exposure epoch has been ended, e.g. by calling `MPI_WIN_WAIT`. To summarize, the synchronization operations not only synchronize processes but also update the private and public window copy in the separate window model.

In addition to the semantic concerning the completion of the communication operations, the MPI standard defines further restrictions for those operations. For example, multiple modifications of the same window location result in undefined behavior. While it was forbidden to perform such accesses in MPI-2, it has been changed to be undefined since MPI-3. This can be considered as result of criticism on the previous API version which prevented the implementation of PGAS languages on top of MPI [65, p. 454][84, 144]

Opposed to the undefined behavior of concurrent `PUT` operations, the outcome of multiple accumulate operations is defined. That is, if multiple of these operations (like `FETCH_AND_OP` or `COMPARE_AND_SWAP`) are concurrently issued on the same destination with the same data type, the result is as if the operations were issued *"in some serial order"* [65, p. 461]. The atomic modifications are defined element-wise not for the complete portion of the window specified by the function arguments [65, §11.7.2].

However, the most important restrictions are provided by the rules S1–S3 and U1–U5 which a program must follow to use MPI correctly. Essentially, those rules forbid local operations on an exposed window that might interfere with RMA operations. That is, while an exposure epoch is active, neither local load nor store operations should be performed. This is not restricted to the portions of the window that will actually be modified but applies to the complete window. [65, p. 454 ff.]

4.1.2 One-Sided Communication in RCKMPI

As pointed out in Section 2.3.3, RCKMPI, the SCC-specific MPI implementation, uses the Message Passing Buffers to transfer data. Because it is derived from MPICH the transfer is based on messages This applies not only to the point-to-point communication scheme or the synchronization for one-sided communication (see Section 3.3.1) but also the communication operations of the OSC API.

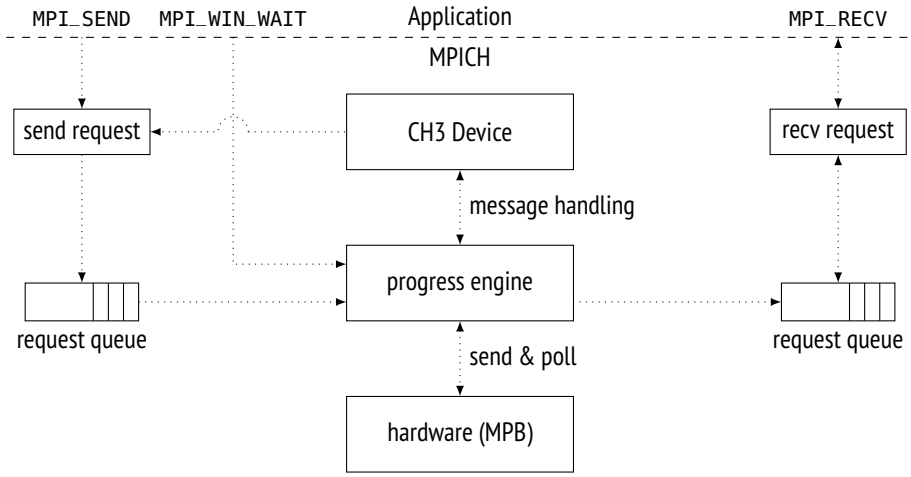


Figure 4.3: *Message reception and delivery inside MPICH.*

Implementation of One-Sided Communication

As a consequence of the message-based implementation, the realization of one-sided communication requires active participation on the target side. This contradicts the goal of the scheme to not involve the target in the communication operations. However, achieving the goal of portability justifies this design decision of MPICH, since truly one-sided communication is not possible on every platform, e.g. in cluster computers connected via standard Ethernet. Thus messages are used for the implementation.

Since MPICH also employs the deferred synchronization scheme (cf. Section 3.2.1), the arguments of a function call are stored inside a queue rather than being executed right away. The deferred synchronization assists this approach, since it is not possible to execute operations within that scheme when the synchronization is not yet performed.

Figure 4.3 shows the basic principles of the message reception and delivery by the progress engine of MPICH's CH3 device. Point-to-Point send operations as well as one-sided operations, due to their message-oriented implementation, create so-called request objects. These are generic containers for outstanding operations such as sends and receives. When a send operation is issued, a send request is created and stored in a queue from where it is picked up by the progress engine. The progress engine needs to be actively triggered. Most communication operations do so to ensure progress or

even completion of the issued operation.

When the progress engine is invoked, it processes outstanding send requests from the queue by transmitting them over the underlying hardware. In addition, the hardware is also checked for incoming data. When an MPICH message is completely received, it is handed over to the CH3 device layer for further processing. The responsible routine is determined by a type field of the message's header which makes this approach similar to active messages. During processing, the message handler decides whether the request is completed, is stored in a queue for pickup by receive functions (like `MPI_RECV`) or needs more data from the channel. The device layer can also create new (send) requests during message handling.

The implementation of OSC operations inside the CH3 device relies on specific message handlers. For `PUT` messages the, target side handler copies the data from the received message into the local window memory. The message handlers for `GET` and the accumulate functions at the target side generate new send requests that return the requested data to the origin. There, another message handler copies the received data to the buffer in the application memory.

Implementation issues

The description given above applies to MPICH and thus to RCKMPI as well since it is an MPICH derivate. However, the implementation of the MPB-based CH3 channel device in the original RCKMPI version contained flaws that prevented one-sided communication to work correctly on the SCC.

First, the message handlers were not correctly invoked. Instead, every received package was stored in the receive request queue. This prevented both the synchronization and the communication operations to work correctly (see Section 3.4.1). Second, the progress engine of the CH3 channel did not account changes in the generated requests by the CH3 device. That is, after reception of a full message, every request was considered to be completed. However, some RMA operations, such as `GET`, require multiple messages rather than a single one. As a result, the library got stuck when a subsequent message is not processed as expected and the according request was considered to be completed. Third and finally, data that has been received was not copied into a provided buffer when required by the library. Thus, even when the message handler was invoked a `PUT` operation did not replace data at the target side. [4]

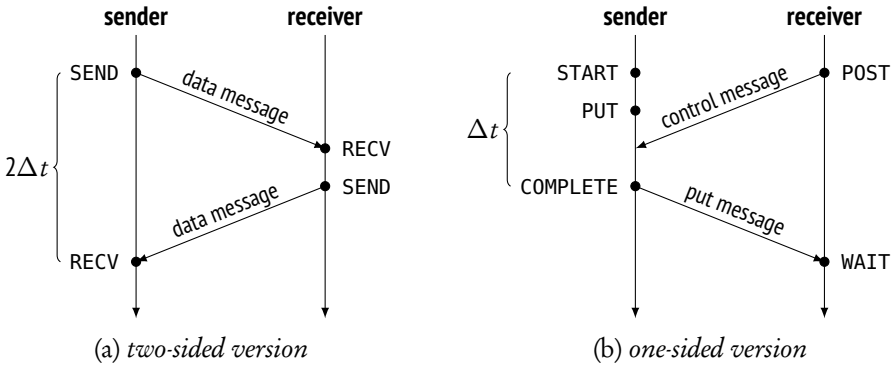


Figure 4.4: *Message flow of RCKMPI for the OSU latency benchmark.*

As sketched in Section 3.4.1, these issues have been resolved as part of the research for this thesis [4]. Required but missing code paths were added to comply with the CH3 device requirements. These fixes in the implementation were carried out after extensive debugging and source code study of the library. After the errors have been fixed, the MPICH test suite (cf. Appendix A) succeeded in mostly all test cases with regard to one-sided communication. Exceptions are those cases which test MPI-3 functions that address the unified memory model and therefore rely on cache-coherent shared memory systems. Those fail for obvious reasons on the SCC.

Performance

Besides the (resolved) implementation issues, MPICH's approach itself constitutes another issue for efficient OSC in the SCC. Since messages are used, the immanent copy overheads (see page 83 and Figure 3.10) make the communication inefficient. Compared to two-sided communication (with `MPI_SEND` or `MPI_RECV`) where the progress engine only creates receive requests for later pickup by the receive calls, additional message handling is required at the CH3 layer.

To assess the performance degradation by the message handling, the raw communication performance is analyzed in the following. The OSU micro-benchmarks version 5.0 [131] are employed for the experiments. Both latency and bandwidth are examined for two-sided and one-sided communication. For the two-sided version, the `osu_latency` and the `osu_bw` are used. The latency benchmark is a ping-pong benchmark (see Figure 4.4a) where the derived latency (Δt) is half the round-trip time of the message sent using blocking calls (`MPI_SEND` and `MPI_RECV`). Differently,

the bandwidth benchmark sends multiple messages in a non-blocking fashion and waits for a single (empty) reply from the partner. The time to complete to all transfers is used to derive the bandwidth.

The one-sided benchmarks are `osu_put_latency` and `osu_put_bw`. Both behave similar to the two-sided counter parts. To determine the latency, the total time for an access epoch containing a single put operation and including the synchronization (see Figure 4.4b) is recorded. For the bandwidth, multiple puts are issued within the access epoch and the total time for the epoch is recorded. Thus, all benchmarks include the cost for both implicit (two-sided) and explicit (one-sided) synchronization.

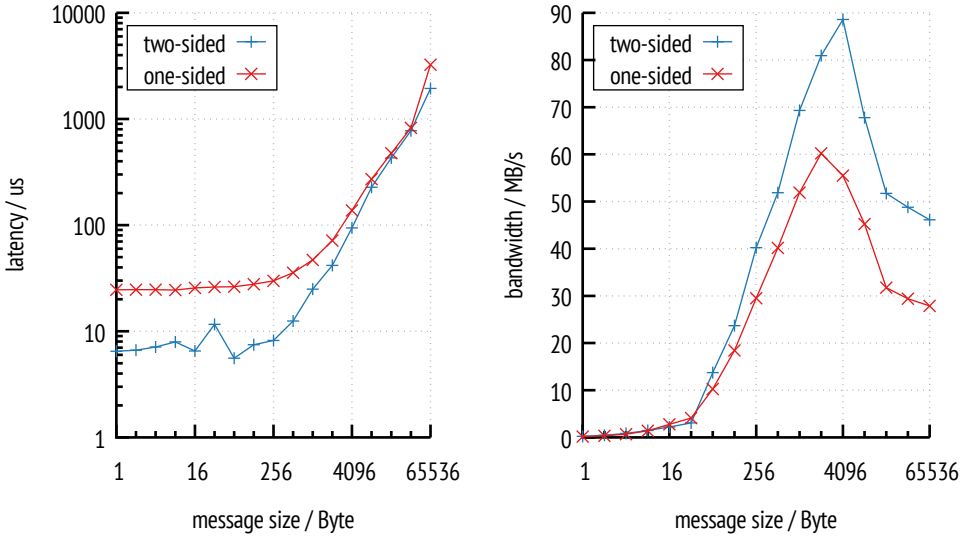
Moreover, the internal behavior of MPICH/RCKMPI concerning the message flow at CH3 level is nearly identical for the two communication schemes (see Figure 4.4). However, the one-sided version appears to be more lightweight as the control message does not contain data as it is the case for the "pong" message in the two-sided version.

In the experiments, no more than two processes are started. The processes run on core `0` and 1, respectively. For the one-sided benchmarks, general active target synchronization is employed. The remaining settings of the OSU benchmark were kept at the defaults. All other system settings are identical to those described in Section 3.5.1 on page 99. The tests were conducted until a message size of 1 MB was reached which is already far above the size of the MPB (8 KB) that serves as message transport medium.

Figure 4.5 presents the result of the latency and bandwidth benchmarks. It is evident that the performance of the one-sided communication is significantly slower in terms of bandwidth and latency than for the two-sided version. Nevertheless, the number of messages is equal between those versions. In case of the one-sided benchmark fewer data is transferred, as the initial control message does not contain data since it is the case for the acknowledge message in the two-sided benchmark (see Figure 4.4).

The reason for the latency differences can be attributed to the message processing on the receiver (target) side. This step causes a delay of the next control message (issued during `POST`) which in turn prolongs the access epoch since `COMPLETE` waits for the arrival of that control message. As a result, the latency is increased. In contrast, the actual transfer time of the `PUT`'s data is very likely to be equal to the two-sided one since the same amount of data is transferred.

The lower performance of RCKMPI's implementation of OSC also affects application performance. Figure 4.6 shows the strong scaling of the cellular automaton application



(a) *latency benchmark* (b) *bandwidth benchmark*

Figure 4.5: *Performance comparison of RCKMPI's two and one-sided communication.*

that was used for preliminary investigations in Section 3.4.5. In the application, two neighboring processes of a one-dimensional torus-shaped process topology have to exchange ghost zones for the employed stencil computation. The application uses 2400 lines, each having a size of 8 kB. The resulting problem size exceeds the capacity of the caches even when 48 processes are used. Two lines are exchanged per iteration and process. In total, 50 iterations are performed during the runtime of the application. The application was executed nine times per given number of MPI processes. The median of these nine runs is displayed in Figure 4.6. As apparent from the results, the one-sided version scales significantly worse than the two-sided version on the SCC.

For comparison, the one-sided version was also executed on an InfiniBand-based cluster with 28 nodes and using one process per node. The cluster's parameters are listed in Appendix C. The runs on the cluster, 24000 lines were used to account the larger caches of the installed Nehalem Xeon processors and avoid super-linear speedup caused by problems fitting into the cache. The source was compiled using GCC C-Compiler version 4.9.1, and Open MPI 1.8.2 was employed as InfiniBand-aware MPI implementation. The results from that platform confirm that one-sided communication can deliver close-to-linear scaling. However, it also emphasizes the drawback of the message-based implementation that RCKMPI inherits from MPICH.

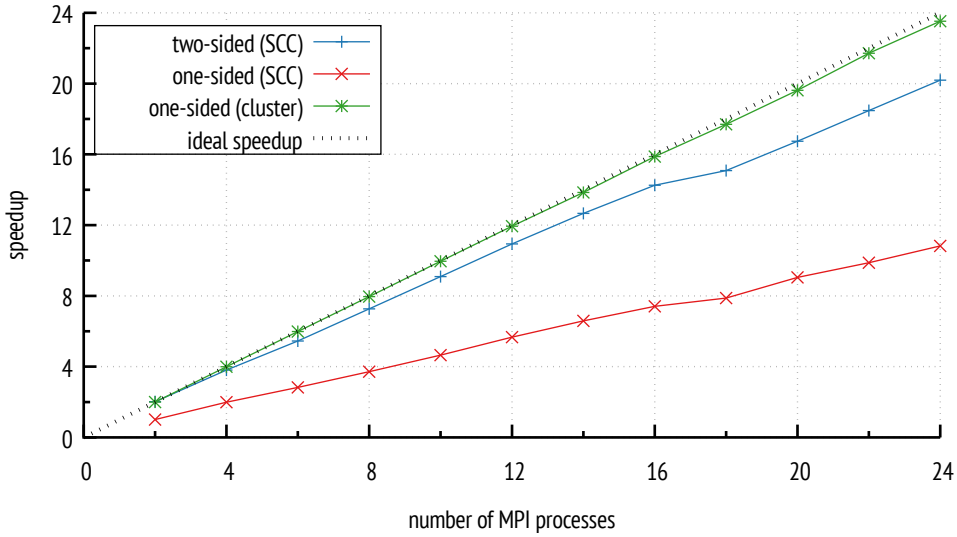


Figure 4.6: *Cellular automaton scaling using one and two-sided communication.*

4.1.3 Other MPI Implementations

In addition the original RCKMPI, library extensions to the library exists as well as other MPI implementations that address nCC systems, inluding the SCC.

RCKMPI Extensions

To improve RCKMPI's partitioning scheme (see Section 2.3.3) that distributes memory equally to all processes, providing topology information using the MPI API have been identified as a beneficial measure by CHRISTGAU ET AL. [2] and in the author's master thesis [6]. By passing the topology information, the more space inside the buffer is attributed to the communication partners thus increasing the bandwidth between them. Experimental results show a four to five times higher bandwidth between communication partners compared to the original implementation when a one-dimensional ring topology is applied.

Opposite to the explicit approach presented in [2], UREÑA AND GERNDT [145] discuss an implicit solution named RCKMPI2. The static partitioning is kept but does not depend on process count anymore. Instead, a fixed number of 48 write sections is

allocated in the MPB. However, their size is reduced 64 bytes, leaving 5 KB free for other packages. Of this free space, 4 KB are used to store larger message fragments at the sender side for pickup by remote processes. Protocol meta data is used to signal presence and position of data in the 4 KB region. Experiments with SKaMPI (see Section 3.5.3) show latency improvements by a factor of 3 to 6 for multiple MPI collective operations, especially for high core counts and message sizes. Also, a slightly improved performance of the NAS parallel benchmarks kernels can be observed.

Although these two extensions address the channel implementation and improve the message performance, which theoretically would improve the one-sided performance as well, they are still affected from the problems depicted in Section 3.4.1 and 4.1.2. That is, they do not provide a working implementation for one-sided communication. Even in that case, their message-based heritage would cause performance similar to the one observed in the previous section.

SCC-MPICH

CLAUSS ET AL. [146] developed another MPICH-derivate for the SCC, called *SCC-MPICH*. It introduces additional transfer protocols that avoid copies into buffers by the MPI library. Nevertheless, it is still a message-based implementation.

An approach to support MPI OSC by means of shared memory is described in [53]. But, REBLE ET AL. disable the caches for memory regions that belong to windows created and allocated by `MPI_WIN_ALLOCATE`, e.g. Despite the disabled caches, the authors apply the unified memory model of MPI to the SCC (cf. Section 4.1.1). This is at least questionable as the concept of private (located in the cache) and public window (located in memory) copies is merely applicable for disabled, i.e. practically non-existing, caches. As a consequence, the challenge of managing the cache coherence and synchronization between the two types of copies is not faced.

Concerning performance, the message-based approach still outperforms the shared memory approach by a factor of up to 2 for message sizes between 1 and 256 KB when using UniDir-Put of the Intel MPI Benchmarks. For all other sizes, messaging is significantly slower. For small message sizes, the authors attribute the slow performance to the signaling and list-keeping overhead required inside the MPI library. However, the maximum bandwidth is 40 MB/s which is below the performance of RCKMPI's one and two-sided communication (cf. Figure 4.5).

MPI/SX

TRÄFF ET AL. [113] describe the implementation of MPI's one-sided communication for NEC SX-5 vector supercomputer. Those machines are made out of shared-memory nodes which are connected via a crossbar switch. Each of these nodes contains up to 16 vector processors. Data transfers between the nodes can be offloaded from the CPU to the switch hardware. In addition, the hardware allows to use global shared memory that is accessible across the nodes via dedicated copy functions of the switch hardware. However, cache coherence is not maintained *"for performance reasons"* [113].

The MPI implementation for the SX-5 is called MPI/SX. It makes use of the global shared memory and the switch hardware capabilities. Thanks to both features, data transfers can be issued solely by the origin and do not need interaction with the target (besides the synchronization). To maintain the coherence of the caches, a *"cache clear operation"* is issued in the ending `FENCE` or `WAIT` calls of an exposure epoch. A write-through cache policy is used to enforce the visibility of updates in main memory [144]. However, the direct data transfers are only possible if the window memory was allocated inside the global shared memory using MPI's `MPI_ALLOC_MEM`. If the window memory resides in the private process memory, the implementation falls back to a message-based data transfer. The underlying data transfers then still relies on the same switch hardware capabilities as for the truly one-sided implementation.

Concerning the performance, the authors observe a significant advantage for windows in global shared memory. It outperforms the message-based OSC implementation by a factor of up to 2.9 when exchanging a contiguous data block between eight processes. This is notable as the message-based implementation uses the same hardware features for the data transfers.

Nevertheless, even the fastest implementation of the OSC exchange benchmark is drastically slower than a two-sided exchange using `MPI_SENDRECV`. Only for very large data exchange sizes the one-sided benchmark outperforms the two-sided. The effect is attributed to the additional effort of explicit synchronization for the one-sided version. However, the work shows that a truly one-sided implementation of the MPI API can clearly outperform a solution that requires participation of the target side.

In summary, the available MPI libraries for the SCC implement one-sided communication over messages. In presence of shared memory, this is inefficient as depicted in Section 4.1.2. As a result, an improved scheme for one-sided communication that addresses nCC systems is developed in the remainder of this chapter.

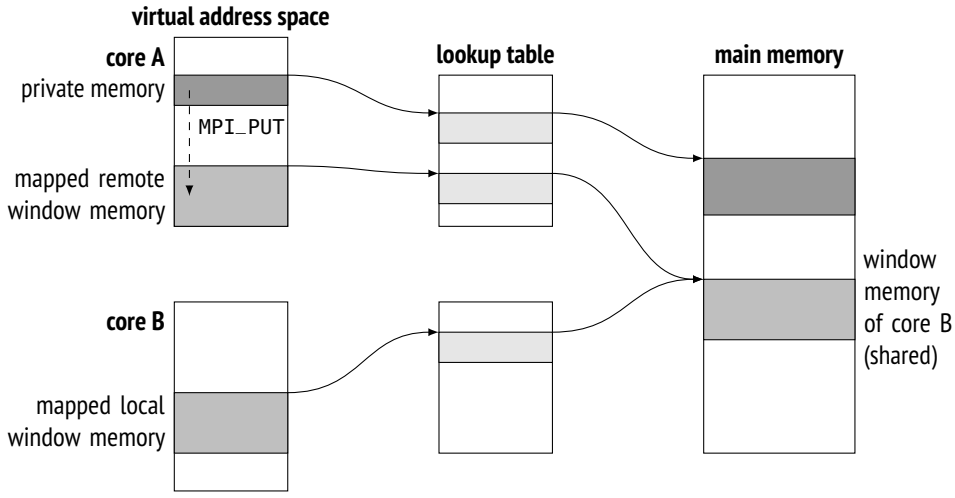


Figure 4.7: *Using LUTs to define shared memory for windows.*

4.2 SCOSCo: An Approach for the Intel SCC

From the analysis of RCKMPI's implementation and the conducted experiments, it is clear that a message-based implementation of one-sided communication is not optimal for the SCC. This is also supported by observations in other research, such as the work on MetalSVM, where it was found that message passing introduces significant overhead compared to direct memory accesses (see Section 2.4.3).

While RCKMPI supports the principle of one-sided communication, i.e. providing the communication parameters only at the origins, at an API level, the actual implementation still requires close interaction between origins and target. Within the following sections, it is shown that this concept is sub-optimal and shared memory can be used to achieve true one-sided communication even on an non-cache-coherent architecture.

To address the mentioned conceptual and performance issues, the hardware features of the SCC are exploited. Given a window is created in the main memory, its system address is known or at least can be derived by assistance of the underlying operating system. The system address of the window can be exchanged with all origin processes. As shown in Figure 4.7, free entries of the core LUTs (cf. Section 2.2.6) can be reconfigured to create shared memory between origin target processes.

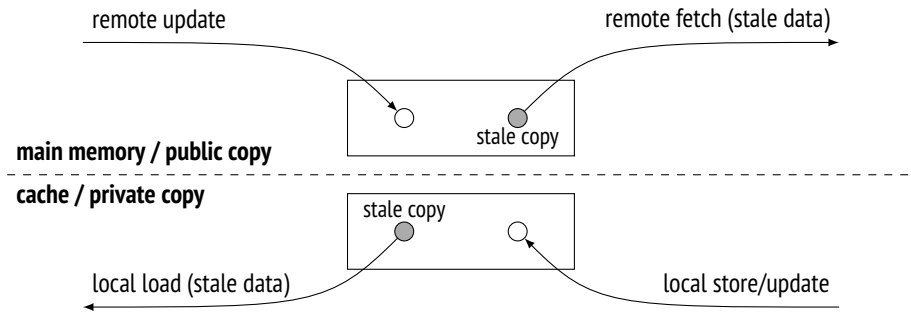


Figure 4.8: *Cacheing issues for shared-memory-based one-sided communication.*

Subsequently, the established shared memory can be used to access the window memory directly, i.e. without message creation and processing overheads. As in other MPI implementations for shared memory system, the `PUT` and `GET` operations become `memcpy` operations, while the accumulate can be implemented by accessing the remote memory directly.

4.2.1 Cache Coherence Management

As discussed in Section 4.1.1, the separate memory model of MPI applies to the SCC. This is because changes that have been applied to the public window copy (residing in the main memory) are not automatically propagated to the private copy which might be stored in a core's cache. The same applies to the other direction: local modifications are not automatically committed into RAM but only affect the local cache if the modification is applied on cached data. These situations are illustrated in Figure 4.8.

In summary, caches can hold copies of window data. If multiple accesses to the same window (memory) location are performed and at least one of them is a write (i.e. local store or `PUT` operations) the changes should be observed by the other (subsequent) read operations. This is known as cache coherence problem. [27, p. 10]

The obvious reason for this issue on the SCC is the lack of hardware-based cache coherence. As a result, the cache coherence needs to be managed in software. SCOSCo is the name under which this concept and its implementation with a focus on one-sided communication is designed in the remainder of this thesis.

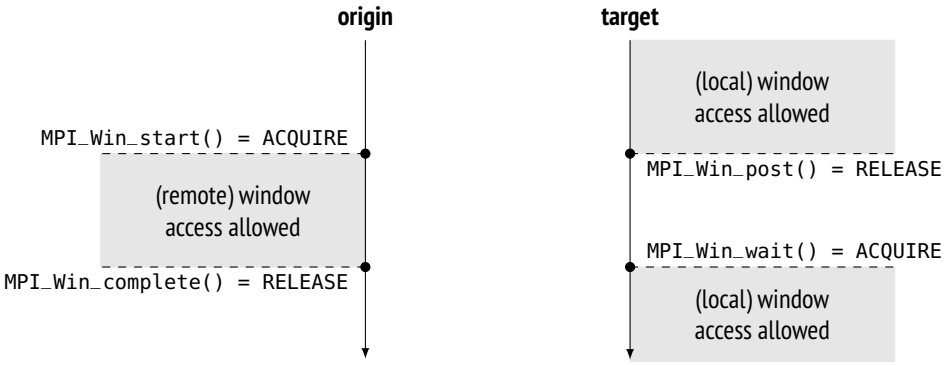


Figure 4.9: *Matching of release consistency operations to MPI's GATS calls.*

4.2.2 Memory Model

The management of cache coherence is essential to get a working shared-memory based implementation of MPI's one-sided communication. However, a memory model is required which describes when changes have to be observed by processes. As pointed out in Section 4.1.1, the MPI standard defines the according semantics for different hardware architectures. The separate model has been identified as the appropriate one that matches to the SCC.

Nevertheless, as outline in Section 2.1.1 the literature defines different memory models, i.e. consistency models which *"define the allowed behavior of [...] programs executing with shared memory"* [27, p. 21].

Of the memory consistency models, the RC model is a good match to MPI's separate memory model for one-sided communication. This model requires an *acquire* operation to precede memory accesses to a shared location. On the other hand, a *release* operation ensures that all previously issued operations have been performed, i.e. their effect became visible to other processes. [30]

The release and acquire operation resemble the methods of general active target synchronization from MPI (cf. Section 3.1.3) to some extent when the whole window is considered as a single shared memory location. Figure 4.9 illustrates the matches between the release consistency model and MPI RMA model.

With `MPI_WIN_POST`, an origin process ensures the private and the public copy a window are synchronized. That is, any change by a store operation is made visible [65, p. 453,

rule 5]. In addition, it serves as kind of a local memory barrier that ensures completion of all previously issued memory reads from the window. Thus, a `MPI_WIN_POST` matches a release. Vice versa, a call to `MPI_WIN_START` has to precede all operations on a remote window (shared memory in sense of RC) which makes it an acquire operation.

Similar, `MPI_WIN_COMPLETE` ensures completion of all window accesses, such as `PUT`s [65, p. 453, rule 3]. Thus, it is a release operation. Because a `MPI_WIN_WAIT` call has to be performed before any subsequent local access in order to make the effects of RMA accesses visible [65, p. 453, rule 6], it has to be considered as an acquire operation. The MPI separate model supports this argumentation as it forbids local accesses to memory while an exposure epoch is active [65, p. 455].

Different to the RC semantics, the acquire and release operations in context of MPI's one-sided communication have to be distinguished depending on which process they are executed. The release (`MPI_WIN_POST`) and acquire (`MPI_WIN_WAIT`) methods on the target side must be considered as exclusive. That is, only the calling process (the target) can access the memory after completion of the acquire method. This matches the RC model.

However, for the targets, an acquire does not imply exclusive access. Because access to the window is possible for all origins that issues an acquire operation (`MPI_WIN_START`), it has to be considered as an shared access.

4.2.3 Requirements for MPI One-Sided Communication

To realize the RC-like separate memory model, the coherence of the private and public window copy has to be managed by the SCOSCo approach. This involves the following tasks of both the origin and the target side.

1. Target processes have to ensure the following:
    a) Potentially cached data of a local window is written back to main memory when an exposure epoch starts. Consequently, remote operations like `GET` retrieve the most current data from the public copy (i.e. memory).
    b) Contrary, potentially cached but stale data of a local window needs to be invalidated when an exposure epoch is closed. This enables the target process to load current data from the local window that was potentially modified by remote `PUT` operations.

2. Origin processes have to perform the following steps:
   a) At the beginning of an access epoch, an origin process needs to invalidate all cache lines originated from remote windows that have been accessed in previous access epochs. This is required as such data might be modified meanwhile by other origins or the target itself. In this case, GET operations would return stale cache lines.
   b) In addition, all outstanding (cached or buffered) modifications of remote memory have to be flushed to become visible in the public window copy when the origin closes its access epoch.

In essence, these tasks require to invalidate cache portions and to flush cached data with the restriction to a specific window. This approach prevents invalid cache states based on software actions. Thus SCOSCo is a software-based *self-invalidate* method which does not rely on invalidation actions by a hardware cache coherence protocol.

4.2.4 Memory Type Considerations

When a window is stored in shared memory, the different memory types of the SCC (cf. Section 2.2.4) can be used to map it into the address space of an MPI process. Depending on the memory type, the costs in terms of runtime overhead for the required invalidation and flush operations (see above) have to be considered. In the optimal case, they should not impose additional costs that outweigh the advantage of the proposed `memcpy`-based communication operation over the message-based approach. Thus, the four available types and their implications on the coherence management and performance are discussed in detail.

DCM (definitely cacheable memory) The privileged x86 instruction `WBINVD` can be used to flush and invalidate the cache. While the Pentium's instruction set specifies that this opcode flushes and invalidates both internal and external caches, the actual implementation on SCC only affects the L1 cache content. The same applies to `INVD` which only invalidates the L1 cache but does not affect the L2.

To invalidate and flush the L1 content with `WBINVD`, the instruction requires more than 10,000 clock cycles (equaling about 18 µs at a clock rate of 533 MHz) in addition to the transition into kernel space which takes around another 2,000 clock cycles [141]. Since the complete L1 cache and not only the window's data is invalid afterwards, the instruction leaves the application with a completely cold L1 cache. The same drawback applies to `INVD`.

To work around the issue of a missing hardware-based L2 flush, the modified kernel offers an address range-based method. It reads garbage data to enforce write-backs. This will take even more cycles than the L1 flush. Even worse, the offered method invalidates the cache without respect to its content. Given a window that is larger than the L2 cache, all data will be flushed and invalidated, not just the window data which might constitute only a small fraction of the cache. Also, no pure per-line invalidation is available, only a write-back.

Due to the large overhead of cache invalidation and flushes as well as the lack of pure invalidation that drops data from the cache without writing it back, this memory type is not considered to be managed by software cache coherence.

NCM (non-cacheable memory type) Obviously, the use of non-cacheable memory relieves an implementation of the task of maintaining the cache coherence, since no caches will be involved when all window memory is mapped with this type. Due to its poor performance [53], it is not considered as a solution here.

MPBT-WB (Message Passing Buffer type – write-back) When memory is mapped with the MPBT memory type (independent of the cache write policy) the cached lines are marked with a bit indicating the memory type (Section 2.2.4). The `CL1INVMB` instruction can be used to invalidate only those tagged lines. This partially solves the problem of DCM's complete cache invalidation although it does not provide an address based operation. Thus, using the `CL1INVMB` instruction, the required cache invalidation can be achieved.

To propagate write operations to memory, the WCB that comes with MPBT can be flushed explicitly (cf. Section 2.2.4). Together with the `CL1INVMB` instruction, the WCB flush fulfills the requirements for origins (flush of outstanding writes and invalidation of window memory) and partially for target processes (invalidation).

However, a cache flush dedicated only to MPBT lines is not available. This is the missing requirement for targets in order to ensure that stores are pushed into memory. Alternatives, like `WBINVD` or a manual flush, which also imply invalidation, are costly anyway (see DCM above) even if the MPBT is only L1-cacheable. Despite the fast invalidation, the missing fast cache flush for targets makes the MPBT-WB a less favorable choice for a window's memory type.

MPBT-WT (Message Passing Buffer type – write-through) This type would circumvent the need for cache flushes as the write-through semantic ensures updates of the

public copy and thus solves the missing issue of MPBT-WB. The other beneficial features of MPBT memory (explicit and fast invalidation, flush of WCB) still apply.

The additional latency penalties that come with write-through semantics might be hidden by the WCB. Anyhow, as with MPBT-WB, the drawback of MPBT-WT is the restriction to the L1 cache.

From the discussion, MPBT-WT appears to a valid choice for a memory type for the memory that is exposed through a MPI window. This is supported by observations of other researchers.

ROTTA ET AL. [100] (see Section 2.4.4) report the costs for L2 cache write-back, which implies usage of DCM memory, to be 7500 clock cycles per cache line (while running the cores at 800 MHz). The cost for flushing the whole L2 cache takes about 600,000 cycles which equals around 700 µs. If a completly modified L2 cache is written back the time doubles according to the data provided in the publication, which underlines the high performance penalty for L2 cache line flushes.

In their analysis, the authors also observe congestion on the memory controllers when more than three processes perform concurrent read accesses. This questions the usage of write-through memory, but it clearly contradicts the results from VAN TOL ET AL. [141] where using more than 12 cores, i.e. more than the cores assigned to a memory controller, exhibit congestion. However, the measurement setups were different, as VAN TOL ET AL. use copy operations and ROTTA ET AL. discuss read operation latency.

For their examined graph applications with software-based coherence, ROTTA ET AL. [100] observe highest runtimes and therefore a bad scaling behavior for the when DCM is used for memory. Even using the NCM provides better performance. Further improvements are achieved by using the MPBT-WT variant. This is attributed to the L1 cache which is activated by that memory type. The effect of the WCB on memory accesses is not mentioned in the publication. Nevertheless, the authors present another application where the cache flushing overhead of DCM is compensated by more computational demanding tasks than in the first application. Overall, the difference to the MPBT-WT version is small and their investigation of different memory types confirms that MPBT-WT memory can facilitate software-managed cache coherence as it is flushable and modifications become visible in main memory after an explicit flush.

Another notable argumentation for using the write-through memory according to CHEONG AND VEIDENBAUM [107] type is to avoid bursts on the interconnect when a cache flush is issued. Since this operation occurs at synchronization points (e.g. after a access epoch and before and exposure epoch), multiple processors would send large amounts over the network, which is time-intensive (see discussion for DCM in Section 4.2.4) and might cause network congestion. The MPBT-WT in combination with the write combine buffer, as they are used by SCOSCo, avoids these issues as well. Note that write-through configurations were used in other systems as well, such as the NEC SX [144] and the IBM RP3 [147] where the programmer is responsible for managing the coherence [148].

In consequence of the above analysis, MPBT-WT is most appropriate for SCOSCo. It enables to manage the cache coherence with only little software overhead for the coherence management, i.e. careful invalidation of cached data and flush of the WCB. This is supported as only MPBT cache lines are affected by an invalidation performed with `CL1INVMB` and does not cause the whole cache to be wiped.

4.2.5 Implementation Sketch

When the chosen memory MPBT-WT type is used for MPI windows, the cache coherence management can be implemented within the MPI synchronization calls. Thereby, MPI's separate window model is realized. The following discussion concentrates on the GATS scheme and describes the additional steps that need to be performed to fulfill the requirements from Section 4.2.3.

`MPI_WIN_POST` For targets, the WCB is flushed before the synchronization with origins starts to update the public window copy as it could buffer writes to the window memory that is accessed within the following access epoch (requirement 1a).

`MPI_WIN_START` The origin invalidates potentially cached window memory with the `CL1INVMB` instruction which fulfills requirement 2a. To ensure that no modifications to local memory are lost due to invalidation, the WCB is flushed beforehand.

`MPI_WIN_COMPLETE` As for the targets' `POST` operation, the WCB is flushed when an access epoch ends to ensure completion of outstanding write operations to remote windows caused by `PUT`s (requirement 2b).

MPI_WIN_WAIT The target invalidates the cache with `CL1INVMB` to read the most current data from the (potentially modified) public window copy (requirement 1b).

As illustrated, the MPBT-WT memory type allows to fulfill the requirements for cache coherence and thus the RC-like separate memory model of MPI with little overhead in terms of an implementation of SCOSCo.

4.3 Implementation

Based on the presented sketch, the implementation of the SCOSCo is discussed in the following. The implementation of the cache coherence and communication operations was done along with the optimized synchronization scheme presented in Chapter 3. Thus, MPICH 3.1.3 with the MPB-based channel from RCKMPI (see Section 3.4.1) was used as foundation for the following changes.

The required methods were added by providing the `MPIDI_CH3_Win_fns_init` function, which initializes a function pointer table. The according entries were modified to override the default CH3 routines and to point to the new implementation

4.3.1 Window Creation

As outlined in Section 2.3.5, the collective creation of a window object is a prerequisite for one-sided communication in MPI. The window creation is typically connected with allocation of according memory that will be exposed with a window. For the SCOSCo approach, other processes need to access this memory directly with the help of the LUT re-configuration. For this, the system address of the locally allocated memory is required to be known because parts of the address constitute the LUT entry information (cf. Section 2.2.2).

To compute the system address of a given virtual address, the physical address of that memory region needs to be obtained. Using the physical address, the used LUT entry can be determined. Thereby its content is known which is the system address of the matching system memory page (see Figure 2.3). The physical address of any virtual address is obtainable with the help of the SCC-specific kernel driver providing the `/dev/dcm` device.

Memory Allocation

The device driver enables to determine the physical address of an arbitrary virtual memory address and, thus, allows making every memory be part of a window that is exposed to other processes. However, the MPBT-WT memory type has been identified to be used for window memory in order to fulfill the requirements for one-sided communication (see Section 4.2.3). The memory type can be changed on a page basis only. That is, other data residing on a page will be affected as well. This exposes risks to the proper execution of an application. If data residing on the stack is used for a window, a change in memory type can corrupt local variables and function return addresses.

To avoid these issues, the prototypical implementation of SCOSCo requires allocating memory for a window via `MPI_ALLOC_MEM`. This function allows reserving memory that is suited for OSC [65, § 8.2]. Alternatively, the `MPI_WIN_ALLOCATE` routine can be used which allocates a memory and creates a window for it in one step.

However, a custom implementation of memory allocation inside MPICH's CH3 channel was — different to many other OSC-related functions — not possible. To overcome this lack of functionality, a patch was provided to the MPICH developers. It was accepted and integrated into the main source code tree of the library[1] enabling future implementations to adjust a CH3 channel to allocate RMA-friendly memory.

Inside the implementation, the POPSHM memory area of the SCC (cf. Section 2.2.6) is used to hold the window memory. It is used for window data only. At process startup, the local core's POPSHM memory is mapped into the address space of the MPI process. The memory type used for the mapping can be configured via an environment variable[2]. While the MPBT-WT memory is the first choice, all other previously discussed memory types (DCM, NCM and MPBT-WB) are possible as well.

Similar to the synchronization data inside the window database, a simple allocator is used that increases an offset upon allocation. The offset holds the distance from the start of the POPSHM memory to the first unused byte. A more sophisticated allocation scheme is omitted in the prototype. As a result, the `MPI_FREEMEM` operation does not free memory.

[1] http://lists.mpich.org/pipermail/devel/2015-March/000528.html last accessed 2016-09-16
[2] `RCKMPI_LOCAL_MAP_MODE`

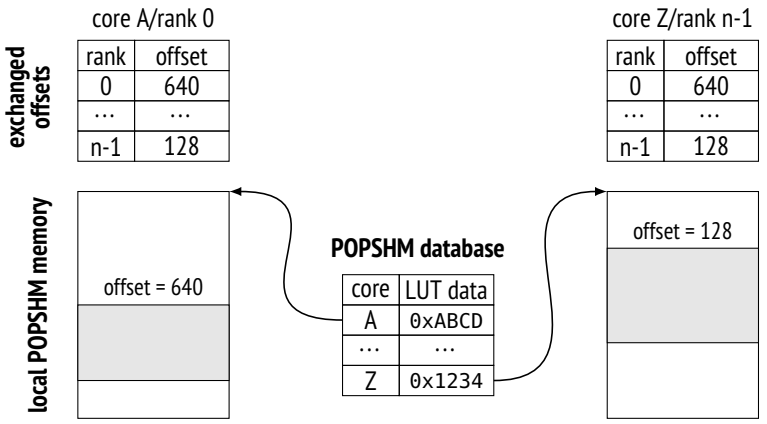


Figure 4.10: *POPSHM data and exchanged offset enable access to remote windows.*

Address exchange

Using the POPSHM approach also facilitates the system address determination of a remote window memory. The POPSHM database (see Section 2.2.6) holds the LUT information for any remote core's POPSHM memory region. In combination with the offset from the memory allocation, the base address of a remote window can be derived.

In order to access a remote window, the memory allocation offset of a remote window inside the owner's POPSHM memory must be known. This is achieved by performing an `MPI_ALLGATHER` operation during the collective window creation, e.g. of `MPI_WIN_CREATE`. The all-gather operation fills a data structure with information about all remote windows (see Figure 4.10). Besides the allocation offset, it also includes sizes of all windows.

The above mechanism causes data duplication. Every process hosts the window information data structure that contains the same information on every process. Using the POPSHM or legacy shared memory it would be possible to reduce the memory footprint and make the data structure more scalable when considering higher core counts [118]. Albeit this might be a critical issue in future systems, this optimization and a discussion of its potential trade-offs (de-duplication vs. higher access latencies) is left out for the prototypical implementation of SCOSCo.

4.3.2 Communication Operations

To access the window of a remote process, the exchanged offsets and the system addresses from the POPSHM database (see above) are used.

Before an RMA operation is to be performed, the unoccupied LUT entries 132 to 190 (see Table 2.1 on page 28) of the local core are reconfigured with the address information of the target memory. Thereby, shared memory is created. To access this memory, again the special kernel devices are utilized to map the matching physical addresses into the virtual address space of the origin MPI process (see Figure 4.7). After that, remote memory can be accessed in the same way as local memory. Similar to the local window memory, the type used for mapping remote memory can be configured via an environment variable[3].

When the mapping is established, the complete POPSHM memory of the remote core is mapped in to the address space of the accessing MPI process. By default every core reserves four 16 MB pages of system memory for POPSHM. Those need four LUT entries to be made accessible. Thus, the available (free) 58 LUT entries can hold up to 14 mappings to remote POPSHM memory regions. If more mappings need to be established, existing mappings are removed from the LUT. A least-used replacement strategy is used to select the mapping for eviction. Usage is counted by the number of RMA operations (`PUT` and `GET`).

With an established mapping, `PUT` and `GET` operations are basically implemented as calls to `memcpy`. For the prototypical implementation, all remaining operations, like `ACCUMULATE` or `FETCH_AND_OP` are left unimplemented. However, their realization appears to be feasible, e.g. by re-using the MPICH's default implementation for shared memory systems for applying the MPI operations for different datatypes. The required atomicity of `COMPARE_AND_SWAP` could be achieved with the help of locks.

4.3.3 Management of the Cache Coherence

As outlined in Section 4.2.5, the synchronization is performed inside the synchronization calls. It was implemented for general active target synchronization and fence synchronization.

[3] `RCKMPI_REMOTE_MAP_MODE`

```
MPIDI_CH3I_win_start(...)
{
    /* init local data for sync. */

    /* coherence management */
    MPIDI_CH3I_CL1INVMB;
}

MPIDI_CH3I_win_complete(...)
{
    /* coherence management */
    MPIDI_CH3I_Flush_WCB();

    /* synchronize with targets */
}
```

```
MPIDI_CH3I_win_post(...)
{
    /* coherence management */
    MPIDI_CH3I_Flush_WCB();

    /* synchronize with origins */
}

MPIDI_CH3I_win_wait(...)
{
    /* synchronize with origins */

    /* coherence management */
    MPIDI_CH3I_CL1INVMB;
}
```

Listing 4.1: *Coherence management implementation in the GATS routines.*

For GATS, the methods that open epochs (`MPI_WIN_POST` and `MPI_WIN_START`) perform the coherence operations before the actual synchronization takes place. This ensures an up-to-date state of the public copy when processes get synchronized. The reverse applies to the calls closing an epoch (`MPI_WIN_COMPLETE` and `MPI_WIN_WAIT`). In those cases, the private copy (i.e. the caches) is updated when the processes have synchronized. The pseudo-code in Listing 4.1 shows the explained steps for the GATS routines.

In order to invalidate the window data, the `CL1INVMB` instruction is issued in `MPI_WIN_START`, `MPI_WIN_WAIT`, and `MPI_WIN_FENCE`. To flush the write combine buffer during `MPI_WIN_POST`, `MPI_WIN_COMPLETE`, and `MPI_WIN_FENCE`, a dummy variable that is allocated in the legacy shared memory is written. The according memory region of the LSM is mapped as MPBT-WB which ensures, that the WCB is affected by the write. No other data lies on the cache line of the dummy variable. Thus, a write to that variable flushes the WCB.

4.4 Experimental Evaluation

The presented implementation of the software-managed cache coherence and the shared-memory based one-sided communication is evaluated within the next subsections. The experiments are conducted in the same environment described in Section 3.5.1. For all performance measurements, the extended RCKMPI library was compiled with optimization (level 2). The same applies to the benchmarks.

4.4.1 Functional Tests

To verify that the implementation fulfills the functionality specified in the MPI standard, test cases from the MPICH test suite are reused. As this thesis focuses on one-sided communication, only tests from the `rma` sub-tree were used to check for bugs in the implementation of both the synchronization and the data transfer.

The sub-tree contains 122 test-cases addressing one-sided communication. Since only the fence and GATS synchronization styles have been implemented with support to cache coherence, only those were considered for inclusion in the final test set. Since the SCOSCo implementation requires window data to be allocated with `MPI_ALLOC_MEM` or `MPI_WIN_ALLOCATE` (see Section 4.3.1), the final number of tests was reduced to eight tests with communication.

The set includes the seven tests from the evaluation of the synchronization calls (see Section 3.5.2) with the exception of `nullpscw` which does not use communication. In addition, the `putfidx` and the `test5_am` test case were used. Further, `allocmem`, `attrorderwin`, `badrma`, `wincall`, `winname`, and `window_creation` were employed using two processes (as defined in MPICH's test suite) to ensure that the supporting functions of MPI still work after the modifications in the CH3 channel implementation.

The tests were executed using the MPBT-WT memory type to map both the local and remote window memory. The result was that all tests passed with no errors. This also includes checks for the validity of the transferred data. Thus, the tests show that the implementation of SCOSCo provides a working MPI library.

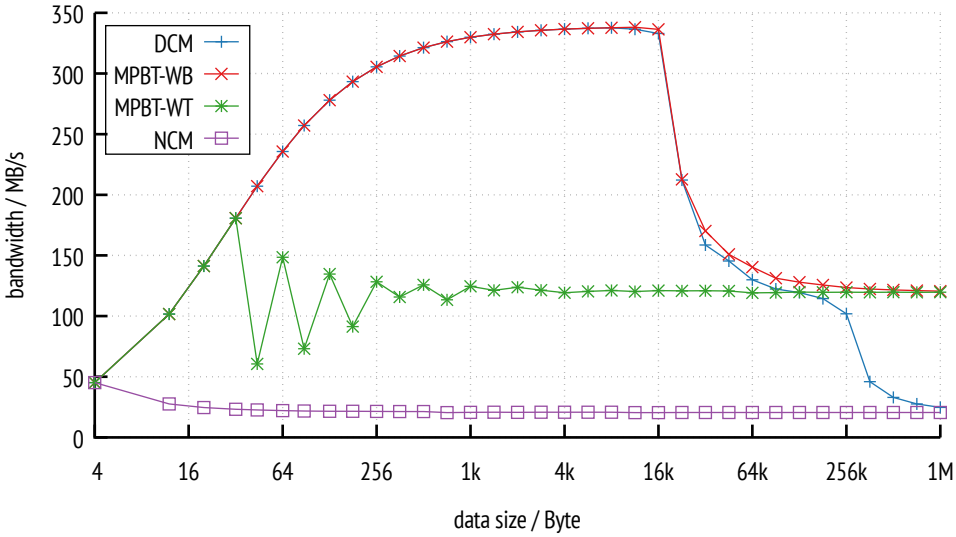


Figure 4.11: *Store bandwidth of the SCC's memory types with a warm cache.*

4.4.2 Memory Performance

The SCOSCo implementation requires an application to allocate window memory via the MPI library. This is required to ensure that memory is mapped with the MPBT-WT memory type which in turn is a prerequisite for the software-based cache coherence management. Even though MPBT-WT is considered to be well-suited in terms of software managed cache coherence, its performance especially with regard to the write performance has to be evaluated. This will support a better understanding of application performance.

To evaluate the performance, a simple load and store benchmark is used since loads and stores are the foundation of the OSC implementation. The benchmark is based on the work of by VAN TOL ET AL. [141]. If compared to the usual DCM memory type, the possible performance penalty of MPBT-WT due to its write-through semantic and the limitation to the L1 cache are revealed. The measurements from [141] compared the DCM, NCM and MPBT-WB memory types, but lack an investigation of the MPBT-WT, especially its read performance.

The benchmark loads/stores 4 byte from/into a 32-bit general purpose register at a time with the `MOV` machine instruction and measures the time to do so for a given amount of memory. The time is obtained from `RDTSC` readings. This step is repeated

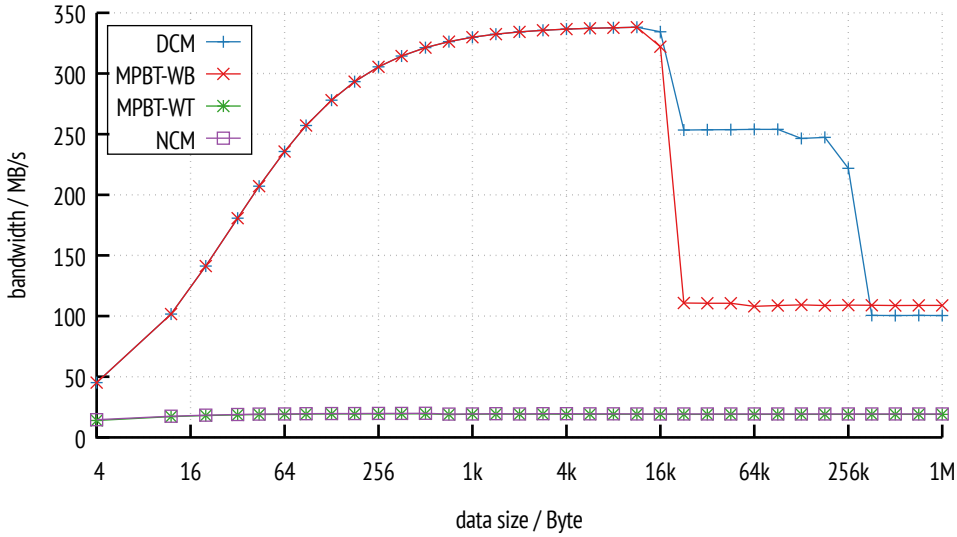


Figure 4.12: *Load bandwidth of the SCC's memory types with a warm cache.*

501 times and the median of the measured times is reported. Before each repetition, data is prefetched into the cache by reading the memory in the same direction as in the subsequent measurement loop. Appendix B.1 provides an extract of the essential source code. The benchmark was executed on core 0.

Figure 4.11 shows the obtained bandwidth of store operations. The results clearly show the effects of the warm caches. The bandwidth drops when data exceeds cache level size. Because the SCC caches are write-around, they do not cause eviction of present lines when a write miss occurs. Thus, write hits are observed for the fraction of data that is still in the cache after warm up. Therefore, the bandwidth gradually declines rather than dropping immediately when the cache is exceeded. The observed bandwidths qualitatively validate the results from [141] for DCM, NCM as well as MPBT-WB.

Concerning the memory types, the non-cacheable characteristics of NCM is apparent by a low store bandwidth. On the other hand, DCM and MPBT-WB benefit from the cache warm-up and subsequent cache hits which cause high bandwidths. MPBT-WT exposes the effects of the write-through cache configuration and the write combine buffer. The write bandwidth is lower than for cached memory. This can clearly be attributed to the write-through configuration. Nevertheless, without the WCB every write to memory would go through the on-chip network and low bandwidth would be

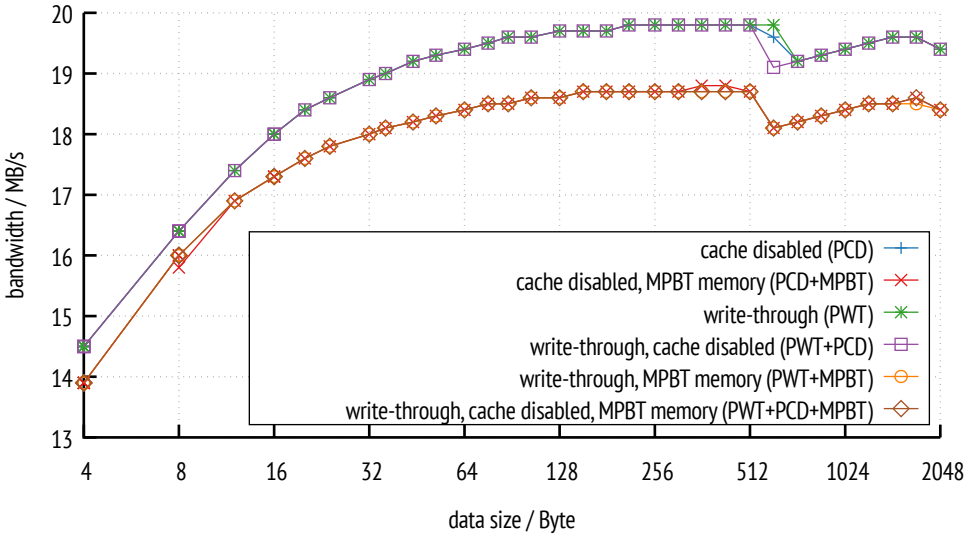


Figure 4.13: *Load bandwidth for different page table settings.*

observed as it is with NCM. However, the behavior of the benchmark is synthetic. By writing to subsequent addresses, the WCB is fully exploited and constitutes a best-case scenario. Depending on its actual access pattern, a real application would observe effective bandwidths between NCM and MPBT-WT.

In Figure 4.12, the obtained load bandwidths are shown. Sharp bandwidth reductions are observed when the cache level size is exceeded. This is the result of using the same direction during cache warm up and the measurement (see above). For DCM, the impact of the first and second level caches is clearly visible. As MPBT-WB memory is only cached in L1 the bandwidth drops at 16 kB.

The most notable result, however, is the load bandwidth of MPBT-WT. Since the MPBT memory type is L1-cacheable, an identical bandwidth as for the MPBT-WB should be observable. However, the bandwidth matches the one of the NCM memory, which is not cached. In order to verify this effect, different page table settings were used and the according memory load bandwidth was measured using the same methodology as above.

The results are illustrated in Figure 4.13. The data reveals that whenever the L1 cache behavior was configured as write-through, the read performance dropped to the NCM level.

A consulted Intel engineer, Werner Haas of the SCC team, confirmed this conclusion and suspected this might be due to the chip's design since the design of the L2 cache is not compatible with the original specification. Final confirmation was not possible as the servers with the design files had been finally shut down after Intel decided to stop research on the SCC.[4]

As evident from the results above, the MPBT-WT memory type would cause low application performance, if a local window memory is mapped with this type and the application operates much on that local window data. However, this is due to the (potential) bug in the SCC's hardware but not due to the concept of SCOSCo. To show that the concept is working and produces correct results, the MPBT-WT memory type is used in the further experimental evaluation as the default setting. As explained, this might be at the cost of absolute application performance.

The performance results from above seem to contradict the results in previous work, especially ROTTA ET AL. [100]. Therein, the MPBT-WT memory type is used and a better performance for this memory type is observed which is attributed to an activated L1 cache (see page 48 ff.). However, the publication only provides an according statement rather than experimental evidence if the observed performance gain is in fact due to the L1 cache. It is also possible that the performance is improved due to the WCB. This argumentation is supported by the relatively small difference between NCM and MPBT-WT memory observed in [100]

Similar, SIVARAMAKRISHNAN ET AL. [98] observed better performance when the MPBT-WT memory type is employed (see page 43 ff.). Similar to ROTTA ET AL. [100], neither a comparison for different memory types nor a raw performance comparison is conducted. Of the observed memory accesses, only 10% actually hit MPBT-WT memory. A ratio between read and write accesses is not stated in the article. Thus, the performance gain, which is around 40% for object mutation, can be attributed to the WCB as well.

LAM ET AL. [102] (see page 45 ff.) also used the MPBT-WT memory configuration for data sharing on the SCC. Compared to the introduced virtual shared memory mode, the MPBT-WT memory exhibits lower performance. This is attributed to the smaller cache size that is available to that memory type.

[4] *" Tja Steffen, deine Daten sind wirklich ueberzeugend [...] Ich wuerde fast den gleichen Schluss ziehen.[...] Ich koennte mir vorstellen, dass man WT 'kaputt' gemacht hat weil der L2 ja nicht der Spezifikation der Intel Architektur entspricht (siehe WBINVD). Leider wurden inzwischen die Server in Braunschweig abgeschaltet und so kann ich nicht mehr einfach im Design nachsehen."* Werner Haas, personal e-mail correspondance, 2014-04-30.

However, the authors also point out that the performance counters indicate *"many times higher"* bus utilization and stall cycle counts than in their developed SVM mode. Those effects can also be attributed to the bandwidth results presented above.

Subsuming the above, previously presented results can be explained with the lack of cachability and positive effects of the WCB rather than the limited size of the L1 cache. A validation of this hypothesis by repeating the other researchers' experiments was out of the scope of this thesis. At least, the previous results do not contradict the observed low load bandwidth of the MPBT-WT memory type.

4.4.3 OSU Micro-Benchmarks

To assess the raw performance of the proposed approach, the one-sided PUT bandwidth benchmark of the OSU benchmark suite is employed. MPBT-WT memory type is used to map the remote window into the local address space. It is also used to map the local window memory into the processes memory. To fulfill the requirement of SCOSCo, the benchmark was configured to use the `MPI_WIN_ALLOCATE` function for memory allocation and window creation (see Section 4.3.1). Further, GATS synchronization is used which includes the software-managed cache coherence (see above). The two processes were mapped to cores `0` and 1.

The benchmark itself uses a pre-allocated buffer (in conventional cacheable memory) to read data from. The data is subsequently transferred from the origin via `MPI_PUT`s into the (remote) window of the single target process. Local windows are not accessed by the benchmark in its original version. However, to ensure correct data transfer, the benchmark was extended to validate the received data against expected one after the exposure epoch has ended. The transferred data was adjusted to be unique in every `PUT` operation.

The bandwidth for the SCOSCo implementation of OSC is compared with the message-based OSC implementation and its two-sided counter-part from RCKMPI (using the two-sided PUT benchmark). The results for the latter two were already presented in Section 4.1.2. For each employed implementation, 25 warm-ups and 50 measurement iterations were performed by the OSU benchmark. Nine runs of the OSU benchmark were executed and the median of the reported bandwidth per message size is taken. The range between minimum at maximum was never higher than 10% of the minimum bandwidth.

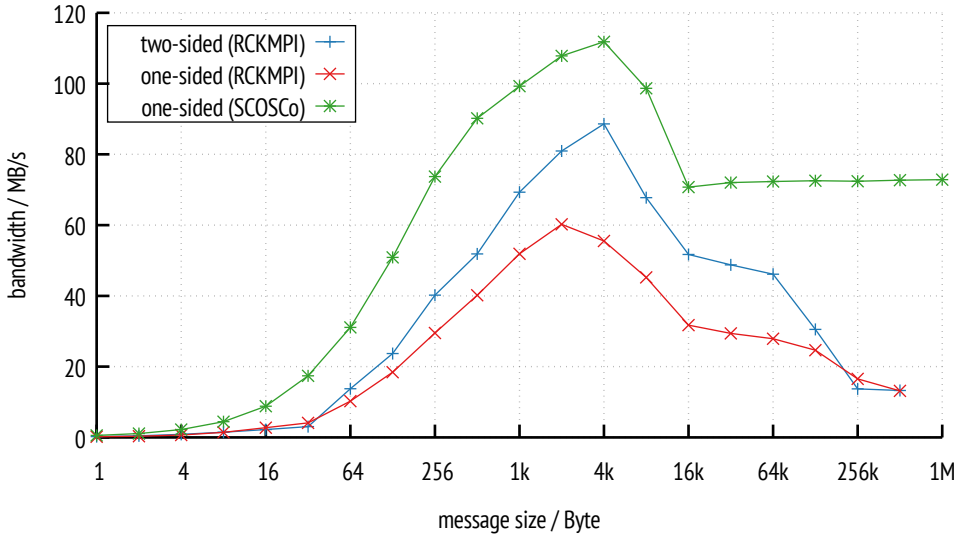


Figure 4.14: *Results for the OSU one-sided PUT bandwidth benchmark.*

In Figure 4.14, the results of the put bandwidth benchmark are shown. In all experiments, the correct data was observed by the target, especially in case for the SCOSCo implementation.

The CH3 channel that realizes SCOSCo clearly outperforms the message-based channel from RCKMPI. For small messages, an improvement by a factor of two is achieved, while larger messages are transferred at five times higher bandwidths. The data also reveals that the bandwidth exceeds the bandwidth of the two-sided communication that uses the fast on-chip Message Passing Buffer. Here, the benefit is lower, but still significant. That is, the SCOSCo variant is 20 MB/s faster using a two-sided message-based data transfer. Note that the results also show a clear improvement compared to the maximum bandwidth of 40 MB/s reported in [53] for message sizes between 64 Byte and 1 MB.

The results underline the benefit of true one-sided data transfers via shared-memory. These avoid processing overhead at the target side and message fragmentation from which the RCKMPI channel implementation suffers.

Furthermore, the results reveal that the write-combine buffer, which is activated when an MPBT memory type is used, contributes to a high and steady bandwidth. RCKMPI's CH3 device employs the WCB as well, but suffers from MPB-to-memory transfers and message fragmentation which attenuate the effect of the WCB.

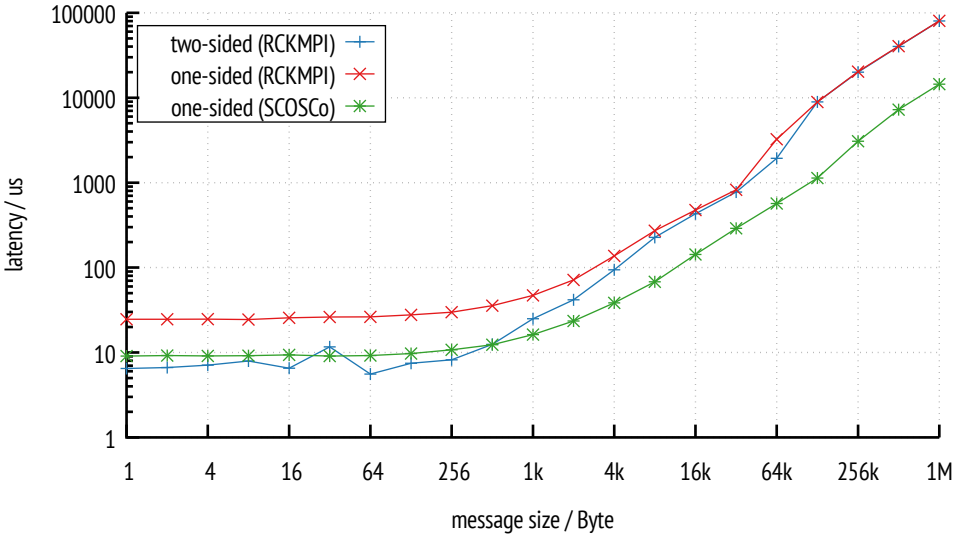


Figure 4.15: *Results for the OSU one-sided PUT latency benchmark.*

Figure 4.15 shows the results from the OSU latency benchmark. The same system and benchmark settings as in the bandwidth analysis were used. Again, the results are compared with the ones obtained in the preliminary performance studies of RCKMPI from Section 4.1.2.

As for the bandwidth, the performance of SCOSCo in terms of latency is significantly better than for the one-sided latency of RCKMPI. For the smallest displayed message size, i.e. 1 Byte, the SCOSCo latency is smaller by a factor of 2.7 (9.1 µs vs. 24.8 µs). The gap increases with increasing message size. For messages of 4 kB, SCOSCo is 3.7 times faster than one-sided RCKMPI.

Opposite to the above results, the two-sided latency is smaller for messages up to 512 Bytes. For 1 Byte messages, only 6.4 µs are required to transfer the data. Thus, SCOSCo is slower by a factor of 1.4.

The reason for this behavior can be attributed to the implementation of the `MPI_PUT`. Here, it must be checked if a mapping to the addressed window exists, and it must be established if it does not exist. In addition, checks for contiguous MPI datatypes must be performed, but those might be suboptimal compared MPICH's. However starting from 512 Byte, the data transfer latency for RCKMPI is higher than for SCOSCo and increases to the level of the message-based OSC implementation.

4.4.4 Three-Dimensional Fast Fourier Transform

To examine the performance of a real-world application, a parallel three-dimensional fast Fourier transform (3D-FFT) is analyzed. It exhibits a large fraction of communication and is suited for a realization in OSC [90, 120]. In his diploma thesis, OHMANN [87] evaluated the usage of different programming APIs, including UPC and MPI's one- and two-sided communication. With respect to their performance, no notable differences were observed for an InfiniBand-based Cluster environment [87, § 5.3]. Despite their different programming approaches the examined implementations do not differ significantly regarding the code used for communication. The MPI source code that was created within the progress of the diploma thesis is re-used for the experimental investigations in this section.

Application Overview

The (discrete) three-dimensional Fourier transform is based on the repeated application of the one-dimensional Fourier transforms. In general, a Fourier transform, converts periodic signals from the spatial domain into their matching frequency spectrum. The mathematical key concept is that a signal is the result of the superposition of trigonometric functions with different frequency and amplitudes. A deeper discussion is out of scope of this thesis. Details can be found in OHMANN [87, § 2] and BUTZ [149].

A 3D-FFT operates on a regular three-dimensional grid of complex numbers. The outcome of the transform is as if the one-dimensional transform was successively applied to every dimension of the grid. Applications of the (multi-dimensional) FFT range from signal processing to solving partial differential equations.

The benchmark employed in the analysis constitutes a kernel of an application solving such an equation. It comes from the NAS parallel benchmark suite which was developed at NASA and is named FT. As stated in the benchmarks description, *"it is a rigorous test of long-distance communication performance"* [150]. Thus it is suited to serve as a benchmark for the SCOSCo implementation.

The NPB benchmark defines several problem classes. This includes both the size of the grid and the number of iterations. Within a single iteration, a full 3D-FFT of the compute domain is computed. Table 4.1 lists the problem classes used for the analysis within this section.

Table 4.1: *Parameters of the NAS FFT benchmark classes.*

Class	Iterations	Problem Size	Messages	Transf. Data
S	6	$64 \times 64 \times 64$	15 360	39 MB
W	6	$128 \times 128 \times 128$	30 720	78 MB
A	6	$256 \times 256 \times 128$	61 440	1.3 GB
B	20	$512 \times 256 \times 256$	409 600	17 GB

Benchmark Implementation

The implementation used for the analysis is based on the UPC version of the benchmark[5] which itself has been converted from the original Fortran source. In this benchmark, a two-dimensional domain decomposition is applied to the three-dimensional grid of complex numbers. As a result, each process owns a chunk of memory in which only one dimension of the grid is completely present.

To compute the FFT for a single dimension, it is required to have all grid elements of that dimension in the local memory. After the first FFT, this condition is not met anymore. Thus, communication is required to exchange the elements such that the data is transposed between each application of the one-dimensional FFT. This includes exchange of the data and rearrangement of the received data such that it is correctly transposed. Figure 4.16 illustrates the exchange of data with different colors and shades. Note the changes in the coordinate systems which represent the orientation of the data.

In order to optimize the performance of the 3D-FFT, the exchange of already computed FFTs can be overlapped with computation of the next ones [90]. That is, the data is not communicated in a single large block after all local FFTs have been completed (Figure 4.17a). Instead, a single plane is communicated in a non-blocking fashion as soon as it was transformed (see Figure 4.17b). More fine-grained schemes are possible [90] as shown in Figure 4.17c, but they turn out to cause large overheads [87, 151]. Subsequently, they are not disucssed in this thesis.

Two basic versions of the benchmark are used. They differ in the style of communication used for the data transfer. The first version uses non-blocking two-sided calls (`MPI_ISEND` and `MPI_IRECV`) for the communication. The second one employs one-

[5] https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/npb-upc-ft/

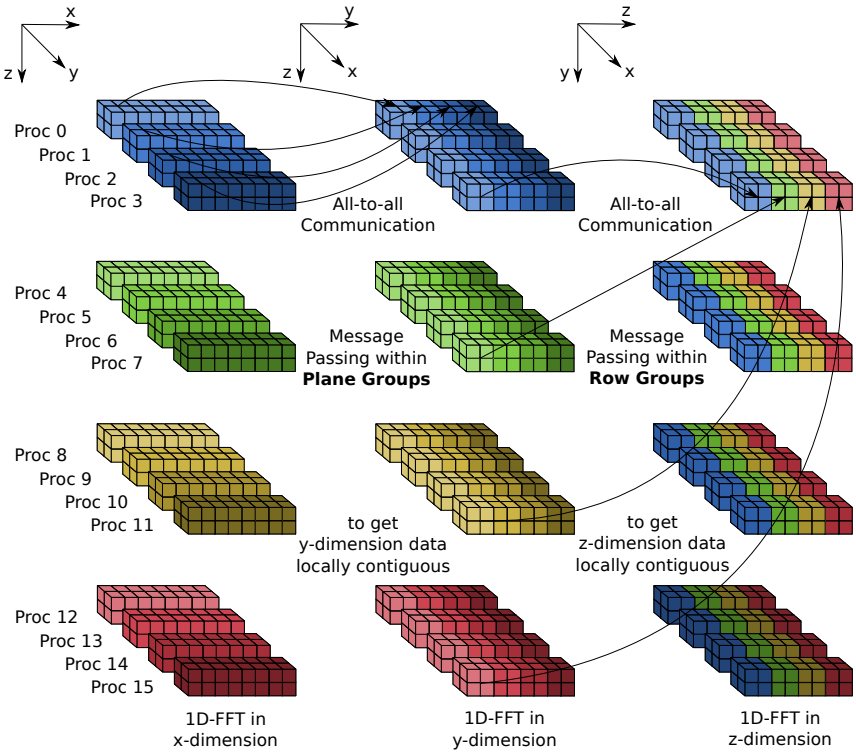
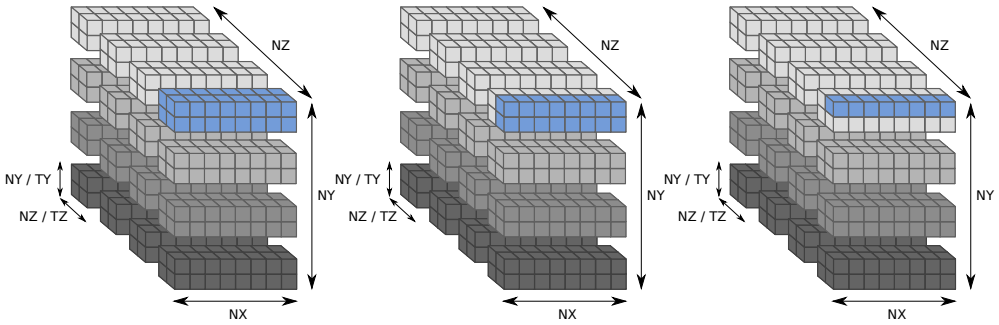


Figure 4.16: *2D domain decomposition and communication of the 3D-FFT [87].*

sided communication using `MPI_PUT`s. GATS is used as synchronization style. Due to the limited number of communication partners, this synchronization style is the best fit from an application programmer's perspective. However, fence synchronization may achieve similar performance [87, p. 49].

In case of the OSC version, the computed FFT slabs are moved to a remote window which is solely dedicated to data transfer. In a subsequent transpose step, a target process takes the transferred FFT results out of the window, transposes them, and finally rearranges the data in another buffer for performing further FFTs. This combination of local computation, data transfer, and transposing data is performed twice per time step, followed by a third FFT.

In essence, the application only reads from the local window memory during the transpose step. The local window is modified only by the RMA operations. After the aforementioned steps, pre-defined elements of the compute domain are summed and checked against pre-computed values.



(a) *packed slabs* (b) *slabs* (c) *pencils*

Figure 4.17: *Communication variants of the 3D FFT benchmark [87].*

In the experimental study the two-sided version is used to compare the performance of SCOSCo against an implementation of the benchmark that uses both explicit messages and the fast on-chip Message Passing Buffer. As underlying MPI library, RCKMPI is used. This program version is denoted as *RCKMPI/2SC*.

For the one-sided benchmark version, two versions are employed in the following. One uses RCKMPI and thus the message-based OSC implementation with deferred synchronization (see Section 3.3.1). This is labeled *RCKMPI/OSC*. The other one uses the implementation of SCOSCo and is labeled accordingly. All three benchmark variants use FFTW[6] (version 3.3.4) as basic FFT library.

Performance Considerations

As discussed at the end of Section 4.4.2, the MPBT-WT memory type is used for both the local and remote windows in the experimental evaluation to assess the performance of SCOSCo despite the bad performance delivered by that memory type. Because of the SCC's behavior for MPBT-WT, a high number of non-cacheable memory accesses will occur when the local window is accessed. In case of the FFT, this affects the aforementioned transpose step. As a result, the benchmark's runtime for the transpose step is likely to increase for the SCOSCo variants and thus may increase the overall runtime as well. The benchmark versions that use RCKMPI's channel implementation will not face this situation. This channel is not able to map memory with different memory types and always use conventional, i.e. DCM, memory for windows.

[6] http://www.fftw.org, last accessed 2016-10-01

Results

The benchmark was executed for the problem classes listed in Table 4.1 using a static process layout of 4×8 processes which were assigned to the cores 0 to 31. The compute domain of the next larger problem class C has a size of $512 \times 512 \times 512$ elements. With two double precision floating point values (8 bytes) for each complex value of a grid point, this results in a overall data size of 2 GB. The available POPSHM memory which contains the window memory has exactly the same size ($32 \times 64\,\text{MB} = 2\,\text{GB}$). However, some of SCOSCo's internal data structures are allocated in that memory as well and therefore prevent usage of class C in the experiments.

During the experiments, three measurements were performed for each of the problem sizes. All time stamps were recorded with `MPI_WTIME` that is implemented using the `RDTSC` machine instruction. No frequency scaling was performed during the experiments, so the readings of the time-stamp counter deliver stable data. The MFLOPS rate was obtained following the original NPB source. For each of the examined NPB classes, the MFLOPS rate showed less than 3% deviation between the individual measurements. As a result, the median is taken as reported value.

The results, in terms of MFLOPS, are shown in Table 4.2. Using RCKMPI, the OSC variant of the benchmark is always slower compared to the two-sided implementation (2SC). However, for all NPB classes, both the one-sided and the two-sided versions of RCKMPI are outperformed by SCOSCo. In the fourth column of Table 4.2, the MFLOPS performance using the de-facto uncached MPBT-WT memory for the window is shown. Compared to RCKMPI's OSC version, the gain in performance ranges from 19% (for class A) up to 30% for class S. For the RCKMPI-based versions and the SCOSCo MPBT-WT variant, the checksum verification of the computed result succeeded in all experiments.

To analyze the reason for this improvement and identify possible side-effects, the contribution of communication to the overall runtime is examined for each of the NPB problem classes and each of the benchmark versions. For this purpose, the individual runtimes of all 32 processes are summed up and broken down into the time for computing the FFT on the local sub-domain, performing the communication of the obtained results, and transposing the received input data into the required local format (see above). Communication includes both data movement (`MPI_ISEND` and `MPI_PUT`) and synchronization operations (`MPI_WAITALL` and GATS calls). In addition, the time for internal bookkeeping, like memory initialization, and for barriers to synchronize the computation stages is recorded.

Table 4.2: *MFLOPS performance of the NPB FFT benchmark.*

	RCKMPI		SCOSCo			
Class	2SC	OSC	MPBT-WT	Impr.	DCM	Impr.
S	512.9	477.2	618.0	30%	639.4	34%
W	514.8	487.7	586.5	20%	712.2	46%
A	595.3	577.9	685.9	19%	835.1	45%
B	483.9	473.7	569.2	20%	663.7	40%

From the illustrated breakdown (shown in Figure 4.18), the reason for the improved application performance can be clearly identified: The share of communication on the runtime is reduced drastically. In case for class B the time is reduced by a factor of 4.3 from about 2100s (for both two-sided and RCKMPI one-sided) down to 524 s for SCOSCo using MPBT-WT memory. For class A, the overall communication cost is even reduced by a factor of 4.95.

For class B, the aggregated time for communication is composed of 414 s for RMA operations, i.e. data transfer, and 110 s for synchronization operations. This equals 26% for invoking the PSCW routines. Note that this large fraction is caused by process skew as the time for computing the FFT and performing the transposition differs per process because of the SCC's NUMA characteristics. For class A, W, and S the share of synchronization routines is 16%, 17%, and 14%, respectively. The process skew is observed in those cases as well.

The results from Figure 4.18 also reveal that the different communication schemes and their implementation do not affect the local FFT computation and the NAS-specific bookkeeping steps. Hence, the approach of SCOSCo has no negative side effect on the application's computation. However, barriers take even less time (for class S and W) compared to the other program versions when SCOSCo is used. We attribute this to different process skews.

A significant change in runtime share is observed for the transpose step. When the MPBT memory is used, its runtime is tripled compared to the two RCKMPI benchmark variants. The reason for this is the bad load performance of the MPBT-WT memory type that is used for the local window. Thus, reading from the local window during the transposition step causes a performance degradation as discussed in the previous subsection.

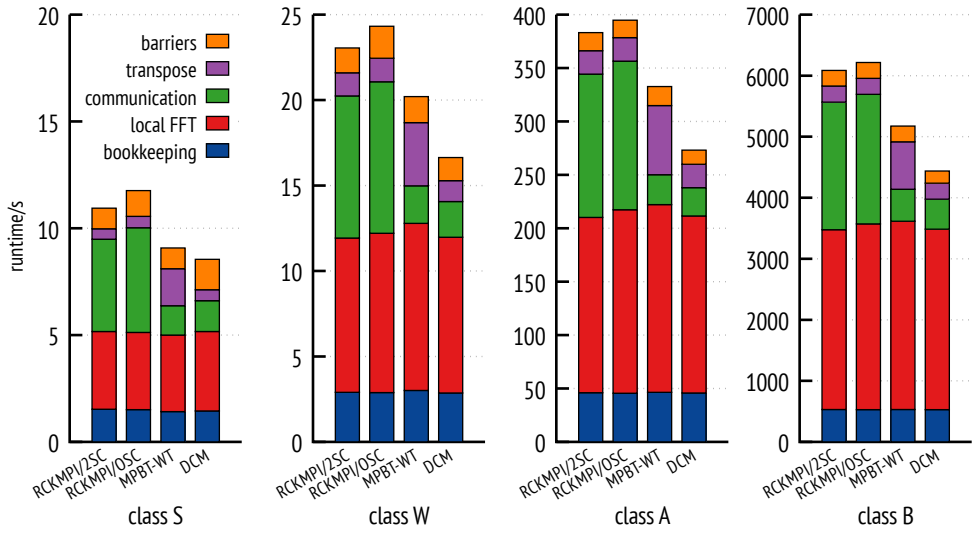


Figure 4.18: *Runtime breakdown of the 3D FFT. Note the different runtimes scales.*

To assess the performance when MPBT memory would cache the data, we conducted the same experiments as above using DCM memory for accessing the local window. The remote window was mapped using MPBT-WT. The implemented cache coherence operations remained active and therefore are included in the timing data.

Because DCM allows no (efficient) invalidation of cached data (the `CL1INVMB` instruction only affects MPBT memory), stale data from the window is read if it is not removed from the cache otherwise. As a result, wrong checksums are observed for class S and W.

In contrast, class A and B show correct results since the stale data is evicted from the cache by the behavior of the application which operates on much more data. The eviction/invalidation would normally be managed by SCOSCo. The timings for the two large classes therefore show what would be possible with a correct working, i.e. cacheable, MPBT-WT memory.

Due to the DCM memory, the transpose performance reduces to the time found in the RCKMPI variants (see Figure 4.18). With the time for data transfer unchanged (due to the usage of MPBT-WB memory for the remote window), the application's runtime decreases and the performance is significantly increased (see last column of Table 4.2).

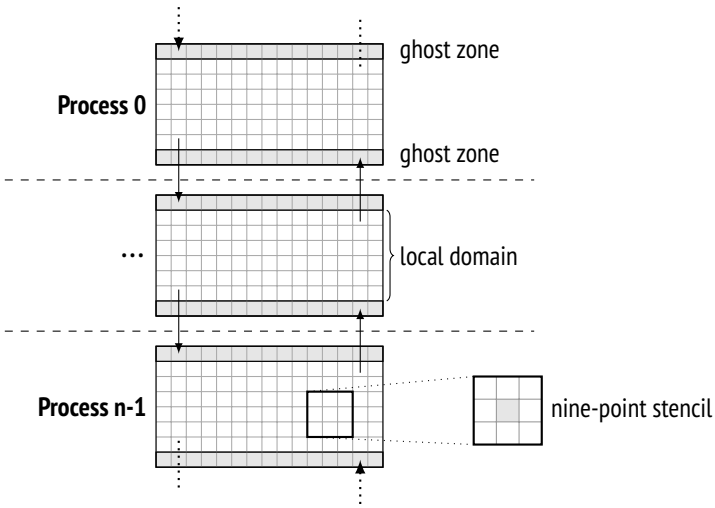


Figure 4.19: *Stencil and domain decomposition of the cellular automaton.*

4.4.5 Cellular Automaton

A prominent type of applications in the field of high-performance computing are stencil computations. Within this section, the one-sided implementation of an instance of this class is examined with respect to its performance using the SCOSCo implementation.

Application Overview

The regarded stencil application is a CA. It is based on the automaton presented by SANDERS AND WORSCH [152, § 4][7]. Each cell of regular two-dimensional grid has two possible states. The application performs a given number of iterations and evolves the state of each grid cell according to a given automaton. The inputs for the per-cell automaton are the state of the considered cell and the states of the surrounding eight cells. Thus, the application uses a nine-point stencil with a Moore neighborhood and radius of one as illustrated in Figure 4.19.

In the implementation, two arrays are employed. The first one holds the current state of the grid, while the second one is used to store the outcome of the state transition.

[7] http://www.cs.uni-potsdam.de/bs/research/labsCa.html, last accessed 2016-12-23

Pointer switching is used to swap both arrays at the end of a time step. As a result, neither the code that computes the stencil must be replicated nor a transfer between the arrays must be performed.

Parallel Versions

The above description applies both the serial and the parallel versions of the applications. Two different parallel versions were created for the purpose of this thesis. They share a domain decomposition approach but differ in the way communication is performed.

Both versions use one-dimensional domain decomposition. That is, each process allocates only a fraction of the overall grid in its local memory. To avoid large load imbalances, excess lines occurring when the number of grid lines is not a multiple of the process count are distributed among the processes. To compute the lines at the horizontal border of the local grid according to the stencil, the states from border lines of remote processes are required. Those lines are buffered locally as so-called *ghost zones*.

Those ghost zones also allow the overlap of communication and computation. To achieve this overlap, the application first computes the local grid boundary lines and initiates their transfer to the remote ghost zones. This includes the early initiation of an according reception of the remotely computed data in the local ghost zones. While these communication steps may still be in progress, the computation of the remaining i.e. the inner lines of the local grid can be conducted.

In the first parallel version of the CA, two-sided communication is used to transfer the ghost zones. `MPI_Irecv` and `MPI_Isend` are used in conjunction with `MPI_Waitall`. The receive operation is initiated first to receive the ghost zones required for the next time step. After that, the local boundary is computed which in turn precedes the transfer of those lines. Then, the remaining local grid is computed. The wait operation is issued at the end of the time-step to complete the send and receive operation. Appendix B.3.1 provides code of the time step.

The one-sided version uses GATS (see Section 3.1.3) for the synchronization along with `MPI_PUT`s for the data transfer. Two windows are created, one for each of the two arrays (see above). Depending on the time step, the appropriate window which receives the computed boundaries is selected. On each process, access and exposure

epochs are opened at the same time to allow both the access of remote processes to the local window as well as transfer manipulation of the remote ghost zones. Overlap is enabled by the application by opening and closing the epochs as early and as late as possible.

Memory Type Issues

Since the two data arrays are entirely placed in a window, all of their memory has the same type. For a fair comparison between all three versions, the memory type has to be equal. However, the two- and one-sided RCKMPI versions are not able to map the data array with a memory type other than DCM which is the memory type of conventional memory allocations. While it would be possible to put only an additional buffer that contains the ghost zone data it would create an application version that artificially contradicts MPI's one-sided programming paradigm. It would require the application to copy data from the window into the real ghost zone buffer. Consequently, the DCM memory is chosen for the SCOSCo variant of the CA as well. Using other memory types reduces application performance significantly due to limited or missing caching. However, DCM prevents correct cache coherence management as outlined in Section 4.2.4.

As a result of the DCM usage, the computational result is incorrect because the requirements for correct one-sided communication for the nCC architecture (see Section 4.2.3) are not fulfilled. This causes transferred, i.e. `PUT`'ed, ghost zone data to be overridden with stale data that is evicted from the cache due to the application's behavior.

Nevertheless, the evaluation using the DCM memory type is considered to provide an upper bound for the application's performance using SCOSCo. Correct results can be obtained if an invalidation of memory is available. Especially an invalidation of potentially cached data that was affected by the ghost zone transfer would be beneficial. However, since neither such a mechanism nor a lightweight invalidation is available for DCM, valid computational results are not achievable without large performance penalties.

In addition, using other memory types for the evaluation is not possible. RCKMPI lacks support for specifying the memory type of a window. Even if it would be possible, three memory types are left, but those expose other disadvantages: NCM exhibits huge performance penalties since the cellular automaton is heavily memory-

bound. Thus, the computation takes considerably more time than the communication which will hide potentially improvements. The same applies to MPBT-WT which has shown to behave like uncached memory for reads. Finally, the MPBT-WB type suffers from the same problem as DCM which prevents efficient invalidation or flushes of cached data.

Results

For assessing the performance impact of SCOSCo on the application, both the two-sided and the one-sided version of the automaton are examined. For comparison, the one-sided version uses either the message-based RCKMPI implementation of MPI's OSC or the SCOSCo implementation.

The experiments were conducted in the same environment as in the previous ones. The automaton was configured to have 8190 elements per line of the compute grid. Including the two vertical border cells that mimic torus-like boundary conditions (see Figure 4.19), the size of a single line in the grid is exactly 8 KB. This equals the message size during ghost zone exchange.

The number of grid lines was kept constant at 2400 and a strong scaling analysis was conducted. Even for the maximum number of 48 processes each of them computes 50 lines and therefore holds 400 KB per each of the two grid arrays. This clearly exceeds the size of the cache which avoids super-linear speedup.

As depicted above, the strong scaling analysis was performed despite the presence of wrong computational results. The local windows were mapped using DCM memory. Whereas remote windows use the MPBT-WT as in the 3D-FFT (see previous Section). The number of iterations, i.e. time steps, of the benchmark was set to 100. The time to complete all iterations loop was recorded. Initialization overhead is therefore not accounted in the results. Further, each of the three application versions was executed three times per process count. The median of these three times was selected to compute the speedup. The individual timings showed deviations of less than 1.7% on average. The employed serial baseline is shared among all versions. Its runtime was 401.05 s.

The results of the strong scaling are shown in Figure 4.20. Compared to the scaling on the InfiniBand-based cluster from Figure 4.6, the nearly ideal speedup is not achieved. The reason for this outcome is the NUMA-characteristic of the SCC. Figure 4.21

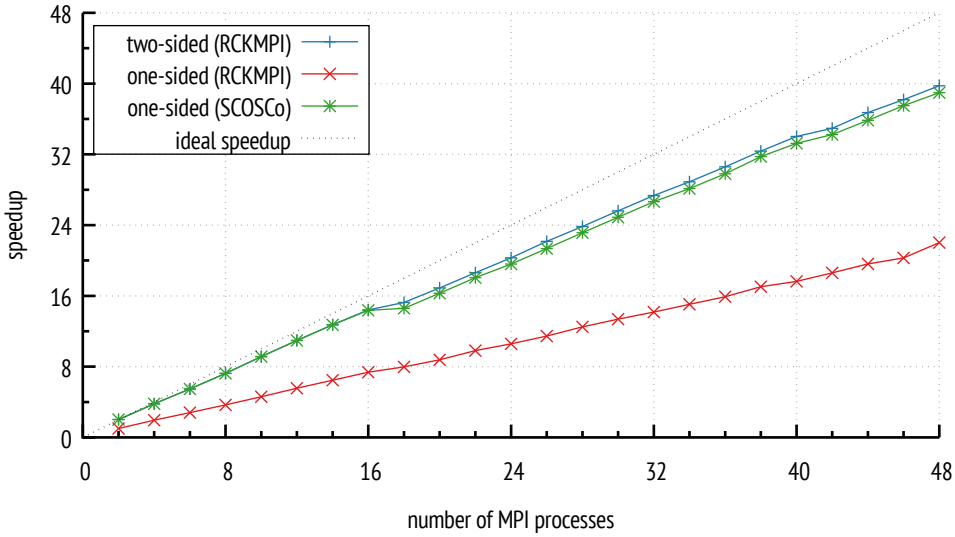


Figure 4.20: *Strong scaling of the cellular automaton application.*

shows the per-process time to compute the local array for the median run from Figure 4.20 using all 48 cores. Due to different times to compute the local domain, process skew is introduced. Because of the CA's torus-like process topology, this affects the whole application and reduces the parallel efficiency.

In the results, RCKMPI's one-sided communication exhibits a low scaling which was already discussed in Section 4.1.2. In contrast, the SCOSCo version of the automaton scales well and nearly identical to RCKMPI's two-sided version with a slight advantage for the latter one.

On the one hand, these results show a clear benefit of the SCOSCo version over RCKMPI's message-based OSC implementation. On the other hand, SCOSCo performs slightly worse than the message-based two-sided variant. This appears to contradict the previously presented results which indicated superior performance in micro-benchmarks (see Section 4.4.3) and communication-intense applications (cf. Section 4.4.4) where SCOSCo clearly outperforms two-sided communication.

To analyze the reasons, the recorded application runtime was split, similar to the analysis in Section 4.4.4. The time for computation, communication (`MPI_ISEND/IRECV` and `MPI_PUT`), as well as synchronization (`MPI_WAITALL` and the GATS routines) was recorded separately. The communication and synchronization costs were grouped as they represent the overhead introduced by the parallelization.

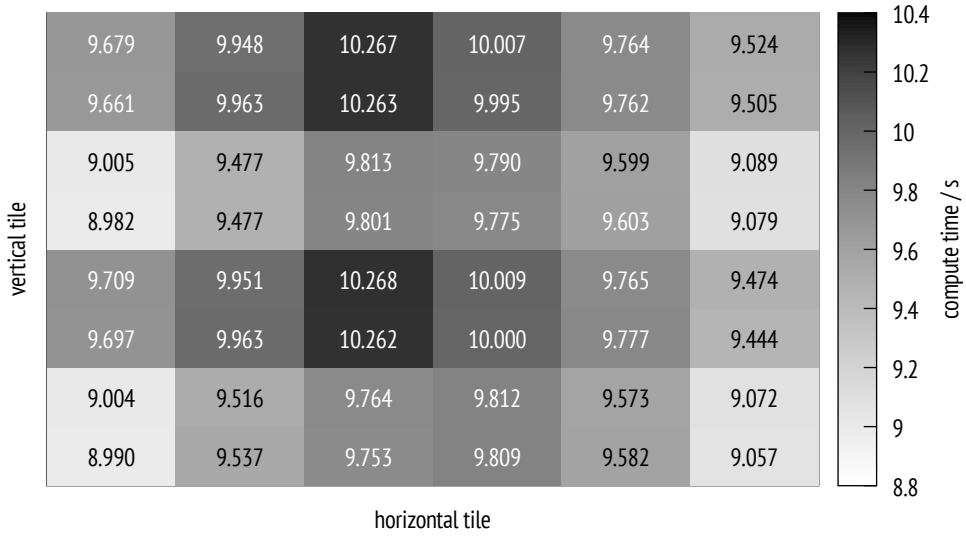


Figure 4.21: *Time to compute the CA's local domain using the whole SCC.*

Table 4.3 shows the obtained runtime shares of the application runtime. The timing data was taken for rank `0` from the same run that was used as median to compute the speedup in Figure 4.20. Results are shown for the runs with 24 and 48 processes only, but similar results were observed for all process counts larger than 16.

The data reveals that the time for data exchange (communication + synchronization) closely matches the two-sided RCKMPI and the one-sided SCOSCo variant of the CA. However, the share of computation differs slightly although the algorithmic structure of the kernels are kept equal (see Appendix B.3).

The reason for this lies in the implementation of the non-blocking since two-sided communication which is actually deferred. Upon a non-blocking send, the library tries to send data to the receiver. However, at most one internal MPICH packet is sent. If the data could not be communicated at once, the remaining data is transferred during the synchronization calls (`MPI_WAITALL`, e.g.).

In case of the CA this means that the majority of boundary data is actually communicated at the end of of each time-step. As a result, their data is copied from the application to the MPB. Thereby, the boundaries are also fetched into the cache again. When the next time-step starts, the new boundaries are computed first and the old boundary data can be fetched from the cache and the computation runs

Table 4.3: *Runtime breakdown of the cellular automaton application.*

	24 processes		48 processes	
	two-sided	SCOSCo	two-sided	SCOSCo
computation time / s	17.270	17.897	8.644	8.990
comm. + sync. time / s	2.473	2.568	1.442	1.301

faster. The SCOSCo version does not offer this benefit to the application since data is communicated directly when the communication calls are issued. The cache benefit outweighs the benefit from the communication and therefore causes better speedup values. Nevertheless, the absolute runtime differences are small.

In order to proof that the communication share was reduced by SCOSCo, the runtime breakdown is also applied to the one-sided RCKMPI version of the CA. Figure 4.22 shows the summed communication and synchronziation times for the median run from Figure 4.20. As apparent from the data, the deferred message-based OSC implementation is clearly outperformed by SCOSCo and also by the two-sided non-blocking implementation.

Since both the two- and the one-sided RCKMPI version used deferred data transfers and rely on messages, the reason for the high amount of communication time must be found at deeper levels. Concerning the one-sided version, more efforts are required in processing the internal messages. In addition, the OSC synchronization contributes additional message exchanges (see Section 3.3.1) to the runtime. With SCOSCo, these disadvantages are removed and a fast implementation of one-sided communication is provided.

4.4.6 Summary

Overall, the results clearly show the performance benefits of the SCOSCo approach even in the presense of a low memory performance due to a possible bug of the SCC's hardware. Both explicit message passing and and the implicit usage of messages for implementing one-sided communication have been outperformed by the implementation of the truely one-sided SCOSCo concept. It was also shown that software-managed cache coherence does not have negative performance impacts which confirmed findings from the literature (see Section 2.4). In addition, the hardware and software features of the SCC were beneficial for the SCOSCo implementation.

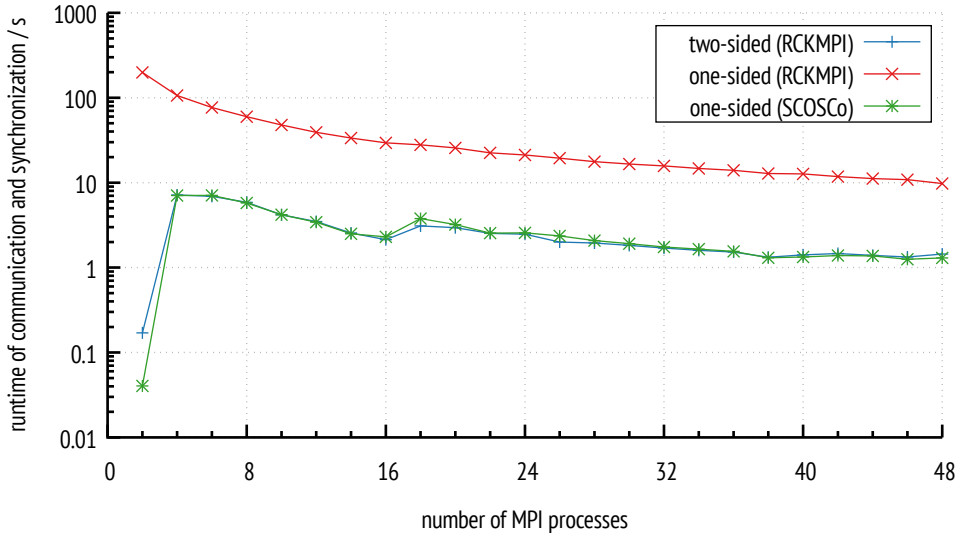


Figure 4.22: *Time for communication and synchronization of the CA.*

4.5 Possible Optimization

Although the results presented above show significant performance improvements, further optimizations are possible. As introduced in Section 3.1, the MPI synchronization calls allow to specify so-called assertions. Those describe the application behavior and allow the MPI library to omit certain steps inside the implementation when an assertion is provided.

The MPI standard defines some pre-defined assertions. Among them are `MPI_MODE_NOSTORE` and `MPI_MODE_NOPUT` which apply to `MPI_WIN_FENCE` and `MPI_WIN_POST` calls which open an exposure epoch. The assertions specify that the window was not subject to local stores before the exposure epoch (`NOSTORE`) and is not going to be modified by remote `PUT`/`ACCUMULATE` operations within the exposure epoch, respectively. As the standard states, they *"may avoid the need for cache synchronization"*.[65, p. 450]

In case of SCOSCo, these predefined assertions can be used to omit some of the required steps in the cache coherence management. If the `NOSTORE` assertion is provided, the application guarantees that the local window was not modified before the synchronization. In that case, committing the data into the RAM, i.e. the public copy, is

not required since it still holds the most recent data. Regarding the NOPUT assertion, it would enable to omit the cache invalidation at the end of an exposure epoch. This is because no modification of the public window copy was performed. As a result, the private copy still contains valid data.

Beyond those pre-defined assertions and their semantics, further ones can be considered for SCOSCo. The NOPUT can be applied to access epochs as well. When an access epoch is started with that assertion, the flush of outstanding writes that might be buffered within the WCB at its end can be omitted.

4.6 Conclusions for Future Systems

From the experiments and the theoretical discussion in the previous sections, recommendations for potential future non-cache-coherent architectures are compiled to support SCOSCo.

4.6.1 Configurable Shared Memory and Memory Registration

To define arbitrary memory regions as being shared, a feature like the dynamically reconfigurable LUTs is a very basic requirement for the presented SCOSCo approach. In case of the SCC, it is required to know the system address of the shared memory region (see Section 4.2) to configure the LUT accordingly. As it is possible to configure any memory region as shared, the direct manipulation of the LUTs with system addresses is usually a dangerous approach as it subverts the memory protection mechanisms of the operating system and the CPU's memory management unit. Consequently, an API has to be provided by both the hardware and the operating system to support the middleware in creating shared memory regions between the cores at runtime.

With respect to the concept of MPI windows, the API should provide means to collectively create a handle to which memory regions can be attached. During the collective window creation such a handle is allocated by a single process and exchanged to all other processes. With that shared handle, the API must allow registration of a local memory region (a base pointer and size) along with an unique identifier, like the rank of the process performing the registration. With the help of the operating system, the physical and thereby the system address is determined and attached to the

shared handle. When a target window is accessed, the shared handle and the rank of the target process (stored as unique identifier during registration) are used to create a mapping to the shared memory in the local virtual address space.

This approach is similar to memory registration used in InfiniBand applications and lifts SCOSCo's restriction to use memory from the POPSHM area for windows. In addition, the concept of POPSHM is generalized and is not restricted to a fixed number of pages reserved by the kernel (see Section 2.2.6). In conjunction with the process identifier attached to memory location, the presented API serves as an abstraction to the window database from Section 4.3.1.

4.6.2 Guaranteed Commit to RAM

To fulfill both requirements 1a and 2b from Section 4.2.3 cached data must be committed to RAM. For this purpose, two approaches can be considered.

Write-Through Memory

A cache configuration that allows write-through behavior for all cache levels with little to no performance penalty would support the software managed cache coherence to a great extend (cf. Section 4.2.4). Additionally, a write combine buffer that is explicitly flushable (like on the SCC), e.g. with a memory write barrier used in contemporary CPUs, should be present to speed up performance of such memory for write accesses.

Selective Cache Line Write-Back

The previous approach requires to have a write-through configuration of the page table entries that are used for the window memory. In the concept of SCOSCo, this is ensured by using a mapping with the MPBT-WT memory type. However, a write-through configuration might be slower than a conventional write-back setup. In addition, the page-based configuration granularity of cache behavior causes a performance degradation for all accesses on that page, not only for the window data, which can reside on the stack as well.

Instead of using such a write-through configuration, a forced write-back of modified cache lines would achieve the aforementioned requirements. This is actually supported by the IBM Power and the most recent Intel instruction set architecture (Intel 64) with instructions like `dcbst` [153, p. 773] and `CLWB` [154, p. 3-146] respectively. During `MPI_WIN_POST` and `MPI_WIN_COMPLETE` those instructions can be used to make changes on the given window visible in RAM. However, the performance of that approach has to be analyzed.

4.6.3 Selective Invalidation of Cache Lines

To invalidate a cached region of memory containing window data (see requirements 1b and 2a), an invalidation instruction like the `CL1INVMB` is required. However, the instruction suffers from drawbacks in its current version. The instruction invalidates all cache lines of this type. If both local and remote memories are mapped with a MPBT memory type, its invocation removes the cache lines of both memories, even in the case where only the cached memory of the local window memory needs to be invalidated (cf. requirement 1b in Section 4.2.3). Moreover, such a general invalidation may also affect other memory regions, like other windows or even memory of other processes using the same memory type. This maliciously influences application performance.

Selective Line Invalidation in Local Caches

With respect to these drawbacks, more fine-grained invalidation is required to prevent wasteful invalidation. Therefore, it would be beneficial to supply the starting (virtual) address and the size of the region to the invalidation instruction. Similar to the selective cache line write-back (see previous subsection), the invalidation of the local window (according to the requirements) could be performed by the middleware.

However, such a selective invalidation could be used to omit the invalidation of the whole window memory on the origin side. To fulfill requirement 2a with respect to `GET` operations, the invalidation may only be done when such an operation is issued and only invalidates the cached data that will be actually accessed. Since the virtual address used to access the mapped remote window memory is known when a `GET` operation is issued, the `MPI` implementation can invalidate according cache lines which may contain outdated data. The subsequent accesses to the remote window are then

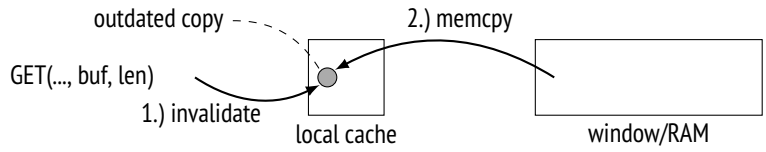


Figure 4.23: *Local selective cache line invalidation for GETs.*

guaranteed to result in caches misses. Consequently, the most recent data is retrieved from the window when it is accessed (see Figure 4.23).

Selective Remote Cache Line Invalidation

Thinking the previous idea further, such an address based approach can also be used to avoid whole window invalidation at the target (req. 1b) if explicit invalidation of remote cache lines is supported. In that case, an origin process could remotely invalidate the cache lines which will be affected by `PUT` operations and transfer the data as illustrated in Figure 4.24. This ensures that the target fetches the most recent data when its exposure epoch ends. Since, the target will not access the modified data before that epoch ends, the order of data transfer and remote cache invalidation inside the `PUT` operation does not matter.

To support such an operation in hardware, an instruction that invalidates a line with a given system address in all caches of a given (remote) core would be required. The system address can be derived from the parameters of the `PUT` operation and the window database (or generalized approach from Section 4.6.1). Side effects from the invalidation of remotely cached data, such as deletion of modified application data, are not expected for the MPI OSC use-case since processes have synchronized before communication and therefore committed any changes on window memory to RAM. In addition, target processes should not access their exposed window memories according to the standard [65, § 11.7].

4.6.4 Non-blocking Data Transfer

To enable non-blocking communication calls as required by the MPI standard, an efficient data transfer mechanism like a DMA engine would be beneficial. VAN TOL ET AL. [141] proposed the concept of *copy cores*. With such a component, communication/computation overlap would be possible.

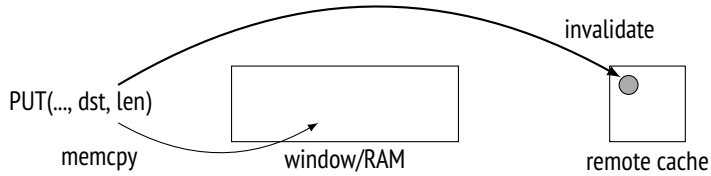


Figure 4.24: *Remote selective cache line invalidation for PUTs.*

4.7 Summary

After the previous chapter dealt with the synchronization of MPI's one-sided communication on non-cache-coherent many-cores, this one discussed the communication part on those platforms. Due to the consciously abandoned hardware support for cache coherence, a software-based approach was developed for the SCC.

In the outcome, SCOSCo a concept that uses shared memory and realized coherence transparently within the MPI synchronization calls was developed. With that, according requirements were formulated which include the need for fast and selective invalidation as well as the possibility for guaranteed, i.e. committed, memory store operations.

Based on these general requirements, a careful discussion on memory types that match the requirements was conducted. The SCC's MPBT-WT memory-type was identified to match those requirements.

Subsequently, a prototype implementation of SCOSCo was integrated in the MPICH-based RCKMPI library. It exploits architectural features of the SCC, such as configurable lookup tables to define shared memory, an explicitly flushable write combine buffer, and the POPSHM software library for shared memory.

However, based on the experimental observations, a performance critical malfunction in the SCC's memory subsystem that affects the load operations on the chosen MPBT-WT memory was documented. Using this memory for local application data was therefore not considered. Nevertheless, when using this type along with SCOSCo for storing information in remote memory, turns out to be beneficial. In terms of bandwidth it is 2× to 5× faster than the SCC's tuned message-based implementation of one-sided communication. In terms of latency, an improvement of 1.4× to 2.7× was observed.

Considering the application performance, the communication-intense NPB 3D-FFT benchmark exhibited a significant improvement in runtime and performance in terms of MFLOPS. This was due to a 5× reduction of communication compared to a two-sided non-blocking implementation of the application. For the less communication-intense cellular automaton application, such a performance gain was not observed and the one-sided SCOSCo version shows nearly identical runtimes as a two-sided version. However, compared to the message-based implementation of one-sided communication, the runtime was halved. This shows that the SCOSCo approach can deliver superior or at least comparable performance while managing cache coherence in software.

To optimize the performance and assist implementations for future non-cache-coherent systems, recommendations from the experiences on the SCC were concluded. These primarily include the need for explicit write-backs to memory and invalidation of cache lines. Especially a remote invalidation would support implementations of software based coherence to great extend.

5 Conclusions and Outlook

This thesis focused on the one-sided communication model and its efficient realization on a shared-memory-based non-cache-coherent many-core CPU. It was discussed that such systems are becoming relevant with the still increasing number of cores on a CPU, the technical limits of cache coherence, and the advent of large-scale shared-memory systems.

One-sided communication does not rely on cache coherence and thus makes it an appropriate choice as programming model for those platforms. Within this thesis, one-sided communication was considered with a focus on the Message Passing Interface standard. Both essential aspects, i.e. synchronization and communication, were extensively discussed in the progress of the thesis.

5.1 Results and Discussion

Chapter 3 presented the design of an efficient MPI general active target synchronization scheme for the non-cache-coherent SCC. It was shown that schemes from coherent shared memory architectures can be adopted to systems without cache coherence. Despite the lack of hardware-based coherence, shared memory can be efficiently used in combination with uncached accesses to implement the synchronization. The efficiency of that approach was proven by a comparison with a tuned message-based implementation. In the outcome, the shared memory solution could outperform the message-based one in terms of synchronization latency by a factor of up to five.

As an additional benefit of the discussion, an existing classification scheme for GATS implementations was extended. Together with the presented survey and classification of contemporary implementations, the categorization provides a condensed overview of best practices which might ease the development of future synchronization algorithms. Further, the survey also revealed an erroneous scheme in a production MPI library.

Concerning the communication on a shared-memory nCC system, the software-based management of cache coherence becomes a critical aspect. With SCOSCo, a concept for maintaining the coherence with respect to the MPI standard was designed in Chapter 4.

Requirements for a software-based approach were formulated. Based on those, an appropriate choice of the memory type and the cost of the required operations were discussed. While being well-suited for the desired use-case, the selected memory type turned out to provide only very low performance; an observation that was not made in previous research but was confirmed by measurements and is supported by an involved engineer. Nevertheless, the conducted experiments using adequate substitutes for the memory type revealed significant performance improvements compared to the tuned message-based MPI implementation on the Single-Chip Cloud Computer. For both micro-benchmarks and two application workloads significant lower communication costs were observed. In case of the presented 3D-FFT benchmark, a reduction by a factor of up to five was achieved.

The presented results confirm that even with software-based cache coherence directly accessed remote memory outperforms messaged-based transfers for which the SCC's hardware was originally designed for. In addition to these experimental results, explicit flushes to memory along with local and remote cache line invalidation were identified as properties to support the presented SCOSCo approach in future many-core architectures.

The aforementioned results show that the MPI notion of the one-sided communication programming model can be efficiently supported on a nCC many-core architecture. On a conceptual level the presented work might also be applied to other programming environments that support OSC. UPC with its underlying GasNET library is a possible target for adopting the SCOSCo approach.

Nevertheless, the results of this thesis do not demand to use OSC in all parallel programs that are used on nCC systems. It has been shown in previous research that "classic" two-sided communication and its implementation in middleware can provide good application performance and scaling properties (cf. Figure 4.20 on page 155). Thus, the choice of the programming and communication model is still a task for the application programmer. With SCOSCo, it was shown that even on non-cache-coherent many-core architectures one-sided communication can compete with message passing (see cellular automaton in Section 4.4.5) and if the application fits to the OSC programming model even outperforms the two-sided communication (see 3D-FFT in Section 4.4.4).

5.2 Future Work

The discussion of the process synchronization in Chapter 3 focused on the general active target synchronization scheme from the MPI standard. An analysis of the passive target synchronization scheme was left out (see Section 3.1.4). However, it enables a more shared memory-like programming style, e.g. to facilitate programming distributed data structures [91]. Both the process synchronization and the management of the cache coherence are to be considered. GERSTENBERGER ET AL. [120] present a scalable protocol for the passive target synchronization. An analysis of its suitability for nCC architectures or an alternative scheme is required. In addition, the requirements for the cache coherence need to be reconsidered since only access epochs exist and the completion of remote accesses can be repeatedly enforced during such an epoch (cf. Section 3.1.4).

For the synchronization as well the allocation of window data and memory, atomic operations on integers were employed. On the SCC, the only possibility to implement this kind of access is the usage of the per-core test-and-set register. Different to preceding works, no contention on this scarce resource was observed (cf. Section 3.5.4). However, having only a single resource for atomic operations per core appears to be a architectural drawback of the SCC's architecture. Future work on the architecture of many-core processors might focus on more sophisticated support for atomic operations to provide synchronization primitives like atomic compare-and-swap or fetch-and-add operations.

To analyze the performance impact of the proposed forced cache flush, a performance study on processors where such operations are available (see Section 4.6) can be conducted. With this, the benefits or drawback of such an operation in the context of SCOSCo can be assessed.

It will be interesting to see if and how efficient the suggested enhancements (cf. Section 4.6) are getting implemented in future platforms. Those would also provide the opportunity to confirm the benefits of SCOSCo and the synchronization on platforms other than the SCC.

A Employed MPICH test cases

The following enumerations list the unit test cases shipped with MPICH version 3.1.3 that were used to check the RCKMPI implementation for errors (cf. Section 4.1.2) after applying the necessary fixes to get a working one-sided communication implementation. The lists shows the test case name and the number of processes MPICH's test suite uses respective outcome of the test. The list does not apply to the SCOSCo extension.

A.1 Succeeded Test Cases

rma/winname (2), rma/allocmem (2), rma/putfence1 (4), rma/putfidx (4), rma/getfence1 (4), rma/accfence1 (4), rma/adlb_mimic1 (3), rma/accfence2 (4), rma/putpscw1 (4), rma/accpscw1 (4), rma/getgroup (4), rma/transpose1 (2), rma/transpose2 (2), rma/transpose3 (2), rma/transpose3_shm (2), rma/transpose5 (2), rma/transpose6 (1), rma/transpose7 (2), rma/test1 (2), rma/test2 (2), rma/test2_shm (2), rma/test3 (2), rma/test3_shm (2), rma/test4 (2), rma/test5 (2), rma/lockcontention (3), rma/lockcontention2 (4), rma/lockcontention2 (8), rma/lockcontention3 (8), rma/lockopts (2), rma/transpose4 (2), rma/fetchandadd (7), rma/fetchandadd_tree (7), rma/wintest (2), rma/wintest_shm (2), rma/contig_displ (1), rma/test1_am (2), rma/test2_am (2), rma/test2_am_shm (2), rma/test3_am (2), rma/test3_am_shm (2), rma/test4_am (2), rma/test5_am (2), rma/fetchandadd_am (7), rma/fetchandadd_tree_am (7), rma/accfence2_am (4), rma/test1_dt (2), rma/nullpscw (7), rma/nullpscw_shm (7), rma/attrorderwin (1), rma/wincall (2), rma/baseattrwin (1), rma/fkeyvalwin (1), rma/selfrma (1), rma/mixedsync (4), rma/epochtest (4), rma/locknull (2), rma/rmanull (2), rma/rmazero (2), rma/strided_acc_indexed (2), rma/strided_acc_onelock (2), rma/strided_acc_subarray (2), rma/strided_get_indexed (2), rma/strided_putget_indexed (4), rma/strided_putget_indexed_shared (4), rma/strided_getacc_indexed (4), rma/strided_getacc_indexed_shared (4), rma/window_creation (2), rma/contention_put (4), rma/contention_putget (4), rma/put_base (2), rma/put_bottom (2), rma/win_flavors (4), rma/manyrma2 (2), rma/manyrma2_shm (2), rma/manyrma3 (2),

rma/win_shared_noncontig_put (4), rma/win_zero (4), rma/win_dynamic_acc (4), rma/get_acc_local (1), rma/linked_list (4), rma/linked_list_fop (4), rma/compare_and_swap (4), rma/win_info (4), rma/pscw_ordering (4), rma/pscw_ordering_shm (4), rma/badrma (2), rma/acc-loc (4), rma/fence_shm (2), rma/win_shared_zero-byte (4), rma/win_shared_put_flush_get (4), errors/rma/winerr (2), errors/rma/win-err2 (2), errors/rma/win_sync_unlock (2), errors/rma/win_sync_free_pt (2), errors/rma/win_sync_free_at (2), errors/rma/win_sync_complete (2), errors/rma/win_sync_lock_at (2), errors/rma/win_sync_lock_pt (2), errors/rma/win_sync_nested (2), errors/rma/win_sync_op (2)

A.2 Failed Test Cases

rma/win_shared (4), rma/win_shared_noncontig (4), rma/fetch_and_op_char (4), rma/fetch_and_op_short (4), rma/fetch_and_op_int (4), rma/fetch_and_op_long (4), rma/fetch_and_op_double (4), rma/fetch_and_op_long_double (4), rma/get_accumulate_double (4), rma/get_accumulate_double_derived (4), rma/get_accumulate_int (4), rma/get_accumulate_int_derived (4), rma/get_accumulate_long (4), rma/get_accumulate_long_derived (4), rma/get_accumulate_short (4), rma/get_accumulate_short_derived (4), rma/flush (4), rma/reqops (4), rma/req_example (4), rma/req_example_shm (4), rma/linked_list_lockall (4), rma/linked_list_bench_lock_all (4), rma/linked_list_bench_lock_excl (4), rma/linked_list_bench_lock_shr (4), rma/linked_list_bench_lock_shr_nocheck (4), rma/mutex_bench (4), rma/mutex_bench_shared (4), rma/mutex_bench_shm (4), rma/rma-contig (2), rma/get-struct (2), rma/at_complete (2)

B Source Codes Extracts

In the following, source code extracts for the employed benchmarks are presented.

B.1 Load and Store Latencies

```
        for (i = 0; i < num_tries; i++) {
            prefetch((char*) src, size);
            times[i] = rdtsc();

            asm volatile (
                    "pushl %%eax;\n\t"
                    "pushl %%ebx;\n\t"
                    "pushl %%ecx;\n\t"
                    "1:"
#ifdef READ
                    "movl (%%eax), %%ebx;\n\t" /* read */
#else
                    "movl %%ebx, (%%eax);\n\t" /* write */
#endif
                    "addl $4, %%eax;\n\t"
                    "loop 1b;\n\t"
                    "popl %%ecx;\n\t"
                    "popl %%ebx;\n\t"
                    "popl %%eax;\n\t"
                    /* output: none */
                    :
                    /* input: src -> EAX, size/4 -> ECX */
                    : "a" (src),
                      "c" (size/4)
                    : "memory"
                    /* no clobber, as we push registers */
            );
            times[i] = rdtsc() - times[i];
        };
```

This is the essential loop to measure the latency of load and store operations for different memory types. Different version were compiled to get separate binaries for load and stores operations. The memory buffer (`src`) is allocated dynamically and prefetched before each iteration (line 2). Prefetching is done from higher to lower addresses.

B.2 GATS Synchronization Benchmark

The GATS synchronization benchmark is used in Chapter 3 to assess the pure synchronization latency. The times for synchronization calls in each loop iteration is stored in an array. I/O operations are omitted as they generate more traffic on the on-chip network, compared to storing the obtained timings in the arrays.

```
1    for (i = 0; i < NUM_REPS; i++) {
2        if (comm_rank < num_origins) {
3            /* origin */
4            ta = MPI_Wtime();
5            MPI_Win_start(start_group, 0, win);
6            tb = MPI_Wtime();
7            times_start[i] = tb - ta;
8
9            /* access epoch (nop) */
10
11           ta = MPI_Wtime();
12           MPI_Win_complete(win);
13           tb = MPI_Wtime();
14           times_complete[i] = tb - ta;
15       } else {
16           /* target */
17           ta = MPI_Wtime();
18           MPI_Win_post(post_group, 0, win);
19           tb = MPI_Wtime();
20           times_post[i] = tb - ta;
21
22           /* exposure epoch (nop) */
23
24           ta = MPI_Wtime();
25           MPI_Win_wait(win);
26           tb = MPI_Wtime();
27           times_wait[i] = tb - ta;
28       }
29   }
```

B.3 Cellular Automaton

B.3.1 Two-Sided Time Step Kernel

This is a pseudo-code version for the kernel of the two-sided cellular automaton. Note that the ghost zone data for the first iteration can either be transferred before the loop or may be computed locally during the initialization phase. For the thesis, the former variant was chosen and an `MPI_Sendrecv` call preceedes the loop. Timing is performed using `MPI_WTIME` immediately before and after the shown loop.

```
for (i = 0; i < its; i++) {
    /* post receive buffer asap */
    MPI_Irecv(new_local_domain[0], ..., &req[0]);
    MPI_Irecv(new_local_domain[num_local_lines + 1], ..., &req[1]);

    /* update boundaries of local domain */
    ca_simulate(old_local_domain, new_local_domain, 1, 1);
    ca_simulate(old_local_domain, new_local_domain, num_local_lines, 1);

    /* transfer buffer asap */
    MPI_Isend(new_loocal_domain[1], ..., &req[2]);
    MPI_Isend(new_loocal_domain[num_local_lines], ..., &req[3]);

    /* update inner part of local domain */
    ca_simulate(old_local_domain, new_local_domain, 2, 1);

    swap(old_local_domain, new_local_domain);

    /* synchronize */
    MPI_Waitall(4, req, ...);
}
```

B.3.2 One-Sided Time Step Kernel

For the one-sided version, a window for the upper and lower bound in each of the two arrays for the local compute domain are used. Similar to the two-sided version, the initial ghost zone exchange is performed before the main loop starts. Timing is done as in the two-sided version (see above).

```
for (i = 0; i < its; i++) {
    wnd_idx = (i + 1) % 2;

    /* compute boundaries */
    ca_simulate(old_local_domain, new_local_domain, 1, 1);
    ca_simulate(old_local_domain, new_local_domain, num_local_lines, 1);

    /* start exposure and access epoch */
    MPI_Win_post(nb_group, 0, win_upper_bound[wnd_idx]);
    MPI_Win_post(nb_group, 0, win_lower_bound[wnd_idx]);

    MPI_Win_start(nb_group, 0, win_upper_bound[wnd_idx]);
    MPI_Win_start(nb_group, 0, win_lower_bound[wnd_idx]);

    /* transfer of boundaries */
    MPI_Put(new_local_domain[1], ..., win_lower_bound[wnd_idx]);
    MPI_Put(new_local_domain[lines], ..., win_upper_bound[wnd_idx]);

    /* update inner part of local domain */
    ca_simulate(old_local_domain, new_local_domain, 2, 1);

    /* close epochs */
    MPI_Win_complete(win_upper_bound[wnd_idx]);
    MPI_Win_complete(win_lower_bound[wnd_idx]);

    MPI_Win_wait(win_upper_bound[wnd_idx]);
    MPI_Win_wait(win_lower_bound[wnd_idx]);

    swap(old_local_domain, new_local_domain);
}
```

B.4 Communication Patterns

The following two graphics illustrate the communication patterns of the cellular automaton and the 3D fast Fourier transform when using 32 processes for each of the two application. The shade of the rectangle at the intersection of sender and receiver depicts the intensity of the communication between those two processes in terms of uni-directional transferred application data. The darker the gray level, the more data is sent. The scale is relative to the overall maximum.

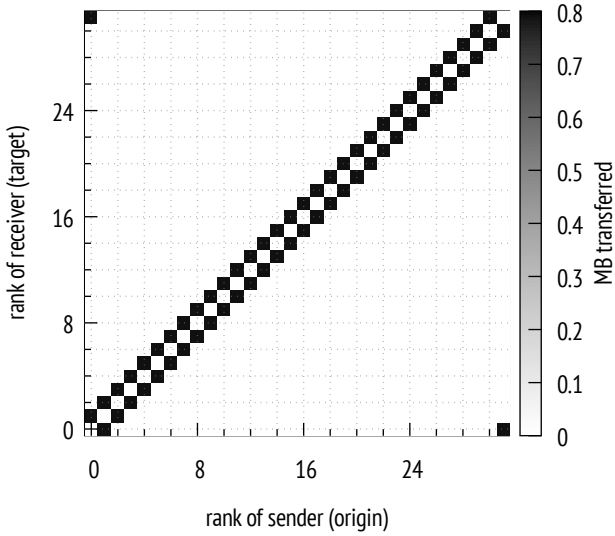


Figure B.1: *Total amount of transferred data for the CA using 50 iterations.*

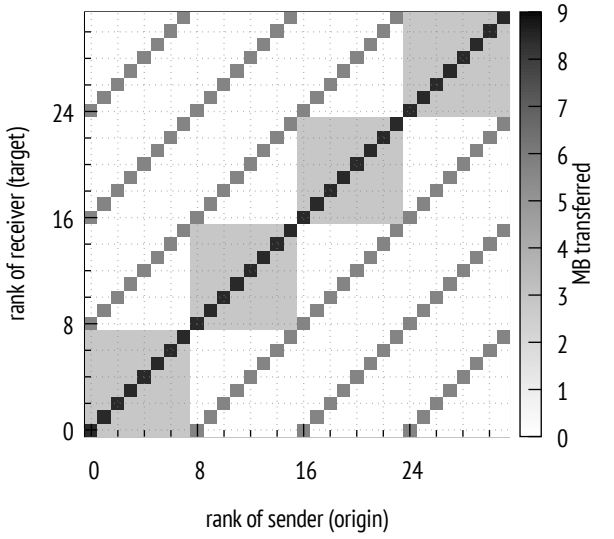


Figure B.2: *Total amount of transferred data for the 3D-FFT (class S, 32 processes).*

C Compute Cluster Properties

The following table shows the system properties and parameters of the InfiniBand-based compute cluster used for the scaling experiments in Section 4.1.2. The cluster consists of 27 heterogenous node, plus one master and one management node, connected to a single 36-port switch. The Gigabit Ethernet management network was not used for communication within the scope of this thesis.

property	value
system	Dell PowerEdge R610
processor	2x Intel Xeon E5520 (Nehalem micro-architecture)
data caches	L1: 16 KB, L2: 256 KB, L3: 8 MB
base frequency	2.27 GHz, TurboBoost and HyperThreading disabled
main memory	48 GB DDR3 1066 MHz
InfiniBand HCA	Mellanox MHGH19-XTC ConnectXZ (20 Gb/s)
operating system	Scientific Linux 5.11
Linux kernel version	2.6.18-406.el5
C compiler	GCC C Compiler, version 4.9.1
MPI library	Open MPI 1.8.2
InfiniBand switch	Mellanox MTS3600R-1UNC

List of Figures

List of Tables

Listings

List of Abbrevations

API application programming interface 31
CA cellular automaton 44
DCM definitely cacheable memory 23
HPC high performance computing 9
GATS general active target synchronization 57
LSM legacy shared memory 27
LUT lookup table 19
MIU Mesh Interface Unit 18
MPB Message Passing Buffer 21
MPBT Message Passing Buffer type 24
MPI Message Passing Interface 31
NCM non-cacheable memory type 23
nCC non-cache-coherent 15
NUMA non-uniform memory access 10
OSC one-sided communication 36
RC release consistency 11
POPSHM privately-owned public shared memory 29
PGAS partitioned global address space 37
RDMA remote direct memory access 66
RMA remote memory access 39
SCC Single-Chip Cloud Computer 16
SCOSCo software-managed cache coherence for one-sided communication . . . 109
SVM shared virtual memory 43
SPMD single programm, multiple data 30
TSR test-and-set register 25
WCB write combine buffer 22

Index

Bibliography

Author's Publications

[1] Steffen Christgau, Johannes Spazier, and Bettina Schnor. *A cross-platform performance and scalability analysis of the tsunami simulation EasyWave on different multi-core architectures*. Tech. rep. TR-2015-01. ISSN: 0946-7580. Potsdam, Germany: University Potsdam, Institute of Computer Science, 2015.

[2] Steffen Christgau and Bettina Schnor. „Awareness of MPI Virtual Process Topologies on the Single-Chip Cloud Computer". In: *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012, HIPS Workshop*. IEEE Computer Society, 2012, pp. 529–536. DOI: 10.1109/IPDPSW.2012.71.

[3] Steffen Christgau and Bettina Schnor. „Synchronization of One-Sided MPI Communication on a Non-Cache Coherent Many-Core System". In: *ARCS 2016 - 29th International Conference on Architecture of Computing Systems, Workshop Proceedings, April 4-7, 2016, Nuremberg, Friedrich-Alexander University, Erlangen-Nürnberg*. Ed. by Ana Lucia Varbanescu. VDE Verlag / IEEE Xplore, 2016.

[4] Steffen Christgau and Bettina Schnor. „One-Sided Communication in RCK-MPI for the Single-Chip Cloud Computer". In: *MARC Symposium*. Ed. by Eric Noulard. ONERA, The French Aerospace Lab, 2012, pp. 19–23.

[5] Steffen Christgau and Bettina Schnor. „Software-managed Cache Coherence for fast One-Sided Communication". In: *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. Ed. by Pavan Balaji and Kai-Cheung Leung. ACM, 2016, pp. 69–77. DOI: 10.1145/2883404.2883409.

[6] Steffen Christgau. „Optimierung von Anwendungen auf dem Single-Chip Cloud Computer“. Master's Thesis. Potsdam: Universität Potsdam, Nov. 2011. URL: http://www.cs.uni-potsdam.de/bs/research/student-thesis/thesis/Christgau_2011/MaThesis.pdf.

[7] Johannes Spazier, Steffen Christgau, and Bettina Schnor. „Automatic generation of parallel C code for stencil applications written in MATLAB“. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2016, Santa Barbara, CA, USA, June 14, 2016*. Ed. by Martin Elsman, Clemens Grelck, Andreas Klöckner, and David A. Padua. ACM, 2016, pp. 47–54. DOI: 10.1145/2935323.2935329.

[8] Bettina Schnor and Steffen Christgau. *Tagungsband – GI/ITG KuVS Fachgespräch "Sensornetze"*. Tech. rep. ISSN 0946-7580 TR-2014-1. Universität Potsdam, Institut für Informatik, 2014.

[9] Steffen Christgau et al. „A comparison of CUDA and OpenACC: Accelerating the Tsunami Simulation EasyWave“. In: *ARCS 2014 - 27th International Conference on Architecture of Computing Systems, Workshop Proceedings, February 25-28, 2014, Luebeck, Germany, University of Luebeck, Institute of Computer Engineering*. Ed. by Walter Stechele and Thomas Wild. VDE Verlag / IEEE Xplore, 2014, pp. 1–5.

[10] Steffen Christgau, Simon Kiertscher, and Bettina Schnor. „The Benefit of Topology Awareness of MPI Applications on the SCC“. In: *MARC Symposium*. Ed. by Diana Göhringer, Michael Hübner, and Jürgen Becker. KIT Scientific Publishing, Karlsruhe, 2011, pp. 47–51.

[11] Philipp Mahr, Steffen Christgau, Christian Haubelt, and Christophe Bobda. „Integrated Temporal Planning, Module Selection and Placement of Tasks for Dynamic Networks-on-Chip“. In: *IPDPS Workshops*. IEEE, 2011, pp. 258–263.

[12] Steffen Christgau and Bettina Schnor. „Exploring One-Sided Communication and Synchronization on a non-Cache-Coherent Many-Core Architecture“. In: *Concurrency and Computation: Practice and Experience* 29.15 (2017). DOI: 10.1002/cpe.4113.

[13] Johannes Spazier, Steffen Christgau, and Bettina Schnor. „Efficient Parallelization of MATLAB Stencil Applications for Multi-core Clusters“. In: *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC*. WOLFHPC '16. Salt Lake City, Utah: IEEE Press, 2016, pp. 20–29. DOI: 10.1109/WOLFHPC.2016.7.

References

[14] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)* Morgan Kaufmann, 2012.

[15] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. „Performance Engineering for a Medical Imaging Application on the Intel Xeon Phi Accelerator". In: *ARCS 2014 - 27th International Conference on Architecture of Computing Systems, Workshop Proceedings, February 25-28, 2014, Luebeck, Germany, University of Luebeck, Institute of Computer Engineering*. Ed. by Walter Stechele and Thomas Wild. VDE Verlag / IEEE Xplore, 2014, pp. 1–8.

[16] Bryan Veal and Annie P. Foong. „Performance scalability of a multi-core web server". In: *Proceedings of the 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2007, Orlando, Florida, USA, December 3-4, 2007*. Ed. by Raj Yavatkar, Dirk Grunwald, and K. K. Ramakrishnan. ACM, 2007, pp. 57–66. DOI: 10.1145/1323548.1323562.

[17] Sylvain Genevès. *An Analysis of Web Servers Architectures Performances on Commodity Multicores*. Research Report. June 2012. URL: https://hal.inria.fr/hal-00674475/file/paper_submitted.pdf.

[18] Sabela Ramos and Torsten Hoefler. „Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi". In: *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*. Ed. by Manish Parashar, Jon B. Weissman, Dick H. J. Epema, and Renato J. O. Figueiredo. ACM, 2013, pp. 97–108. DOI: 10.1145/2462902.2462916.

[19] Timothy Prickett Morgan. *More Knights Landing Xeon Phi Secrets Unveiled*. http://www.nextplatform.com/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/ last accessed 2015-11-02. Mar. 2015.

[20] Jason Howard et al. „A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS". In: *IEEE International Solid-State Circuits Conference, ISSCC 2010, Digest of Technical Papers, San Francisco, CA, USA, 7-11 February, 2010*. IEEE, 2010, pp. 108–109. DOI: 10.1109/ISSCC.2010.5434077.

[21] Jake Edge. *A look at The Machine*. online. https://lwn.net/Articles/655437/, last accessed 2017-01-06. Aug. 2015.

[22] Yves Durand et al. „EUROSERVER: Energy Efficient Node for European Micro-Servers“. In: *17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014*. online http://paul-carpenter.org/durand2014dsd.pdf, last accessed 2017-01-06. IEEE Computer Society, 2014, pp. 206–213. DOI: 10.1109/DSD.2014.15.

[23] C. A. R. Hoare. „Communicating Sequential Processes“. In: *Communications of the ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585.

[24] András Vajda. *Programming Many-Core Chips*. 1st edition. Springer Publishing Company, Incorporated, 2011.

[25] Gudula Rünger and Thomas Rauber. *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition*. Springer, 2013. DOI: 10.1007/978-3-642-37801-0.

[26] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. New York, NY, USA: McGraw-Hill, Inc., 1998.

[27] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011. DOI: 10.2200/S00346ED1V01Y201104CAC016.

[28] Leslie Lamport. „How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs“. In: *IEEE Trans. Computers* 28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[29] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. „x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors“. In: *Communications of the ACM* 53.7 (2010), pp. 89–97. DOI: 10.1145/1785414.1785443.

[30] Kourosh Gharachorloo et al. „Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors“. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture. Seattle, WA, June 1990*. Ed. by Jean-Loup Baer, Larry Snyder, and James R. Goodman. ACM, 1990, pp. 15–26. DOI: 10.1145/325164.325102.

[31] David Mosberger. „Memory Consistency Models“. In: *Operating Systems Review* 27.1 (1993), pp. 18–26. DOI: 10.1145/160551.160553.

[32] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E. Nagel. „Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture“. In: *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*. IEEE Computer Society, 2015, pp. 739–748. DOI: 10.1109/ICPP.2015.83.

[33] Avinash Sodani et al. „Knights Landing: Second-Generation Intel Xeon Phi Product“. In: *IEEE Micro* 36.2 (2016), pp. 34–46. DOI: 10.1109/MM.2016.25.

[34] Intel. *An Introduction to the Intel QuickPath Interconnect*. online. http://www.intel.de/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf, last accessed 2017-01-18. Jan. 2009.

[35] Alain Greiner. „Tsar: a scalable, shared memory, many-cores architecture with global cache coherence“. In: *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*. Vol. 15. slides, online: http://www.mpsoc-forum.org/previous/2009/slides/greiner.pdf, last accessed 2017-01-19. 2009.

[36] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. „Why On-chip Cache Coherence is Here to Stay“. In: *Commun. ACM* 55.7 (July 2012), pp. 78–89. DOI: 10.1145/2209249.2209269.

[37] Thomas J. Ashby, Pedro Diaz, and Marcelo Cintra. „Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters“. In: *IEEE Trans. Computers* 60.4 (2011), pp. 472–483. DOI: 10.1109/TC.2010.155.

[38] Michael Feldman. *HPE Unveils Prototype of The Machine*. online https://www.top500.org/news/hpe-unveils-prototype-of-the-machine/, last accessed 2017-01-19. Dec. 2016.

[39] Curt Hopkins. *An Oral History of The Machine—Chapter Five: Hardware*. online. https://community.hpe.com/t5/Behind-the-scenes-Labs/An-Oral-History-of-The-Machine-Chapter-Five-Hardware/ba-p/6918755, last accessed 2017-01-19. Nov. 2016.

[40] John Goodacre. *Scaling Mobile Compute to the Data Center*. online. keynote talk at 26th International Symposium on Computer Architecture and High Performance Computing, http://sbac.lip6.fr/2014/GoodacreKeynote.pdf, last accessed 2017-01-20. Oct. 2014.

[41] Jim Held, Jerry Bautista, and Sean Koehl. *From a Few Cores to Many: A Tera-scale Computing Research Overview*. online. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-tera-scale-research-paper.pdf last accessed 2015-11-24. 2006.

[42] Sriram R. Vangal et al. „An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS“. In: *2007 IEEE International Solid-State Circuits Conference, ISSCC 2007, Digest of Technical Papers, San Francisco, CA, USA, February 11-15, 2007*. IEEE, 2007, pp. 98–589. DOI: 10.1109/ISSCC.2007.373606.

[43] Jerry Bautista. *Tera-scale Computing and Interconnect Challenges – 3D Stacking Considerations*. Online. http://www.hotchips.org/wp-content/uploads/hc_archives/hc20/1_Sun/HC20.24.130.pdf, last accessed 2015-11-24. 2007.

[44] Timothy G. Mattson, Rob F. Van der Wijngaart, and Michael A. Frumkin. „Programming the Intel 80-core network-on-a-chip terascale processor". In: *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*. IEEE/ACM, 2008, p. 38. DOI: 10.1145/1413370.1413409.

[45] Theodore Kubaska et al. *SCC External Architecture Specification (EAS)*. Intel Labs. Nov. 2010.

[46] *The SCC Programmer's Guide*. Revision 1.0. Intel Labs. Jan. 2012.

[47] Patrick Cichowski, Jörg Keller, and Christoph W. Kessler. „Modelling Power Consumption of the Intel SCC". In: *6th Many-core Applications Research Community (MARC) Symposium. Proceedings of the 6th MARC Symposium, 19-20 July 2012, Toulouse, France*. Ed. by Eric Noulard. ONERA, The French Aerospace Lab, 2012, pp. 46–51. URL: http://hal.archives-ouvertes.fr/hal-00719033.

[48] Pollawat Thanarungroj and Chen Liu. „Power and energy consumption analysis on Intel SCC many-core system". In: *30th IEEE International Performance Computing and Communications Conference, IPCCC 2011, Orlando, Florida, USA, November 17-19, 2011*. Ed. by Sheng Zhong, Dejing Dou, and Yu Wang. IEEE, 2011, pp. 1–2. DOI: 10.1109/PCCC.2011.6108095.

[49] Ehsan Totoni, Babak Behzad, Swapnil Ghike, and Josep Torrellas. „Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs". In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software, New Brunswick, NJ, USA, April 1-3, 2012*. Ed. by Rajeev Balasubramonian and Vijayalakshmi Srinivasan. IEEE Computer Society, 2012, pp. 78–87. DOI: 10.1109/ISPASS.2012.6189208.

[50] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture - a hardware / software approach*. Morgan Kaufmann, 1999.

[51] *Cray T3D System Architecture Overview*. http://bitsavers.trailing-edge.com/pdf/cray/HR-04033_CRAY_T3D_System_Architecture_Overview_Sep93.pdf, last accessed 2015-11-30. Cray Research Inc. Chippewa Falls, Sept. 1993.

[52] Thomas Prescher, Randolf Rotta, and Jörg Nolte. „Flexible Sharing and Replication Mechanisms for Hybrid Memory Architectures". In: *Proceedings of the 4th Many-Core Applications Research Community (MARC) Symposium*. Ed. by Peter Tröger and Andreas Polze. Universitätsverlag Potsdam, Jan. 2012, pp. 67–72. URL: http://pub.ub.uni-potsdam.de/volltexte/2012/5789/.

[53] Pablo Reble, Carsten Clauss, and Stefan Lankes. „One-sided communication and synchronization for non-coherent memory-coupled cores“. In: *International Conference on High Performance Computing & Simulation, HPCS 2013, Helsinki, Finland, July 1-5, 2013*. IEEE, 2013, pp. 390–397. DOI: 10.1109/HPCSim.2013.6641445.

[54] *The SccKit 1.4.x User's Guide*. Revision 1.1. Intel Labs. Nov. 2011.

[55] Pablo Reble, Stefan Lankes, Florian Zeitz, and Thomas Bemmerl. „Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor“. In: *4th USENIX Workshop on Hot Topics in Parallelism, Poster Paper*. https://www.usenix.org/system/files/conference/hotpar12/hotpar12-final9.pdf. 2012.

[56] Jan-Arne Sobania, Peter Tröger, and Andreas Polze. „Linux Operating System Support for the SCC Platform - An Analysis“. In: *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*. Ed. by Diana Göhringer, Michael Hübner, and Jürgen Becker. KIT Scientific Publishing, Karlsruhe, 2011, pp. 31–34. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937.

[57] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. „Early experience with the Barrelfish OS and the Single-Chip Cloud Computer“. In: *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*. Ed. by Diana Göhringer, Michael Hübner, and Jürgen Becker. KIT Scientific Publishing, Karlsruhe, 2011, pp. 35–39. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937.

[58] Michael Ziwisky and Dennis Brylow. „BareMichael: A Minimalistic Bare-metal Framework for the Intel SCC“. In: *6th Many-core Applications Research Community (MARC) Symposium. Proceedings of the 6th MARC Symposium, 19-20 July 2012, Toulouse, France*. Ed. by Eric Noulard. ONERA, The French Aerospace Lab, 2012, pp. 66–71. URL: http://hal.archives-ouvertes.fr/hal-00719038.

[59] Pablo Reble, Jacek Galowicz, Stefan Lankes, and Thomas Bemmerl. „Efficient Implementation of the bare-metal Hypervisor MetalSVM for the SCC“. In: *6th Many-core Applications Research Community (MARC) Symposium. Proceedings of the 6th MARC Symposium, 19-20 July 2012, Toulouse, France*. Ed. by Eric Noulard. ONERA, The French Aerospace Lab, 2012, pp. 59–65. URL: http://hal.archives-ouvertes.fr/hal-00719037.

[60] Jan-Arne Sobania, Peter Tröger, and Andreas Polze. „Towards Symmetric Multi-Processing Support for Operating Systems on the SCC". In: *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*. Ed. by Peter Tröger and Andreas Polze. Universitätsverlag Potsdam, Feb. 2012, pp. 73–78. URL: http://pub.ub.uni-potsdam.de/volltexte/2012/5789/.

[61] Jay Hoeflinger and Larry Meadows. *OpenMP on Clusters*. last accessed 2017-01-09. May 2006. URL: https://www.hpcwire.com/2006/05/19/openmp_on_clusters-1/.

[62] Leonardo Dagum and Ramesh Menon. „OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. DOI: 10.1109/99.660313.

[63] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.5*. Nov. 2015. URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[64] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[65] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Stuttgart: High Performance Computing Center Stuttgart, June 2015.

[66] Mario Flajslik, James Dinan, and Keith D. Underwood. „Mitigating MPI Message Matching Misery". In: *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*. Ed. by Julian M. Kunkel, Pavan Balaji, and Jack Dongarra. Vol. 9697. Lecture Notes in Computer Science. Springer, 2016, pp. 281–299. DOI: 10.1007/978-3-319-41321-1_15.

[67] Hoang-Vu Dang, Marc Snir, and William Gropp. „Towards millions of communicating threads". In: *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*. Ed. by Jack Dongarra, Daniel Holmes, Antonia B. K. Collis, Jesper Larsson Träff, and Lorna Smith. ACM, 2016, pp. 1–14. DOI: 10.1145/2966884.2966914.

[68] Timothy G. Mattson et al. „The 48-core SCC Processor: the Programmer's View". In: *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. IEEE, 2010, pp. 1–11. DOI: 10.1109/SC.2010.53.

[69] Ernie Chan. *RCCE_comm: A Collective Communication Library for the Intel Single-chip Cloud Computer*. online http://communities.intel.com/servlet/JiveServlet/previewBody/5629-102-1-8718/RCCE_comm.pdf, last accessed 2017-01-10. Sept. 2010.

[70] Carsten Clauss, Stefan Lankes, Thomas Bemmerl, Jacek Galowicz, and Simon Pickartz. *iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer*. Tech. rep. online http://www.lfbs.rwth-aachen.de/publications/files/iRCCE.pdf, last accessed 2017-01-10. Chair for Operating Systems, RWTH Aachen University, Nov. 2011.

[71] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. „RCKMPI - Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC)". In: *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. Ed. by Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra. Vol. 6960. Lecture Notes in Computer Science. Springer, 2011, pp. 208–217. DOI: 10.1007/978-3-642-24449-0_24.

[72] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 1999.

[73] Torsten Hoefler et al. „Remote Memory Access Programming in MPI-3". In: *ACM Transactions on Parallel Computing* 2.2 (2015), 9:1–9:26. DOI: 10.1145/2780584.

[74] Jonathan M. D. Hill et al. „BSPlib: The BSP programming library". In: *Parallel Computing* 24.14 (1998), pp. 1947–1980. DOI: 10.1016/S0167-8191(98)00093-3.

[75] Robert W. Numrich and John Reid. „Co-array Fortran for Parallel Programming". In: *SIGPLAN Fortran Forum* 17.2 (Aug. 1998), pp. 1–31. DOI: 10.1145/289918.289920.

[76] William W. Carlson et al. *Introduction to UPC and language specification*. Tech. rep. CCS-TR-99-157. Second Printing. Center for Computing Sciences, Institute for Defense Analyses, May 1999.

[77] *OpenSHMEM Application Programming Interface*. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf, last accessed 2017-01-13. Feb. 2016.

[78] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. „Parallel Programmability and the Chapel Language". In: *IJHPCA* 21.3 (2007), pp. 291–312. DOI: 10.1177/1094342007078442.

[79] *UPC Language Specifications Version 1.3*. online https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf, last accessed 2017-01-13. UPC Consortium. Nov. 2013.

[80] Dan Bonachea and Jaein Jeong. „GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages". http://www.cs.berkeley.edu/ ~ bonachea/upc/paper.pdf, last accessed 2014-09-30. 2002.

[81] Dan Bonachea. *GASNet Specification*. Version 1.8, Rev 1.44. http://gasnet.lbl.gov/dist/docs/gasnet.pdf, last accessed 2014-09-30. LBNL FTG and U.C. Berkeley. Nov. 2006.

[82] James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. „Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication". In: *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, pp. 739–750. DOI: 10.1109/IPDPS.2012.72.

[83] Jeff R. Hammond, Sayan Ghosh, and Barbara M. Chapman. „Implementing OpenSHMEM Using MPI-3 One-Sided Communication". In: *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - First Workshop, OpenSHMEM 2014, Annapolis, MD, USA, March 4-6, 2014. Proceedings*. Ed. by Stephen W. Poole, Oscar R. Hernandez, and Pavel Shamis. Vol. 8356. Lecture Notes in Computer Science. Springer, 2014, pp. 44–58. DOI: 10.1007/978-3-319-05215-1_4.

[84] Dan Bonachea and Jason Duell. „Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations". In: *International Journal of High Performance Computing and Networking* 1.1/2/3 (2004), pp. 91–99. DOI: 10.1504/IJHPCN.2004.007569.

[85] Martin Bauer, Christian Kuschel, Daniel Ritter, and Klaus Sembritzki. „Comparison of PGAS Languages on a Linked Cell Algorithm". In: *Parallel-Algorithmen und Rechnerstrukturen*. Vol. 30. Mitteilungen. 25th PARS Workshop, Erlangen, Germany: Gesellschaft für Informatik e.V., Sept. 2013, pp. 115–122.

[86] Helmar Burkhart, Madan Sathe, Matthias Christen, Olaf Schenk, and Max Rietmann. „Run, Stencil, Run! – HPC Productivity Studies in the Classroom". In: *The 6th Conference on Partitioned Global Address Space Program-*

ming Models. https://docs.google.com/viewer?a=v&pid=sites&srcid=bGJsLmdvdnxwZ2FzMTJ8Z3g6NWIzZTBhZTI2ZGEzZDRkZQ. 2012.

[87] Martin Ohmann. „Implementation and evaluation of three-dimensional FFT using modern parallel programming APIs". Diploma Thesis. Potsdam: Universität Potsdam, Aug. 2015.

[88] D. A. Mallón et al. „UPC Performance Evaluation on a Multicore System". In: *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*. PGAS '09. Ashburn, Virginia: ACM, 2009, 9:1–9:7. DOI: 10.1145/1809961.1809974.

[89] Cristian Coarfa et al. „An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. Chicago, IL, USA: ACM, 2005, pp. 36–47. DOI: 10.1145/1065944.1065950.

[90] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine A. Yelick. „Optimizing bandwidth limited problems using one-sided communication and overlap". In: *20th International Parallel and Distributed Processing Symposium (IPDPS) 2006, Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. DOI: 10.1109/IPDPS.2006.1639320.

[91] C. M. Maynard. „Comparing UPC and one-sided MPI: A distributed hash table for GAP". In: *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*. online http://pgas11.rice.edu/papers/Maynard-Distributed-Hash-Table-PGAS11.pdf, last accessed 2017-01-01. Oct. 2011.

[92] William M. Tang et al. „Extreme scale plasma turbulence simulations on top supercomputers worldwide". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. Ed. by John West and Cherri M. Pancake. ACM, 2016, p. 43.

[93] Mark Bull, Xu Guo, and Ioannis Liabotis. *Applications and user requirements for Tier-0 systems*. PRACE-1IP deliverable D7.4.1. online http://www.prace-ri.eu/IMG/pdf/D7-4-1.pdf last accessed 2017-01-13. PRACE Consortium Partners, Feb. 2011.

[94] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[95] M. Rasit Eskicioglu. „A Comprehensive Bibliography of Distributed Shared Memory". In: *Operating Systems Review* 30.1 (1996), pp. 71–96. DOI: 10.1145/218646.218651.

[96] Xiaocheng Zhou et al. „A case for software managed coherence in manycore processors". In: *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10*. 2010. URL: http://static.usenix.org/event/hotpar10/tech/full_papers/Zhou.pdf.

[97] *Software-Managed Cache Coherence for SCC*. Revision 1.5. Intel Corperation. Nov. 2011.

[98] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. „A Coherent and Managed Runtime for ML on the SCC". In: *Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, November 29th-30th 2012, Aachen, Germany*. Ed. by Stefan Lankes and Carsten Clauss. RWTH Aachen University, 2012, pp. 20–35. URL: http://nbn-resolving.de/urn:nbn:de:hbz:82-opus-43835.

[99] Kai Li and Paul Hudak. „Memory Coherence in Shared Virtual Memory Systems". In: *ACM Trans. Comput. Syst.* 7.4 (1989), pp. 321–359. DOI: 10.1145/75104.75105.

[100] Randolf Rotta, Thomas Prescher, Jörg Nolte, and Jana Traue. „Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC". In: *6th Many-core Applications Research Community (MARC) Symposium. Proceedings of the 6th MARC Symposium, 19-20 July 2012, Toulouse, France*. Ed. by Eric Noulard. ONERA, The French Aerospace Lab, 2012, pp. 13–18. URL: http://hal.archives-ouvertes.fr/hal-00718993.

[101] Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen. „The Path to MetalSVM: Shared Virtual Memory for the SCC". In: *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*. Ed. by Peter Tröger and Andreas Polze. Universitätsverlag Potsdam, Feb. 2012, 7–14. URL: http://pub.ub.uni-potsdam.de/volltexte/2012/5789/.

[102] King Tin Lam et al. „Rhymes: A shared virtual memory system for non-coherent tiled many-core architectures". In: *20th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2014, Hsinchu, Taiwan, December 16-19, 2014*. IEEE Computer Society, 2014, pp. 183–190. DOI: 10.1109/PADSW.2014.7097807.

[103] Andrew Baumann et al. „The multikernel: a new OS architecture for scalable multicore systems“. In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 29–44. DOI: 10.1145/1629575.1629579.

[104] Junghyun Kim, Sangmin Seo, and Jaejin Lee. „An efficient software shared virtual memory for the single-chip cloud computer“. In: *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*. Ed. by Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou. ACM, 2011, p. 4. DOI: 10.1145/2103799.2103804.

[105] Igor Tartalja and Veljko Milutinovic. *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997.

[106] Hoichi Cheong and Alexander V. Veidenbaum. „The Performance of Software-managed Multiprocessor Caches on Parallel Numerical Programs“. In: *Supercomputing, 1st International Conference, Athens, Greece, June 8-12, 1987, Proceedings*. Ed. by Elias N. Houstis, Theodore S. Papatheodorou, and Constantine D. Polychronopoulos. Vol. 297. Lecture Notes in Computer Science. Springer, 1987, pp. 316–337. DOI: 10.1007/3-540-18991-2_19.

[107] Hoichi Cheong and Alexander V. Veidenbaum. „A Cache Coherence Scheme With Fast Selective Invalidation“. In: *Proceedings of the 15th Annual International Symposium on Computer Architecture. Honolulu, Hawaii, May-June 1988*. Ed. by Howard Jay Siegel. IEEE Computer Society, 1988, pp. 299–307. DOI: 10.1109/ISCA.1988.5240.

[108] A. J. Smith. „CPU Cache Consistency with Software Support and Using 'One-Time Identifiers'“. In: *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*. Ed. by Igor Tartalja and Veljko Milutinovic. IEEE Computer Society Press, 1997, pp. 193–201.

[109] David R. Cheriton, Gert A. Slavenburg, and Patrick D. Boyle. „Software-controlled caches in the VMP multiprocessor“. In: *The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions*. Ed. by Igor Tartalja and Veljko Milutinovic. IEEE Computer Society Press, 1997, pp. 184–192.

[110] Gopalakrishnan Santhanaraman, Tejus Gangadharappa, Sundeep Narravula, Amith R. Mamidala, and Dhabaleswar K. Panda. „Design alternatives for implementing fence synchronization in MPI-2 one-sided communication for InfiniBand clusters“. In: *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana,*

USA. IEEE Computer Society, 2009, pp. 1–9. DOI: 10.1109/CLUSTR.2009.5289200.

[111] Sameer Kumar and Michael Blocksome. „Scalable MPI-3.0 RMA on the Blue Gene/Q Supercomputer". In: *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*. Ed. by Jack Dongarra, Yutaka Ishikawa, and Atsushi Hori. ACM, 2014, p. 7. DOI: 10.1145/2642769.2642778.

[112] Sameer Kumar, Amith R. Mamidala, Philip Heidelberger, Dong Chen, and Daniel Faraj. „Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer". In: *International Journal of High Performance Computing Applications* 28.4 (2014), pp. 450–464. DOI: 10.1177/1094342014552086.

[113] Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. „The Implementation of MPI-2 One-Sided Communication for the NEC SX-5". In: *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*. Ed. by Jed Donnelley. IEEE Computer Society, 2000, p. 1. DOI: 10.1109/SC.2000.10023.

[114] Stephen Booth and Fernando Elson Mourão. „Single sided MPI implementations for SUN MPI". In: *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*. Ed. by Jed Donnelley. IEEE Computer Society, 2000, p. 2. DOI: 10.1109/SC.2000.10022.

[115] Pablo Reble, Stefan Lankes, Carsten Clauss, and Thomas Bemmerl. „A Fast Inter-Kernel Communication and Synchronization layer for MetalSVM". In: *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*. Ed. by Diana Göhringer, Michael Hübner, and Jürgen Becker. KIT Scientific Publishing, Karlsruhe, 2011, pp. 19–23. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937.

[116] Adán Kohler and Martin Radetzki. „Latency-optimized Collectives for High Performance on Intel's Single-chip Cloud Computer". In: *Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, November 29th-30th 2012, Aachen, Germany*. Ed. by Stefan Lankes and Carsten Clauss. RWTH Aachen University, 2012, pp. 7–12. URL: http://nbn-resolving.de/urn:nbn:de:hbz:82-opus-43835.

[117] Hayder Al-Khalissi, Andrea Marongiu, and Mladen Berekovic. „Low-Overhead Barrier Synchronization for OpenMP-like Parallelism on the Single-Chip Cloud Computer". In: *Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, November 29th-30th 2012,*

Aachen, Germany. Ed. by Stefan Lankes and Carsten Clauss. RWTH Aachen University, 2012, pp. 26–31. URL: http://nbn-resolving.de/urn:nbn:de:hbz:82-opus-43835.

[118] Pavan Balaji et al. „MPI on millions of Cores". In: *Parallel Processing Letters* 21.1 (2011), pp. 45–60. DOI: 10.1142/S0129626411000060.

[119] Chaoran Yang, Wesley Bland, John M. Mellor-Crummey, and Pavan Balaji. „Portable, MPI-interoperable coarray fortran". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. Ed. by José E. Moreira and James R. Larus. ACM, 2014, pp. 81–92. DOI: 10.1145/2555243.2555270.

[120] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. „Enabling highly-scalable remote memory access programming with MPI-3 one sided". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*. Ed. by William Gropp and Satoshi Matsuoka. ACM, 2013, 53:1–53:12. DOI: 10.1145/2503210.2503286.

[121] William D. Gropp and Rajeev Thakur. „An Evaluation of Implementation Options for MPI One-Sided Communication". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*. Ed. by Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra. Vol. 3666. Lecture Notes in Computer Science. Springer, 2005, pp. 415–424. DOI: 10.1007/11557265_53.

[122] George Almási et al. „An Overview of the Blue Gene/L System Software Organization". In: *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*. Ed. by Harald Kosch, László Böszörményi, and Hermann Hellwagner. Vol. 2790. Lecture Notes in Computer Science. Springer, 2003, pp. 543–555. DOI: 10.1007/978-3-540-45209-6_79.

[123] William Gropp, Ewing L. Lusk, Nathan E. Doss, and Anthony Skjellum. „A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard". In: *Parallel Computing* 22.6 (1996), pp. 789–828. DOI: 10.1016/0167-8191(96)00024-5.

[124] William Gropp. „MPICH2: A New Start for MPI Implementations". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Proceedings*. Vol. 2474. Lecture Notes in Computer Science. Springer, 2002, p. 7. DOI: 10.1007/3-540-45825-5_5.

[125] Rajeev Thakur, William Gropp, and Brian R. Toonen. „Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication". In: *International Journal of High Performance Computing Applications* 19.2 (2005), pp. 119–128. DOI: 10.1177/1094342005054258.

[126] Rajeev Thakur, William D. Gropp, and Brian R. Toonen. „Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*. Ed. by Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra. Vol. 3241. Lecture Notes in Computer Science. Springer, 2004, pp. 57–67. DOI: 10.1007/978-3-540-30218-6_15.

[127] James Dinan et al. „An implementation and evaluation of the MPI 3.0 one-sided communication interface". In: *Concurrency and Computation: Practice and Experience* (2016). DOI: 10.1002/cpe.3758.

[128] Network-Based Computing Laboratory Ohio State University. *MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE*. online. http://mvapich.cse.ohio-state.edu/.

[129] Ping Lai, Sayantan Sur, and Dhabaleswar K. Panda. „Designing truly one-sided MPI-2 RMA intra-node communication on multi-core systems". In: *Computer Science - R&D* 25.1-2 (2010), pp. 3–14. DOI: 10.1007/s00450-010-0115-3.

[130] Sreeram Potluri, Hao Wang, Vijay Dhanraj, Sayantan Sur, and Dhabaleswar K. Panda. „Optimizing MPI One Sided Communication on Multi-core InfiniBand Clusters Using Shared Memory Backed Windows". In: *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. Ed. by Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra. Vol. 6960. Lecture Notes in Computer Science. Springer, 2011, pp. 99–109. DOI: 10.1007/978-3-642-24449-0_13.

[131] Ohio State University. *OSU Micro-Benchmarks*. Online. http://mvapich.cse.ohio-state.edu/benchmarks/, last accesses 2015-09-14.

[132] Jeff Squyres. „Open MPI". In: *The Architecture of Open Source Applications*. Ed. by Amy Brown and Greg Wilson. Vol. II. http://aosabook.org/en/openmpi.html, last accessed 2015-08-19. 2014.

[133] Brian Barrett, Galen M. Shipman, and Andrew Lumsdaine. „Analysis of Implementation Options for MPI-2 One-Sided". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*. Ed. by

Franck Cappello, Thomas Hérault, and Jack Dongarra. Vol. 4757. Lecture Notes in Computer Science. Springer, 2007, pp. 242–250. DOI: 10.1007/978-3-540-75416-9_35.

[134] Nathan Hjelm. „An Evaluation of the One-Sided Performance in Open MPI". In: *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*. Ed. by Jack Dongarra, Daniel Holmes, Antonia B. K. Collis, Jesper Larsson Träff, and Lorna Smith. ACM, 2016, pp. 184–187. DOI: 10.1145/2966884.2966890. URL: http://doi.acm.org/10.1145/2966884.2966890.

[135] Scalable Parallel Computing Lab. *FoMPI Source Code*. online. http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI/foMPI-0.2.1.tar.gz. Sept. 2013.

[136] Lars Schneidenbach. „The benefits of one-sided communication interfaces for cluster computing". PhD thesis. University of Potsdam, 2009.

[137] Adrian M. Partl, Antonella Maselli, Benedetta Ciardi, Andrea Ferrara, and Volker Müller. „Enabling parallel computing in CRASH". In: *Monthly Notices of the Royal Astronomical Society* 414.1 (2011), p. 428. DOI: 10.1111/j.1365-2966.2011.18401.x.

[138] Roberto Belli and Torsten Hoefler. „Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization". In: *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 2015, pp. 871–881. DOI: 10.1109/IPDPS.2015.30.

[139] Hayder Al-Khalissi, Syed Abbas Ali Shah, and Mladen Berekovic. „An Efficient Barrier Implementation for OpenMP-Like Parallelism on the Intel SCC". In: *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*. IEEE Computer Society, 2014, pp. 76–83. DOI: 10.1109/PDP.2014.25.

[140] Randolf Rotta. „On Efficient Message Passing on the Intel SCC". In: *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*. Ed. by Diana Göhringer, Michael Hübner, and Jürgen Becker. KIT Scientific Publishing, Karlsruhe, 2011, pp. 53–58. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937.

[141] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope. „Efficient Memory Copy Operations on the 48-core Intel SCC Processor". In: *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011*. Ed. by Diana Göhringer, Michael Hübner, and Jürgen Becker. KIT Scientific Publishing, Karlsruhe, 2011, pp. 13–18. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023937.

[142] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. „Scalable Memory Use in MPI: A Case Study with MPICH2". In: *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. Ed. by Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra. Vol. 6960. Lecture Notes in Computer Science. Springer, 2011, pp. 140–149. DOI: 10.1007/978-3-642-24449-0_17.

[143] Werner Augustin, Marc-Oliver Straub, and Thomas Worsch. „Benchmarking One-Sided Communication with SKaMPI 5". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*. Ed. by Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra. Vol. 3666. Lecture Notes in Computer Science. Springer, 2005, pp. 301–308. DOI: 10.1007/11557265_40.

[144] Vinod Tipparaju, William Gropp, Hubert Ritzdorf, Rajeev Thakur, and Jesper Larsson Träff. „Investigating High Performance RMA Interfaces for the MPI-3 Standard". In: *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*. IEEE Computer Society, 2009, pp. 293–300. DOI: 10.1109/ICPP.2009.54.

[145] Isaías A. Comprés Ureña and Michael Gerndt. „Improved RCKMPI's SCC-CMPB Channel: Scaling and Dynamic Processes Support". In: *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*. Ed. by Peter Tröger and Andreas Polze. Universitätsverlag Potsdam, Feb. 2012, 1–6. URL: http://pub.ub.uni-potsdam.de/volltexte/2012/5789/.

[146] Carsten Clauss, Stefan Lankes, and Thomas Bemmerl. „Performance Tuning of SCC-MPICH by Means of the Proposed MPI-3.0 Tool Interface". In: *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. Ed. by Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra. Vol. 6960. Lecture Notes in Computer Science. Springer, 2011, pp. 318–320. DOI: 10.1007/978-3-642-24449-0_37.

[147] Gregory F. Pfister et al. „The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture“. In: *International Conference on Parallel Processing, ICPP'85, University Park, PA, USA, August 1985*. IEEE Computer Society Press, 1985, pp. 764–771.

[148] Mats Brorsson. „Local vs. global memory in the IBM RP3: experiments and performance modelling“. In: *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, SPDP 1991, 2-5 December 1991, Dallas, Texas, USA*. IEEE Computer Society, 1991, pp. 496–503. DOI: 10.1109/SPDP.1991.218258.

[149] Tilman Butz. *Fouriertransformation für Fußgänger*. 7. Auflage. Wiesbaden: Vieweg+Teubner Verlag, 2012. DOI: 10.1007/978-3-8348-8295-0.

[150] David H. Bailey et al. „The NAS Parallel Benchmarks“. In: *International Journal of High Performance Computing Applications* 5.3 (1991), pp. 63–73. DOI: 10.1177/109434209100500306.

[151] Rajesh Nishtala, Paul Hargrove, Dan Bonachea, and Katherine A. Yelick. „Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap“. In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161076.

[152] Peter Sanders and Thomas Worsch. *Parallele Programmierung mit MPI - ein Praktikum, Programmtexte im Internet*. Logos Verlag, 1997.

[153] *Power ISA Version 2.07*. online http://fileadmin.cs.lth.se/cs/education/EDAN25/PowerISA_V2.07_PUBLIC.pdf, last accessed 2016-12-29. IBM. May 2013.

[154] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volume 2: Instruction Set Reference, A-Z. online https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf, last accessed 2016-12-29. Intel Corperation. Sept. 2016.

Author's Declaration

I hereby declare that this thesis is the result of my own investigations and research work, except where specific references are made to publications of others. Work done in collaboration with, or with the assistance of others, is indicated as such. This thesis has neither been accepted, rejected nor concurrently submitted in candidature for any other degree to any other faculty or university.

Potsdam, February 1, 2017

Steffen Christgau