MPI-3 algorithms for 3D radiative transfer on Intel Xeon Phi coprocessors

– Masterarbeit –

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Jannek Squar
E-Mail-Adresse:	1squar@informatik.uni-hamburg.de
Matrikelnummer:	6357990
Studiengang:	Informatik
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Prof. Dr. Peter Hauschildt
Betreuer:	Dr. Michael Kuhn

Hamburg, den 18.5.2017

Abstract

One-sided communication has been added to the MPI standard with MPI-2 in 1997 and has been greatly extended with the introduction of MPI-3 in 2012. Even though one-sided communication offers many use cases, from which an application could benefit, it has only sporadically been used for HPC so far. The objective of this thesis is to examine its potential use for replacing a OpenMP section with equivalent code, which only makes use of MPI. This is done based on an already existing application, named `PHOENIX`. This application is currently developed at the observatory of Hamburg and has been designed to be executed on HPC systems. Its purpose is, among other things, to numerically solve the equations of 3D radiative transfer for stellar objects. For utilising HPC hardware at its full capacity `PHOENIX` makes use of MPI and OpenMP. In the course of this thesis a test application has been constructed, which mimics the OpenMP sections and allows to benchmark diverse combinations of MPI one-sided communication operations. The benchmarks are performed on a Intel Xeon Phi Knights Corner and on a Intel Xeon Phi Knights Landing to estimate if a certain approach is suitable for HPC hardware in general. In the end each approach is discussed and assessed which kind of communication pattern might benefit most of MPI one-sided communication.

"The only good bug is a dead bug"

– Starship Troopers (1997)

Contents

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Structure . . . 10

2 MPI **11**
2.1 One-sided communication . . . 11
2.1.1 RDMA . . . 12
2.1.2 Memory consistency model . . . 14
2.1.3 Shared Memory . . . 17
2.1.4 Use cases . . . 20
2.2 Standards . . . 23
2.2.1 MPI-2.0 . . . 23
2.2.2 MPI-3.0 . . . 26
2.2.3 Implementations . . . 27
2.3 Available operations . . . 28
2.3.1 Initialisation . . . 29
2.3.2 Communication . . . 32
2.3.3 Synchronisation . . . 34
2.3.4 Synchronisation application . . . 39

3 Phoenix **41**
3.1 3D radiative transfer . . . 41
3.1.1 Theory . . . 41
3.1.2 Implementation in Phoenix . . . 43
3.2 Program flow . . . 45
3.3 Parallelisation pattern . . . 46
3.3.1 3D radiative transfer . . . 49
3.4 Profiling . . . 50
3.5 Applying MPI . . . 52

4 Intel Xeon Phi **53**
4.1 Overview . . . 54
4.2 Knights Landing . . . 54
4.2.1 Architecture . . . 55
4.3 Xeon Phi cluster . . . 57
4.4 Suitability for Phoenix . . . 58

5 Using MPI-3 RMA **60**
5.1 TrackerSim . . . 61
5.1.1 Variation 0: SMP . . . 63
5.1.2 Variation 1: fence-synchronisation . . . 65
5.1.3 Variation 2: passive synchronisation . . . 67
5.1.4 Variation 3: shared memory with RMA communication . . . 69
5.1.5 Variation 4: shared memory with local communication . . . 71
5.1.6 Concluding discussion . . . 73
5.2 Design approach . . . 74
5.3 Substitution of OpenMP . . . 77

6 Evaluation **79**
6.1 Tools . . . 79
6.2 TrackerSim . . . 80
6.2.1 Comparison of OpenMP and MPI-3 RMA . . . 81
6.2.2 Scaling . . . 83
6.3 Phoenix . . . 88
6.3.1 Comparison of OpenMP and MPI-3 RMA . . . 88
6.4 Evaluation . . . 89
6.4.1 Runtime . . . 90
6.4.2 Memory usage . . . 90

7 Conclusion **92**
7.1 Discussion . . . 92
7.2 Related work . . . 93
7.3 Future work . . . 93

Bibliography **95**

Appendices **99**

List of Figures **100**

List of Listings **103**

List of Tables **104**

1 Introduction

1.1 Motivation

Computational science describes the use of computers to simulate circumstances, which are otherwise too expensive or even impossible to be reproduced through experiments. Parallel computers have become a very popular tool to perform these simulations, because it's more difficult and costly (or even impossible) to achieve the same computing capacity with a single machine than with a parallel computer architecture. Several parallel computational models exist to exploit parallel computers, they differ in using shared or distributed memory, message passing, vector processors, threads, and so forth. Each computational model can be implemented on any hardware, its effectiveness however will vary as there exists for every model a most suitable hardware [GLS14].

Remote Memory Access (RMA, cf. section **RDMA** (p.12)) describes a programming model, which allows to access or update memory remotely (ideally without any corresponding action of the remote target). One implementation of this programming model is MPI one-sided communication. It has been first included into the MPI standard in July, 1997. Still one-sided communication has not attracted a large community. One reason for this situation is the fact, that using MPI one-sided communication quickly becomes very tedious and several use cases of MPI one-sided communication can be implemented with other techniques, which are easier to use and well-established. This is shown by [aMT06], whose authors tried to implement algorithms, which make use of OpenMP, with MPI-2 one-sided communication instead. Even though the substitution was possible the results did not suggest to imitate this procedure. Chapter **Using MPI-3 RMA** (p.60) will discuss this procedure again, but this time MPI-3 one-sided communication is used.

If at all, MPI one-sided communication is used especially for a subdomain of a HPC-application, since in HPC the effort could produce a valuable difference in speedup. The Earth Simulator topped the Top500 list awhile [TOP] and, therefore, one may assume that major effort is put in to optimise any application, which shall be executed on it - an example for such an application is presented in [YIU+02]. Nevertheless even for this application MPI one-sided communication has only be used for a section of the application to manage the transfer of data.

In 2012 the MPI standard expanded the one-sided communication interface by new operations; amongst others it is now possible to make use of shared memory. One major intention of this thesis is to investigate the potential of the improvements of MPI-3 one-sided communication.

`PHOENIX` is an application to solve 3D radiative transfer and is used at the observatory of Hamburg to compute numerically stellar atmospheres. In general it scales very well

with additional threads; PHOENIX is an example for a HPC application and is worthwhile to be deployed on HPC systems. So it is suitable to be used as a test environment for changes made with MPI-3 one-sided communication. The main objective of this thesis is to examine the improvements of MPI-3 one-sided communication and then analyse its potential regarding one special use case: How can MPI-3 one-sided communication be used to replace an OpenMP section with an equivalent MPI-only section on the one hand and to evaluate the resulting performance on the other hand. The changes may then be tested and evaluated by executing PHOENIX respectively an adapted test program on an Intel Xeon Phi, which has been developed to support HPC. This gives an intuition if MPI-3 one-sided communication is an eligible candidate to be used on HPC systems.

This year, Intel released the second generation of the Xeon Phi with the intention to approach the most important issues, which Knights Corner had revealed. The new generation of the Xeon Phi is called Knights Landing and is available as a coprocessor and as a processor as well. Again it offers many cores together with vector processing units to offer hight performance if an application is capable of using the Xeon Phi at its full capacity. A Knights Corner Coprocessor as well as a Knights Landing Processor were used for this thesis.

1.2 Structure

Chapter **MPI** (p.11) gives an overview about new operations, which have been introduced into the MPI standard with MPI-2 and MPI-3. For the sake of brevity only those operations are presented, which are an element of one-sided communication or are necessary to substitute OpenMP. The HPC application PHOENIX is discussed in chapter **Phoenix** (p.41) to get clear, what its parallelisation pattern looks like and how this could be used to gain performance through the use of MPI one-sided communication. Besides a common Linux-PC a Intel Xeon Phi Knights Corner coprocessor and a Intel Xeon Phi Knights Landing processor are used to benchmark several combinations of MPI-3 one-sided communication operations, therefore, the Xeon Phi is presented in chapter **Intel Xeon Phi** (p.53). After providing these basic informations chapter **Using MPI-3 RMA** (p.60) brings MPI one-sided combination, Intel Xeon Phi and PHOENIX together to discuss several possibilities to replace OpenMP with MPI one-sided communication. The evaluation of the suitability of these approaches is then performed in chapter **Evaluation** (p.79). The last chapter **Conclusion** (p.92) summarises the results of the prior chapters and displays related and future work.

2 MPI

MPI (acronym for *Message Passing Interface*) is an interface designed by the MPI Forum [MPIb], which defines the syntax and semantics of operations for message passing between processes, which are able to communicate with each other. These operations cover point-to-point-communication as well as collective communication, one-sided communication, managing processes, debugging, and more. Processes, which communicate with each other, do not need to share the same memory and may rely on local memory - any kind of connection via a network is sufficient. Therefore it has become the preferred standard for machines with distributed memory and in the field of high performance computing in general [VVH+15]. To encourage usage and dissemination of MPI, the MPI Forum laid focus on the portability and decided to include many features of existing message passing systems. Nevertheless it is possible to build applications with just six MPI-functions, which can already use distributed memory to full capacity: `MPI_Init`, `MPI_Finalize`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Send`, `MPI_Recv`.

The MPI standard itself does not include an implementation but there are implementations for a broad variety of hardware made by companies and research communities - see section **Implementations** (p.27) for a selection of established implementations.

The first version of MPI was published in May 1994 and is since then in constant development. MPI-2.0 was released in July 1997 and MPI-3.0 in September 2012. The current release of the MPI standard is MPI-3.1, released in June 2015. There are already working groups assigned to topics, which might be considered for MPI-4 [MPIa]. Each version of MPI comes along with its own documentation *MPI: A Message-Passing Interface Standard*, which are available at [MPIb]. For an introduction into using MPI refer to *Using MPI* [GLS14] or *Using Advanced MPI* [GHTL14].

From now on MPI-X refers to any MPI-X.y subversion whereas MPI-X.y is used to reference to the specific MPI-X.y version.

2.1 One-sided communication

There are two possibilities how two or more processes can communicate with each other: Either through message passing, which requires, that every involved process in this communication participates actively, or through direct memory access. Operations for direct memory access had been added to MPI with the introduction of the MPI-2.0 standard, where they had been specified as one-sided communication operations. The expression *one-sided communication* already implies the main feature of this approach: one process (the so-called *origin*) puts or gets data directly to or from another process' memory (the *target*), without the need, that the target process calls an equivalent

operation. *Origin* denotes always the process, which invokes a communication call, *target* is always the process, whose memory is accessed or updated. Also `origin == target` is explicitly allowed by the MPI standard. Hereafter RMA communication operations, which might change remote memory (like `MPI_PUT` or `MPI_ACCUMULATE`), will be just called *update*; RMA communication operations, which only get a value from a remote memory without any change (like `MPI_GET`), will be called *access*.

The major advantage of one-sided communication is, that the target's memory can be accessed and updated directly without the collaboration of the target. Using two-sided communication, the target would need to react actively to the origin. This would be e.g. calling `MPI_Recv(...)` for the corresponding `MPI_Send(...)`. But if the process, which is the target of `MPI_Send(...)`, does not respond to this call, the message propagation would never complete. Using one-sided communication the target process is not involved in calling communication operations, therefore communication operations are safe from deadlocks. There are still some one-sided operations, which require cooperation of processes, but they are limited to initialisation and synchronisation operations[1].

In contrast to the fact, that today MPI is omnipresent on high performance systems, MPI-2 one-sided communication is barely used, even though it had been published more than a decade ago. Noteworthy applications like [YIU+02], which uses one-sided communication to transfer data for transposition, are an exception [TGR+09, 1]. This was one reason the MPI Forum released MPI-3 with major changes to one-sided communication, which later will be demonstrated in chapter **Using MPI-3 RMA** (p.60). In addition there are many use cases where one-sided communication is not the only solution but a well fitting one. See section **Use cases** (p.20) to get an impression, what one-sided communication could be used for.

2.1.1 RDMA

Remote memory access (RMA) describes a programming model, which allows a process to access or update another process' memory remotely. Before RMA had been introduced to MPI, a well-established possibility to access or update distributed memory were two-sided MPI communication operations. MPI one-sided communication operations allow to use the potential of RMA and to distinguish between remote and local memory. In addition it enables that processes which share the same memory, i.e. they run on the same node, may execute load and store operations on each other's local memory directly, i.e., without passing messages. RMA and MPI one-sided communication are used synonymously in this thesis.

Sometimes RMA is noted in combination with *Partitioned Global Address Space* (PGAS) languages, which also allows to access and update shared or remote memory. PGAS creates on a machine one virtual shared memory which is accessible by every process - sections of this memory space may also be assigned to a specific process to

[1] Because of the need for collaboration, deadlocks are possible to occur during window creation and synchronisation

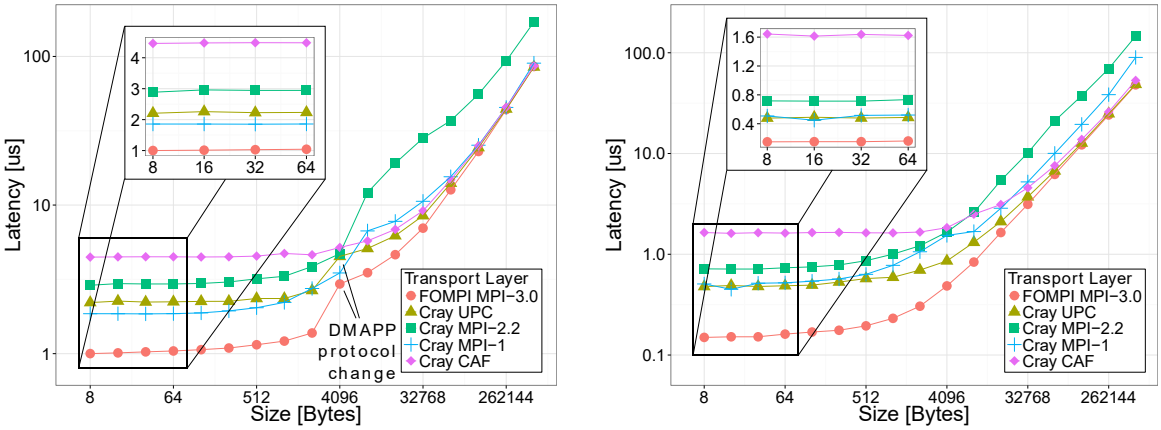
reduce access time. It is not relevant, if the machine really features one large shared memory, a PGAS language will still create the illusion of one large shared memory. Even though MPI implementations are usually tuned for performance, there are some use cases, which are better fulfilled by a PGAS language. This was primarily the case, when MPI-2 was the current MPI standard. MPI-3 faced several disadvantages compared with PGAS, however some are still remaining (for example the constraint that every process, which wishes to access or update shared memory, must take part in the collective window creation). Well-known PGAS languages are for example *Unified Parallel C* and *Coarray Fortran* [BD04][Pad11].

RMA only describes a concept for shared memory, *Remote Direct Memory Access* (RDMA) in contrast is a mechanism to perform RMA operations. RDMA does not rely on a corresponding action by neither the target process nor the operating system on the target node (respectively to the actual data transfer) and offers a high bandwidth and low latency at once. In contrast to RMA and PGAS, which are designed for end users, RDMA is a low-level mechanism. RMA may use RDMA to implement its functions, which are then executed on a hardware level (if the hardware offers support for RDMA). Most interconnection hardware allows an efficient execution of RDMA: InfiniBand, Blue Gene and Ethernet[1] are just some of them [BH14].

To give a rough impression of the efficiency of interconnection hardware, which supports RDMA, the authors of [GBH13] built a MPI-3 implementation (*foMPI*: fast one-sided MPI) for Cray Gemini and Aries systems based on a low-level RDMA API to investigate its potential. Using optimisations like relinquishing remote buffering for synchronisation and mapping RMA communication operations almost unmodified on RDMA functions result in a performance comparable to Unified Parallel C and Coarray Fortran while being able to scale with millions of cores with good performance. An example to support this statement is shown in figure 2.1: Especially for a small message size the authors' MPI-3 RMA implementation outperforms common MPI implementations as well as Unified Parallel C and Coarray Fortran regarding the latency of an internode put operation (the authors found a similar trend for internode get operations even though it was less apparent); the same tendency is observable while looking at the intranode latency of those operations. Another evidence is given with figure 2.2, where irrespective of the location of the target process (intranode or internode) the latency of foMPI's synchronisation with GATS operations was smaller than those of Cray MPI.

Another paper [Bal04] compared the performance of InfiniBand and 10-Gigabit Ethernet (at the time, when this paper has been released in 2004, both technologies offered the same theoretical bandwidth [Mel06]) and discovered that InfiniBand outperformed Ethernet in relation to actual bandwidth and memory requirements, because InfiniBand is other than Ethernet (if RoCE or iWARP are not used) capable of using RDMA. This advantage becomes observable in figure 2.3a and 2.3b, which compare the effective latency and bandwidth between Ethernet and InfiniBand; the usage of RDMA provides InfiniBand with an advance in performance in both cases. Another advantage

[1] Only if RoCE (RDMA over Converged Ethernet) or iWARP (internet Wide Area RDMA Protocol) are supported



(a) Internode put

(b) Intranode put/get

Figure 2.1: Latency of RMA communication operations

becomes evident in figures 2.3c and 2.3d: Communicating over Ethernet, which depends on TCP/IP communication, comes with a massive overhead regarding memory and CPU usage - using the RDMA interface InfiniBand may perform these operations more resource-conservingly.

2.1.2 Memory consistency model

One major difficulty of parallel applications is synchronizing communication with messages to guarantee a well-defined outcome, which in most cases must also be reproducible. As soon as a process starts to exchange messages with another process, the order of execution is non-deterministic as long as no synchronisation (e.g. with `MPI_BARRIER(...)` or blocking communication) is used. This is due to the circumstance that a node's timing is unpredictable: the execution of instructions could be delayed because of background-processes, network load, buffering or similar. Even if both involved processes are executed on the same core, the actual execution order is non-deterministic because of the scheduler, which guides the actual execution order in a unpredictable manner.

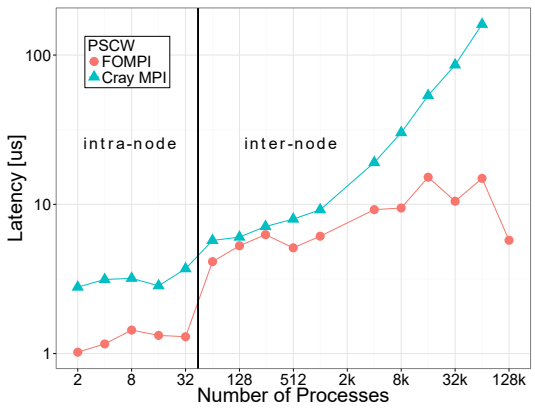
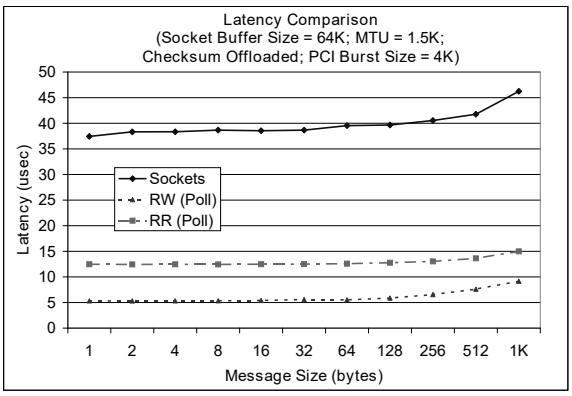
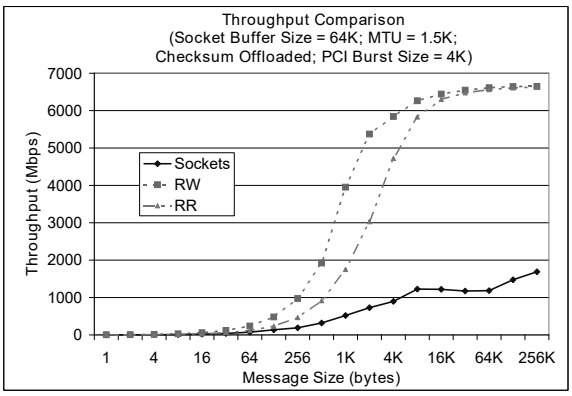


Figure 2.2: Latency of GATS synchronisation

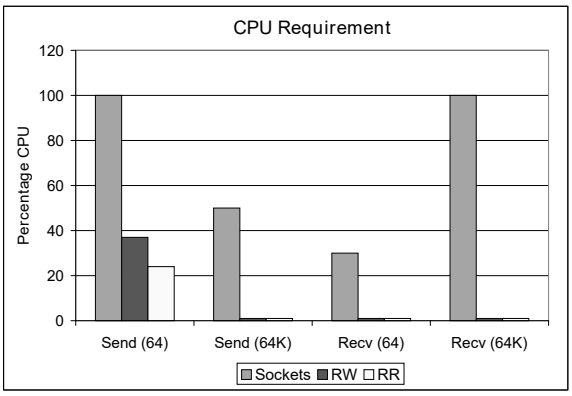
To consider this situation a memory consistency model describes the effect of processes or threads, which operate on the same memory. For demonstration purpose let process A expose some of its memory through a window to allow other processes B_i to access



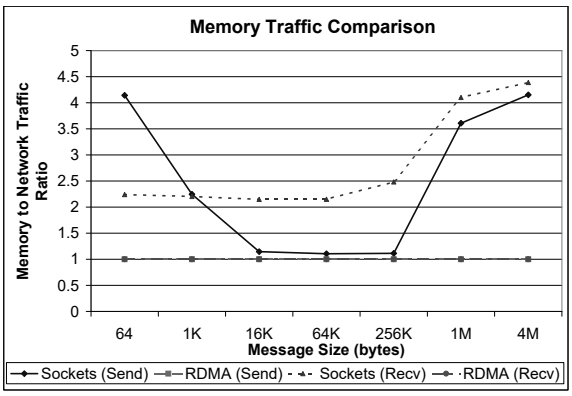
(a) Latency



(b) Throughput



(c) CPU requirement



(d) Memory-network-traffic ratio

Figure 2.3: Comparison between 10-Gigabit Ethernet and InfiniBand regarding performance features

and perform changes on it. The memory model of MPI RMA, which defines the outcome and effects of operations on A's window, introduces the concept of *public* and *private* memory. Both reference the same memory and are characterised as associated window copies, which A exposed through its window:

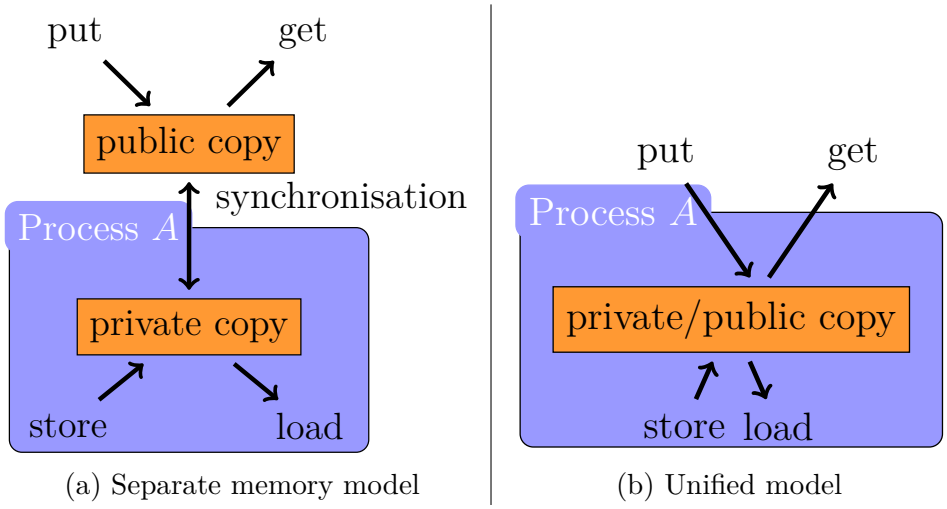
The public copy is accessible by all processes, which took part in the collective window creation. To get/put data from/to the public copy a Process needs to invoke RMA operations - both A and B_i. To guarantee that changes are assigned to the private copy and A's memory is consistent again, RMA synchronisation needs to be used. The private copy belongs to process A[1], and is accessible by local load/store operations.

MPI defines two different memory models, they differ in the fact if the public and private copy are distinct or not.

Separate: This model was introduced in MPI-2 first and was also the only one. In this model the public and private copy may be distinct[2]. Every RMA operation is

[1]The private copy could for example be stored in A's buffer for fast access

[2]MPI-2 does not state that the copies must be distinct



(a) Separate memory model

(b) Unified model

Figure 2.4: RMA memory models based on [HDT+15, 8]

executed on the public copy (regardless of whether A or B_i invoked this operation), any other operation (e.g. local store by A or `MPI_SEND(...)` by B_i) is executed on the private copy. RMA synchronisation operations synchronise the public and private copy to restore the memory consistency. If various operations performed changes on the public and private copy between two synchronisation calls, the content of the memory after the synchronisation is undefined[1]. Figure 2.4a illustrates a window with distinct copies and operations on different copies.

Unified: MPI-3.0 introduced the unified memory model. This model assumes that the public and private copy are identical. Changes to the public copy are eventually propagated to the private copy and vice versa. Therefore less synchronization calls are needed than in the separate model - they are still necessary to guarantee that the changes are visible to get or load operations. Figure 2.4b displays the unified memory model with identical copies.

One needs to distinguish operating on the same window from operating on the same memory: Accessing or updating a local window, does not necessarily affect the associated local memory immediately, whereas a load or store operation on the same local window is executed on the associated memory instantly. The actual behaviour depends on the used memory model, the tables 2.1a and 2.1b displays for both memory models, which combinations of access and update operations may be executed concurrently on the same local window.

OV: Without any constraint

BOV: Without any constraint if operations are executed on single bytes

[1] If MPI-3 is used - if not, these concurrent calls on the same memory would be stated by MPI-2 as erroneous

	Load	Store	Get	Put	Acc
Load	OV+NOV	OV+NOV	OV+NOV	NOV	NOV
Store	OV+NOV	OV+NOV	NOV	X	X
Get	OV+NOV	NOV	OV+NOV	NOV	NOV
Put	NOV	X	NOV	NOV	NOV
Acc	NOV	X	NOV	NOV	OV+NOV

(a) Separate memory model

	Load	Store	Get	Put	Acc
Load	OV+NOV	OV+NOV	OV+NOV	NOV+BOV	NOV+BOV
Store	OV+NOV	OV+NOV	NOV	NOV	NOV
Get	OV+NOV	NOV	OV+NOV	NOV	NOV
Put	NOV+BOV	NOV	NOV	NOV	NOV
Acc	NOV+BOV	NOV	NOV	NOV	OV+NOV

(b) Unified memory model

Table 2.1: RMA memory models based on based on [DBB+16]

NOV: Only if executed on non-overlapping memory

X: Not allowed

The unified model allows to combine local store and remote put operations on non-overlapping memory and eases the restrictions on combinations of local load and remote put operations.

To allow the usage of the unified memory model, the underlying hardware must be able to perform the update propagation without synchronisation calls. Otherwise only the separate model is available. From this it follows that additional effort must be made if the application shall use the unified model and at the same time retain its portability, so it can be executed on machines without hardware-managed coherence, too. Looking at the `MPI_WIN_MODEL` attribute[1] allows the user to use conditional statements for choosing the right amount of synchronisation calls. Currently every interconnection network, which uses RDMA, supports the unified memory model [GBH13, 2]. In case of doubt, choosing synchronisation calls to meet requirements of the separate memory model is always correct.

2.1.3 Shared Memory

In accordance with Flynn's taxonomy most systems for *HPC*[2] fall in the category *MIMD*[3] or are a hybrid of MIMD and *SIMD*[4] (Most processors own a vector unit or a cluster

[1] Possible values: {`MPI_WIN_UNIFIED`,`MPI_WIN_SEPARATE`}
[2] high performance computing
[3] Multiple instructions, multiple data
[4] Single instruction, multiple data

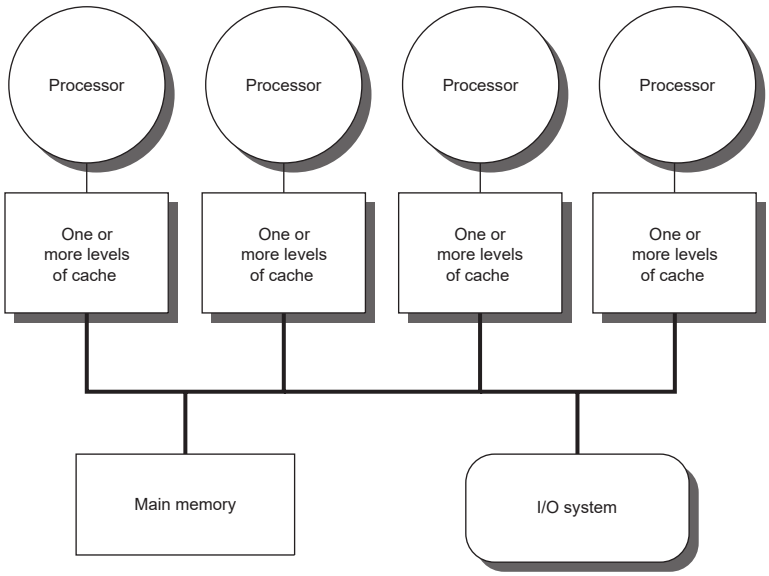


Figure 2.5: UMA architecture[HP06]

contains some accelerators). One important class of a MIMD system is a cluster, which is a composition of nodes, which for their part are a composition of processors. Next, HPC systems can be classified by how memory is organized: Is there only one central memory or several distributed ones? Are the processors interconnected via a fast bussystem or a rather slower network?

Small systems have one shared memory and the processes are connected through a bus (like a ordinary desktop PC or a node in a cluster), the amount of contained processors is not adjustable at will. Figure 2.5 shows the basic structure of such a system. The same memory address points to the same memory location irrespective of which processor dereferences the address, so every process in this figure has access to the same *shared memory*. To access or update data in shared memory every processor simply uses references or assignments. Depending on the basis of the used memory model (cf. **Memory consistency model** (p.14)) changes via assignments become visible for every other processor sooner or later automatically.

Shared memory cannot be used with an arbitrary number of processors because of the increasing demand of bandwidth through the rise in intranode communication. In order to construct big clusters the memory needs to be distributed among the processors - a common use case is to connect nodes of such a big cluster through a interconnection network whereupon each node has its own memory. Regarding to one node, its processors share this memory, regarding to the whole cluster the memory is distributed. Figure 2.6 shows an example for such a cluster, instead of a single processor a node with several processors would be a common alternative. If a process needs to access or update data, which lays on any memory but not its own, passing messages becomes inevitable. It is possible to simulate shared memory based on distributed memory if shared memory is characterized as memory, which can be accessed or updated without the need that any other process except the executing one participates. Then a process could use RMA

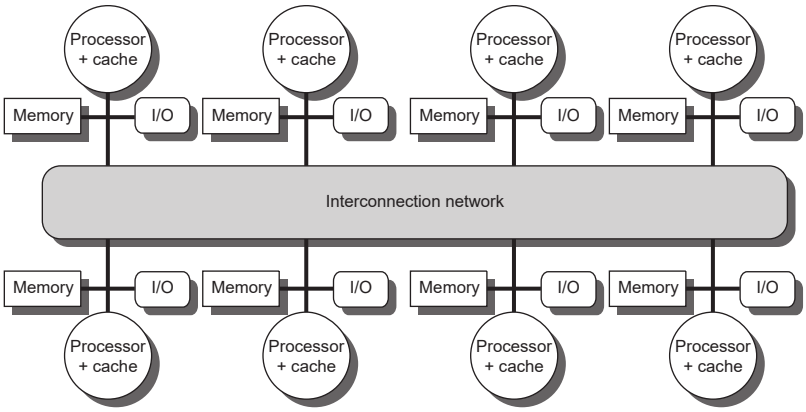
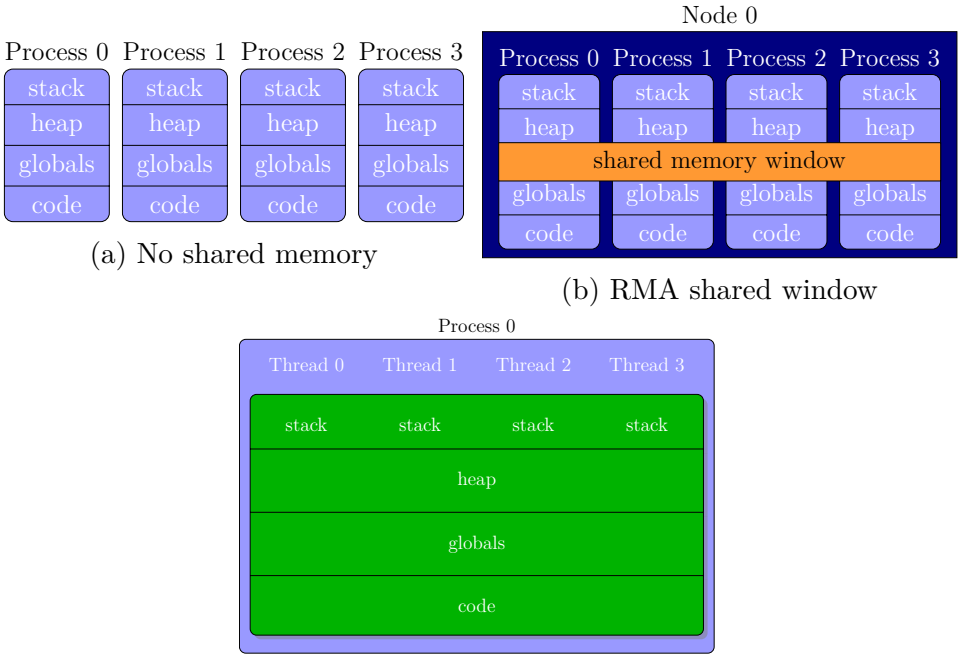


Figure 2.6: Cluster architecture with distributed memory[HP06]

operations with passive synchronisation to access or update data, which is located on another memory. But since this access style still relies on MPI operations this type of memory usage should be called *global memory* [GLS14, Ch. 7].

The availability of shared memory suggests to use rather threads than message passing: Even though it is possible to use MPI to spawn processes and then access/update data on shared memory through two-sided communication, this is normally advised against due to its involved overhead in comparison to threads spawned by e.g. OpenMP or Pthreads. A chance to lessen the overhead of messages is to use one-sided communication: every variable or derived datatype which needs to be shared among processes could be made available by creating a proper shared window with `MPI_WIN_ALLOCATE_SHARED`, followed by assigning the variable's or derived datatypes's value. Figure 2.7 shows all three different kinds of shared memory respectively possibilities to use it: none, if the memory is distributed; a fraction of the whole memory, if a shared RMA window is used; full, if threads are used or a shared RMA window is created with maximum size.

To simulate shared memory as accurately as possible (i.e. no required involvement of other processes in access/update operations) one should use passive synchronisation. The advantage of this approach over using threads is the expandability: if no local store/load operations but only put/get operations are used, it does not matter if the memory is shared or distributed, because MPI communication is not limited to intranode communication but also capable of internode communication. However this renunciation of local operations comes with the price of increased overhead. To improve performance local store/load operations should be used instead of RMA operations whenever it is possible - but then the affected data must be on the same shared memory. **TrackerSim** (p.61) shows different possibilities to replace the use of threads on shared memory by the use of processes. **Substitution of OpenMP** (p.77) shows an approach to alter an application, which relies on a MPI-OpenMP-hybrid into a MPI-only programming. In addition a suggestion for handling complex derived datatypes, which contain pointers, is made.



(a) No shared memory

(b) RMA shared window

(c) Complete shared memory

Figure 2.7: Three kinds of shared memory based on [GHTL14, 158–159]

2.1.4 Use cases

Although the distribution rate of applications, which use one-sided communication, may suggest otherwise, there are several situations where one-sided communication should be favoured over common two-sided communication. One great advantage of one-sided communication is that it includes operations for using shared memory. But one-sided communication provides advantages irrespective of shared memory, too:

- Reduced redundancy:
  Running an application in parallel with n processes[1] results in n copies of this application, which are executed concurrently. The processes communicate with each other by the use of MPI and have typically all the same variables stored in their local memory[2]. Occasionally some variable copies are redundant[3], by what more memory is used as required. Even if the processes are distributed among multiple nodes (without one large shared memory) this redundancy could be prevented, if just one process has these variables and all the rest of the processes access them through one-sided communication. To reduce redundancy is an important topic in HPC, because trends were observed in recent years that both memory capacity and network performance can not keep pace with the progress of the number of cores

[1] Executed with `mpirun -np n ./application`

[2] Data is replicated n times by default

[3] This would be the case, if the variable is the same for every process (e.g. a constant) or changes on every process equally

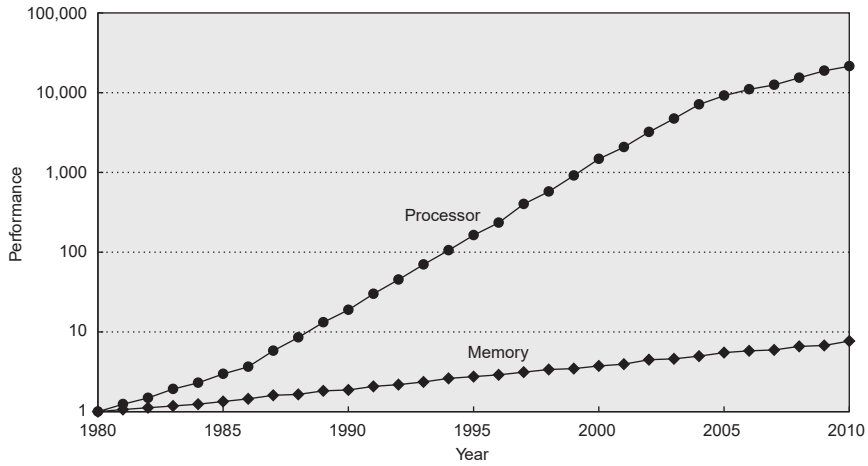


Figure 2.8: Development of performance of hardware over time. Ratio is set to 1 in 1980 [HP06]

per node [HDB+13]. Figure 2.8 shows also the disproportion of the development of the performance of both processors and memory. The ratio of both components is arbitrarily set to 1 in 1980, the processor's performance outdistances the memory's performance in the following years. Because of the decreasing ratio of memory to processing power[1] , adding more cores does not lead to a proportional problem size that can be computed with this architecture. Less redundancy extenuates this disadvantage, an example will be given later in chapter **Phoenix** (p.41).

- Performance:
  There are two different kinds of communication between two processes regarding their node: intranode (origin and target of an operation are on the same node) and internode (origin and target are on different nodes, but still connected) communication. Intranode communication is discussed in **Shared Memory** (p.17). [GT07] investigated the potential of one-sided communication to replace point-to-point communication and focused on the resulting performance. The authors of this paper compared the runtimes of a benchmark, which mimics a common halo exchange, using `MPI_Isend` and `MPI_Irecv` on the one hand and using `MPI_Win_Put` and `MPI_Win_Get` on the other hand. They showed that this substitution may provide an improvement regarding performance. Depending on the test environment (hardware, buffer size, number of RMA communication operations during one epoch) RMA one-sided communication was up to ten times faster (even though the common two-sided communication outperformed RMA for some test cases).

- Homogeneous program model:
  Many applications use MPI basically for distributing data among nodes. Because scientific applications often use large data structures, those applications use then

[1] measured in $[\frac{Byte}{Hz}]$

OpenMP to share those structures on the node's shared memory and perform calculations through OpenMP threads. This leads to a well-established two-stage parallelisation, which does not fully utilise MPI's potential. An alternative is to omit using threads and to pursue the approach, which is discussed in the prior item *Reduced Redundancy*. One more advantage of replacing threads with a processes-only approach is the reduced risk of memory corruption due to program bugs: If threads are used, they share the whole memory (even if parts of it do not need to be shared) - therefore bugs, which corrupt memory, will affect every thread's memory definitely. Using MPI requires to declare shared memory explicitly and might isolate the effects of one process' faulty behaviour [HDB+13].

- Division of labour:
  One possibility to improve performance is to reduce overhead due to essential synchronisation or locks on critical sections. A solution might be to divide the processes' scope of duties so that just one process needs to access a critical section. An example for this case is a group of processes which have alternating phases of computation and writing results into storage (characteristic of bulk synchronous parallel programs [GHTL14, Ch. 3]. One process could be selected for writing the results to storage, which it gets from the other processes through one-sided communication (using passive target communication the other processes do not need to participate in this activity and may continue their computations). The other processes just compute their results and store them into their local memory, which is part of a window. Thereby there are no conflicts anymore in updating the storage. This is just a an example of feasibility to use RMA, usually MPI-IO is more suitable to write data concurrently to storage, because using only one process for writing to storage easily creates a bottleneck.

- Dynamically changing data access patterns:
  To implement dynamic access patterns additional effort is necessary, because the processes must match their operation calls (both the origin and target process need to call corresponding communication operations eventually). This additional effort is rendered unnecessary, if one-sided communication is used and each process can compute which data it needs to access or update - one process alone is responsible for the message exchange then. In case that passive synchronisation is used, the target process is not involved in synchronisation either. The only situation, where processes must act together definitely, is during the initial window creation, which involves a collective call.

Every use case includes the use of one-sided communication and could be implemented solely with operations, which had been introduced with MPI-2. But as long as the one-sided communication is limited to one node with shared memory, every use case could be improved by using MPI-3 extensions [HDB+13] to create shared memory windows, on which all corresponding processes can operate. This allows to use standard load and store operations to dispose of the overhead: instead of transmitting data explicitly through expensive messages, a process can access remote memory directly through intranode

communication [GHTL14, Ch. 5.1.2]. In section **TrackerSim** (p.61) is a comparison between put/get operations and local store/load operations, which shows this overhead. [HDT+13, Ch. 6] gives three more popular use cases, which are more application-oriented.

2.2 Standards

This section is an overview of the changes which were introduced within MPI-2.0 and MPI-3.0. The focus in this chapter lays on changes which have some significance for the topic of this thesis. Refer to [MPI12] for a complete list of changes.

2.2.1 MPI-2.0

Along with corrections and extensions to MPI-1.1 new functionality was added with MPI-2.0 such as one-sided communications and dynamic process creation.

The operations for dynamic process creation made it possible, to spawn new processes during runtime dynamically instead of executing a program with a fixed number of processes. One use case is the implementation of load balancing, especially if the program is executed on a heterogeneous, distributed architecture or uncertain data volume, on which the program performs calculations. In the first case a heterogeneous architecture impedes load balance as it makes additional calculation necessary, which relies on the estimated performance of each architecture component. This could differ from theory during execution time whereas the dynamic approach could spawn new processes if some component is not used to its full capacity. In the second case the program reacts to a unsteady load, which could for instance result from an unknown input size in the beginning or a change of the data volume over time (caused by a input stream for example). If the program is executed with too few processes, the system is not fully loaded, on the contrary too many processes result in overhead and therefore in loss of performance. Spawning processes dynamically allows to fit the number of processes to the current data volume.

Another use case allows the usage of processes as threads. As discussed in **Design approach** (p.74) additional processes replace the threads to achieve MPI-only programming instead of a hybrid version (like a MPI-OpenMP model). If a hybrid program is executed on a distributed architecture the load needs do be distributed via MPI communication before threads start computation on shared memory. Before MPI-2.0 every process had to be spawned at program execution to make a MPI-only approach by what on the one hand too many processes had been spawned. On the other hand it induced the need to apply changes to every existent MPI-call to distinguish processes for distribution from processes for parallel computation (which replaced threads) - otherwise every MPI-call would affect all processes which generally leads to unintentional behaviour. Spawning processes dynamically allows to separate the distribution processes from the thread-like processes: every process spawns as many processes as threads were needed after the distribution phase. For that reason the new spawned processes, which are assigned to a new MPI communicator, do not interfere with existent MPI-calls and only work on

data, which has explicitly been assigned to them. Their lifetime can be limited to the section of parallel computation easily. An example of this usage is given in chapter **Using MPI-3 RMA** (p.60), where the replacement of OpenMP with processes gives an advantage regarding the performance.

Though it is possible to execute a program with just one process, which then spawns its siblings at runtime, it is advised to not do so, if there is no good reason for it. Due to performance reasons one should start every process at once, if possible [MPI03, 83].

Selection of important operations for spawning processes at runtime:

`MPI_COMM_SPAWN(command, ..., maxprocs, ..., comm, intercomm, ...)`

Executes the application specified in `command` with up to `maxprocs` many processes. The group of spawning processes - specified by the *intracommunicator* `comm` - is connected with the group of spawned processes with the *intercommunicator* `intercomm`. An intracommunicator enables communication between processes of the same group (like `MPI_COMM_WORLD`), an intercommunicator on the contrary connects two disjoint groups. The new spawned processes are united in their own `MPI_COMM_WORLD` and have therefore at first no connection to any other process, which has not been spawned by this specific operation. `MPI_COMM_SPAWN` must be called collectively by every process in `comm`.[1]

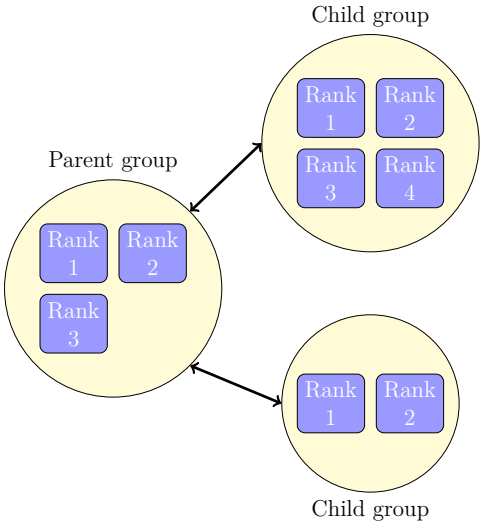


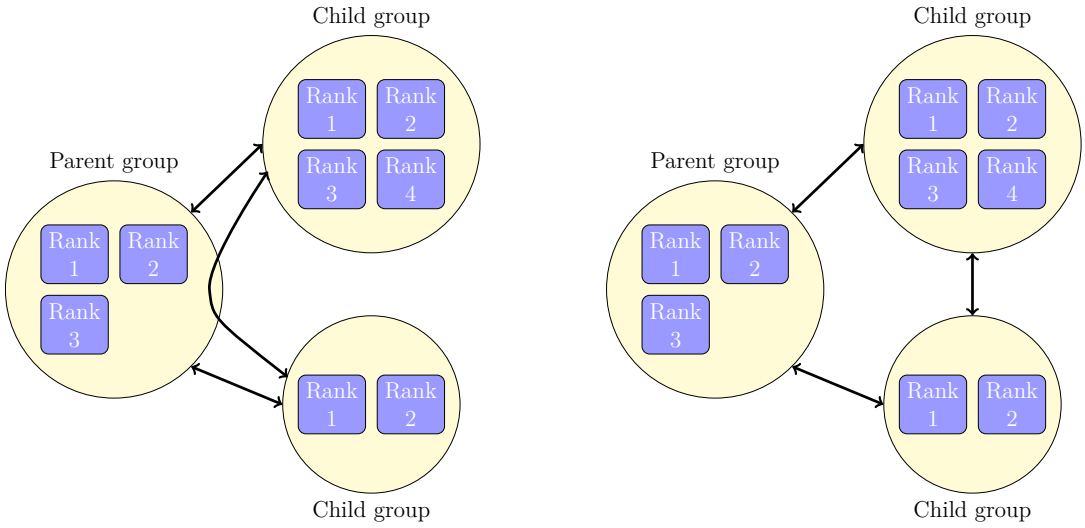
Figure 2.9: Spawning new processes during runtime

`MPI_COMM_GET_PARENT(parent)`

A process which has been spawned by a *parent*-process, has no knowledge of its parent[2]. To establish a communication with its parent, a process needs to get its parent's communicator `parent`

[1] If just one process shall spawn new processes, `comm` can be set to `MPI_COMM_SELF`. This communicator just contains the process itself.

[2] Only the parent has a connection to its *children* through `intercomm` initially



(a) Parent processes manage and redirect child-communication

(b) Direct communication between child-processes

Figure 2.10: Child communication

Figure 2.9 shows a group of processes, which have called `MPI_COMM_SPAWN` with `maxprocs`≥ 4 two times. As mentioned, each group of child processes can only communicate with a member of its own group or with a process of the parent group. To enable communication between spawned process groups there are two different possibilities.

1. The parent-group acts as a connector: Child-processes send messages to their common parent-group, which redirects all messages to the corresponding child-group (cf. figure 2.10a). In this case, the child groups have still no direct connection to each other.

2. The child-groups communicate directly with each other without any redirection (cf. figure 2.10b). This makes higher effort necessary, see [MPI03, 93–102] for an in-depth explanation.

Many operations for one-sided communication request an intracommunicator as input. Since spawning new processes returns an intercommunicator, which connects the group of spawning processes with the group of spawned processes, these groups need to be merged to obtain an intracommunicator, before one-sided communication can be established between parent processes and child processes. An intercommunicator consists always of two disjoint groups: the local and remote group. Every process' own group is his local group, communication between these groups can be established through communication, which is suitable for using an intercommunicator. It depends on the MPI standard, which communication operations fulfil this condition: `MPI_SEND`/`MPI_RECV` and their relatives could always use intercommunicators for example, but for collective operations an intercommunicator was not suitable until MPI-2 [GLS14, ch. 7.3]. `MPI_INTERCOMM_MERGE(intercomm, ..., newintracomm)` creates a new

intracommunicator `newintracomm` containing the groups, which are connected by the intercommunicator `intercomm`.

A major extension to the MPI standard were operations for one-sided communication. So far every MPI-message had been transmitted by a sending process to a receiving process[1] which necessitated each side to call the appropriate operation actively. One-sided communication allows to ease this requirement insofar that the target process of a communication does not need to call the corresponding operation anymore. Due to the fact that MPI-2.0 introduced just a part of the current extent of one-sided operations and that one-sided communication is crucial for the adaption of Phoenix (cf. section **Design approach** (p.74)), this topic will be discussed in detail in section **One-sided communication** (p.11).

2.2.2 MPI-3.0

Version 3.0 is an extension of MPI-2.2 and provides the MPI standard with extensions to nonblocking collectives, neighbourhood collectives, one-sided communication and Fortran 2008 bindings amongst others. See **One-sided communication** (p.11) for an in-depth explanation of presented operations and concepts regarding one-sided communication. Even though MPI-2 was an important step for one-sided communication it contains some flaws regarding performance and memory access patterns: strong restrictions concerning memory access and weak synchronisation semantics make it impossible to achieve the same performance with MPI-2 as with a PGAS language (cf. **RDMA** (p.12)). Corresponding to the MPI-2 standard it is for example erroneous to have concurrent get/put operations on the same shared memory, while this might be desirable. A trivial example would be: every process writes its rank into the same memory to determine, which process executed last. MPI-2 requires the use of locks, even though they are unnecessary in this context. MPI-3 weakens this requirement and declares the result of concurrent get/put operations as undefined but not erroneous anymore [BD04]. This is insofar an advantage, as an erroneous result breaks the program's execution, whereas with an undefined result the execution continues (regarding the use case an undefined result might be sufficient).

MPI-3.0 brings several new features and improvements to one-sided communication:

Communication: Every communication operation, which had been introduced with MPI-2.0, was expanded by a version with a request handle - This offers the possibility to poll the operation's status or even wait for its completion. Every MPI one-sided communication operation is nonblocking, so without a request handle, synchronisation operations were the only possibility to guarantee completion of communication operations. In addition to the request handle more atomic operations for diverse use cases had been added.

Synchronisation: To ease the right usage of used message buffers, passive synchronisation had been amended. This includes a new memory model, which allows to use less

[1] Or from/to a group of processes

synchronisation calls and thereby improves performance.

Window: MPI-2.0 offers only one operation to create a window for remote memory access, MPI-3.0 adds three more operations to allow a more dynamic memory usage. To create shared memory is part of this improvement[1] and an important contribution to this thesis' objective.

By now the current MPI standard is version 3.1 [MPI15], which mostly contains minor changes without any impact for this topic and will therefore be omitted.

2.2.3 Implementations

	MPICH	MVAPICH	Open MPI	Cray MPI	Tianhe MPI	Intel MPI	IBM BG/Q MPI [1]	IBM PE MPICH [2]	IBM Pla<orm	SGI MPI	Fujitsu MPI	MS MPI	MPC	NEC MPI
NBC	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	(*)	✔	✔
Nbrhood collecKves	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	✔	✔
RMA	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	Q2'17	✔
Shared memory	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	*	✔
Tools Interface	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	*	Q4'16	✔
Comm-creat group	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✘	*	✔
F08 Bindings	✔	✔	✔	✔	✔	✔	✔	✘	✘	✔	✘	✘	Q2'16	✔
New Datatypes	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	✔	✔
Large Counts	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	Q2'16	✔
Matched Probe	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔	✔	✔	Q2'16	✔
NBC I/O	✔	Q3'16	✔	✔	✘	✔	✘	✘	✘	✔	✘	✘	Q4'16	✔

Table 2.2: Overview over common MPI implementations as of June 2016 [* Under development, (*) Partly done] [For16]

Table 2.2 gives an overview of well-known MPI implementations and lists, which part of MPI-3.1 lays within their scope. Although it appears that almost every important MPI implementation has a version, which implements the MPI-3.1 interface, it should be noted, that the existence of an implementation does not necessarily mean that it is already optimized. For instance an implementation could mimic the effect of RMA-operations using conventional point-to-point communication. The result of this operations would be the same without utilizing the advantages of one-sided communication. Occasionally the user needs to pay attention choosing the right compilation flags, otherwise the built implementation underachieves. One example is the *Intel MPI Library 5.0 Beta*: Only the multithreaded version of this library uses asynchronous message propagation, which is required for best performance of RMA - synchronous message propagation would

[1] MPI-2.0 just allowed to simulate shared memory by using RMA-operations on a window. As seen in section **TrackerSim** (p.61) this approach is discouraged.

probably suffer loss in performance because of the lower portion of overlapping operations [BSCL14].

Implementation-specific details

On some points the MPI standard does not specify clearly how a correct implementation should handle them, which allows it to integrate different levels of optimisation gradually.

Many RMA operations have an `info` argument, which offers hints about their expected usage. During runtime a MPI implementation may use this information to optimize the operation's execution - but it may also ignore the hint. If feasible, info arguments should be set, it could generate some speedup. See [MPI12, Ch. 11.2] for a selection of info arguments, `info = MPI_INFO_NULL` is always a valid value.

Similar to info arguments some operations take an `assert` argument, which allows an implementation to optimize the execution based on given assumptions. See [MPI12, 451–452] for a selection of assert arguments, `assert = 0` is always a valid value.

It is erroneous to provide a false hint or assert argument. Every implementation may have its own asserts and hints, but using them might restrain an application's portability. [GT07] showed that at the time of their benchmarking many MPI implementations (including MPICH2 1.0.5 and OpenMPI 1.2.0) ignored the majority of these arguments, but this could have changed in the meantime.

The standard grants flexibility to the implementation regarding the completion of RMA operations, which might result in a better performance. The standard just defines phases between which operations must be completed, the specific moment is left to the implementation. To keep a program portable, users should stick to this conservative view, even though some implementations would allow to deviate. See [MPI12, Ch. 11.7.3] for more information about one-sided communication progress.

2.3 Available operations

The following three sections give an overview about available operations for MPI one-sided communication, which are part of the MPI-3 standard. Every section covers a different type of one-sided operation, which are all necessary for performing one sided communication: First off all, memory needs to be prepared for using one-sided communication, different approaches are discussed in section **Initialisation** (p.29). Section **Communication** (p.32) explains the actual communication operations. The last section **Synchronisation** (p.34) covers possibilities for synchronising access and update operations. While the included figures 2.11,2.14,2.15 contain every operation that is part of the MPI standard chapter about one-sided communication, only those are discussed, which are relevant to this thesis' topic. Operations are connected with a dashed line, if they are semantically related. Each operation has the prefix *MPI_Win_*, which is omitted in the figures.

The type annotation of the operations, which are presented in this section, conform the

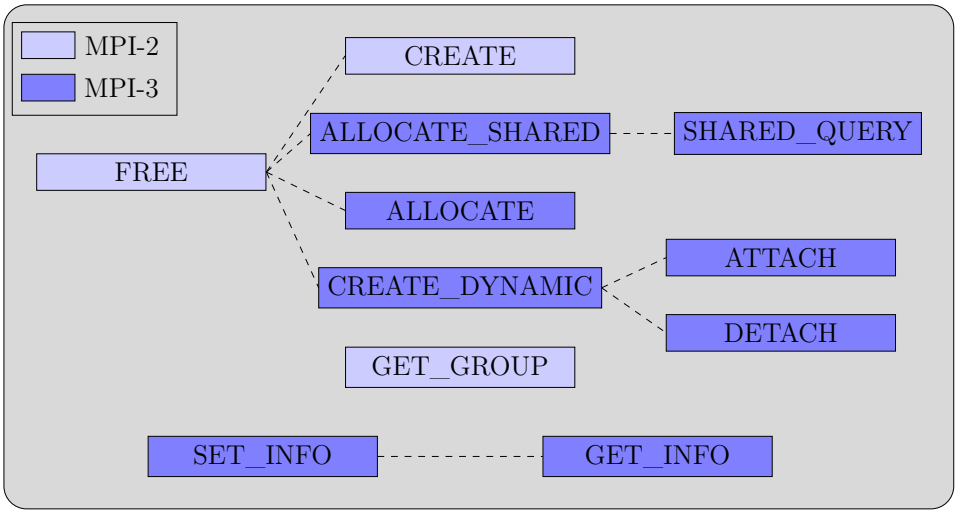


Figure 2.11: RMA initialisation operations

Fortran[1] version used with `USE mpi`, if not mentioned otherwise. Negligible arguments, which are not important to understand a operation's functionality, as well as the last argument `ierror`[2] are omitted. This includes arguments for determining datatypes and displacements: To simplify matters, operations for creating windows and executing communication on them take displacement-values as input argument. A displacement acts as an offset and makes it possible to access elements on a window in the same way as on an array. It is even possible to expose a derived datatype through a window and access or update it in exactly the same way as usual - see **Substitution of OpenMP** (p.77) for such an example. To keep the following description simple, the explanations suppose only a primitive data type for memory exposition unless otherwise noted. See [MPI12, Ch. 11] for an explanation of every operation in detail.

2.3.1 Initialisation

The major feature of one-sided communication in comparison to previous MPI concepts, where it is essential to match every sending operation with a corresponding receiving operation, is the ability, to allow the origin process to perform communication without the need for a corresponding target operation. Memory, on which one-sided communication operations are allowed, must be specified explicitly. This is done by creating a *local window*, that acts as a kind of view on a process' memory. Access or update operations on a target's memory without using a window are impossible. A local window defines a contiguous section of memory with a address and size (amongst other things), which belongs to a process and exposes memory for remote access performed by other processes of the same group. A *window object* is a handle, which is returned by the operations for creating a window. It represents the collection of every process' local window. Obviously

[1] Fortran 90 or higher

[2] Additional argument of every MPI operation in Fortran, which corresponds to the return value of the operation in C.

a local window is different from a window object - nevertheless literature frequently refers to both items simply as *window*, the proper meaning needs then to be taken from the context [GLS14, Ch. 5.3].

Creating a window is a collective call on an intracommunicator, however each process may select any size for the exposed memory irrespective of what the other processes choose. Even `size == 0_MPI_ADDRESS_KIND` is a valid value, which actually is a common use case if only one process designates memory and every other process uses it without designating memory itself. Therefore `size` only sets the minimum amount of bytes, which belongs to a process own memory, the total size of the created window is the sum of every process' `size` argument at least. Except a window, that was created with `MPI_WIN_CREATE_DYNAMIC`, every window is immutable.

Since a window handle cannot be exchanged via messages or similar, the only possibility to get one is to create it with one of the following operations:

`MPI_WIN_CREATE(base,size,...,comm,win)`
: This is the trivial way to create a window. A process, which invokes this operation, creates a window `win` on `size` bytes of its memory with the first element at `base` - this section of the process' local memory section can from now on get exposed. This memory is accessible by every other process in `comm` only through RMA operations with the window handle `win`. The variable `base` needs to have a static type or else be allocated before. The MPI-3 standard does not require, that `MPI_ALLOC_MEM` is used for the allocation, but implementations are free to make this a prerequisite [MPI12, Ch. 11.5.3].

`MPI_WIN_ALLOCATE(size,...,comm,baseptr,win)`
: Different from the prior window creation, this operation does not require memory, which had already been allocated. It allocates on each process of `comm` local memory with at least `size` bytes instead and returns a pointer `baseptr` to this memory. In case of doubt whether to choose `MPI_WIN_CREATE(...)` or `MPI_WIN_ALLOCATE(...)`, the latter should be preferred as it offers an implementation the possibility for optimisations [DBB+16, 5]. Again, accessing and updating the exposed memory is possible only through RMA operations.

`MPI_WIN_ALLOCATE_SHARED(size,...,comm,baseptr,win)`
: As before every process of `comm` allocates at least `size` bytes and returns `baseptr` as a pointer to this memory. The execution of this operation requires that every participating process shares the same local memory. If any process' memory, which takes part in the creation of the shared window, is distributed relating to the other ones, the execution of this operation is not possible. To guarantee that only processes with the same shared memory take part in the collective call, a `comm` must be used, which fulfils this requirement. If every process executes `MPI_COMM_SPLIT(comm, split_type, ..., newcomm)` with its communicator `comm` and `split_type == MPI_COMM_TYPE_SHARED`, then `newcomm` contains the intersection of `comm` and every process, which shares the same memory with the executing process. Provided that the appropriate communicator is chosen for `comm`

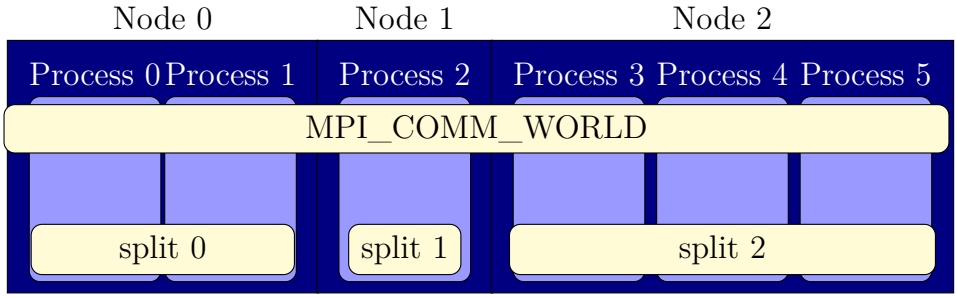


Figure 2.12: Receive communicator with maximum extent respectively shared memory based on [GHTL14, Ch. 5.7]

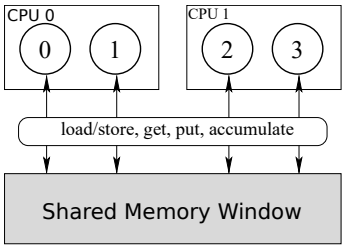


Figure 2.13: Accessing window with shared memory [HDB+13, 5]

this operation provides with `newcomm` a communicator with the largest possible size of shared memory regarding the location of the process, which invokes this operation. Figure 2.12 shows an example for generating communicators on shared memory with maximum size, while the processes are distributed over three nodes, which do not share one major shared memory. Each process of a node may then create collectively a shared window

Before any process may access any other process' window, it must call `MPI_WIN_SHARED_QUERY(win, rank, ..., baseptr)` with the relevant window `win` and the rank `rank`, which the target process occupied in `newcomm`. In `baseptr` the proper address to the memory is set, which is exposed through `win`. Due to the fact, that this function may return varying addresses for the same memory, if executed by different processes, this addresses must not be exchanged. If a process wishes to access another process' shared memory, it must execute `MPI_WIN_SHARED_QUERY(...)` by itself.

These inconveniences come along with one major advantage: contrary to windows created by any other create operations than this, a shared window's memory can be accessed and update with RMA operations as well as with local load/store operations like in figure 2.13. As mentioned in **Use cases** (p.20) and shown in **TrackerSim** (p.61) this may lead to an improved performance.

`MPI_WIN_CREATE_DYNAMIC(...,comm,win)`

Until now every operation to create a window required that the final memory size and location, which shall be exposed through a window, is already known. This

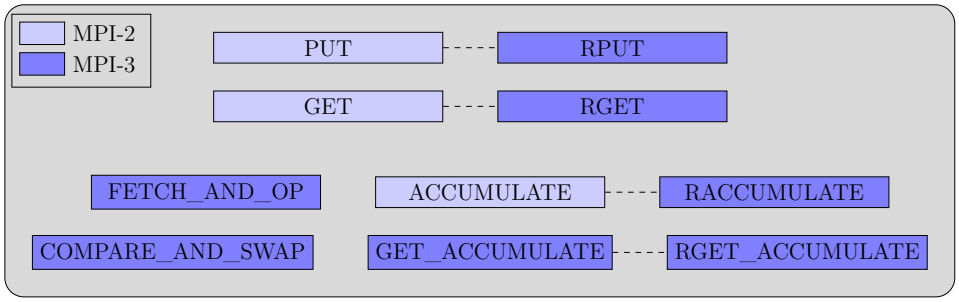


Figure 2.14: RMA communication operations

makes it impossible to change the exposed memory during runtime. To create a window `win` with a dynamic amount of memory, the processes of `comm` need to invoke this operation. Immediate after the window creation `win` has no memory assigned. To attach allocated memory of size `size` to its window `win`, a process needs to execute `MPI_WIN_ATTACH(win, base, size)` with the memory's address `base`. After the attachment the memory is accessible by every process of `comm`. Attached memory sections must not overlap. To remove attached memory from a window `win` a process simply calls `MPI_WIN_DETACH(win, base)`.

This dynamic approach might be beneficial with regard to performance, if the underlaying hardware (e.g. a RDMA network architecture) supports it, but it might be just as well more expansive than a static approach because of the overhead, which is induced by maintaining dynamic memory sections [HDT+13, 5].

A window object can be provided with additional information or even methods, which are automatically executed on the window's duplication or destruction. A possible use case could be a method, which frees memory belonging to a window - this would prevent memory leaks if a user only frees the window object and forgets about the associated dynamic memory [GHTL14, 113][1].

Once a window `win` is not required anymore it can be freed with `MPI_WIN_FREE(win)`. This is again a collective call, which needs to be executed by every process of a group, which is associated with `win`.

2.3.2 Communication

Every RMA communication operation can only be executed on memory, which its process has explicitly exposed for remote access and update, i.e. on a window. A process can only access or update a window with local store/load operations if either the process is the owner of this window (i.e. the window's memory is also part of the process' local memory) or the window references shared memory (i.e. it had been created through `MPI_WIN_ALLOCATE_SHARED`). There are only three different kinds of RMA communication in principle: `MPI_PUT`, `MPI_GET` and `MPI_ACCUMULATE`. Every other RMA communication operation, shown in figure 2.14, is just a variation of these.

[1] See [GHTL14, 115] for current restrictions of this feature

MPI_PUT(origin_addr,...,target_rank,...,win)

This operation transfers data at origin_addr to the process' local memory with rank target_rank, which had been exposed through the window win. The nearest equivalent point-to-point communication operation for MPI_PUT is MPI_SEND (in combination with the required MPI_RECV).

MPI_GET(origin_addr,...,target_rank,...,win)

This one transfers data from the process' window win with rank target_rank to the caller's memory at origin_addr. Compared to the prior operation the direction of data transfer is reversed and would correspond to MPI_RECV.

MPI_ACCUMULATE(origin_addr,...,target_rank,...,op,win)

Similar to MPI_PUT this operation transfers data from origin_addr to the target window win with rank target_rank. The difference is, that on the one hand MPI_ACCUMULATE is executed atomically for each basic datatype and on the other hand the target element is not only replaced by the element at origin_addr. Comparable to MPI_REDUCE the target element will be replaced with the result of the operation op executed on the origin element and target element.

Because RMA operations are not limited to single element with one datatype, one needs to be careful with executing two atomic RMA communication operations concurrently on more than one element on overlapping memory. Without proper synchronisation, the result is undefined.

Using op == MPI_REPLACE or op == MPI_NO_OP allows to use MPI_GET_ACCUMULATE as an atomic version of MPI_PUT or MPI_GET. A list with all possible operations for op is available at [MPI12, 668].

MPI_GET_ACCUMULATE(origin_addr,...,result_addr,...,target_rank,...,op,win)

The disadvantage of MPI_ACCUMULATE is, that the process, which invokes this operation, does not receive the result of op. Many use cases require, that they get and update the target value without the risk of a race condition[1]. In this case MPI_GET_ACCUMULATE can be used: In addition to the behaviour of MPI_ACCUMULATE, this operation returns the value of target_rank to result_addr[2] before op is applied to it.[3]

MPI_FETCH_AND_OP can be substituted for MPI_GET_ACCUMULATE if op shall be executed on one element only. The functionality of MPI_FETCH_AND_OP is less generic than its counterpart and makes an implementation with better performance possible.

[1] See **Substitution of OpenMP** (p.77) for such a use case

[2] result_addr and origin_addr must be disjoint

[3] If the process requires the result of op, it needs to apply op with origin_addr on result_addr itself on its local memory

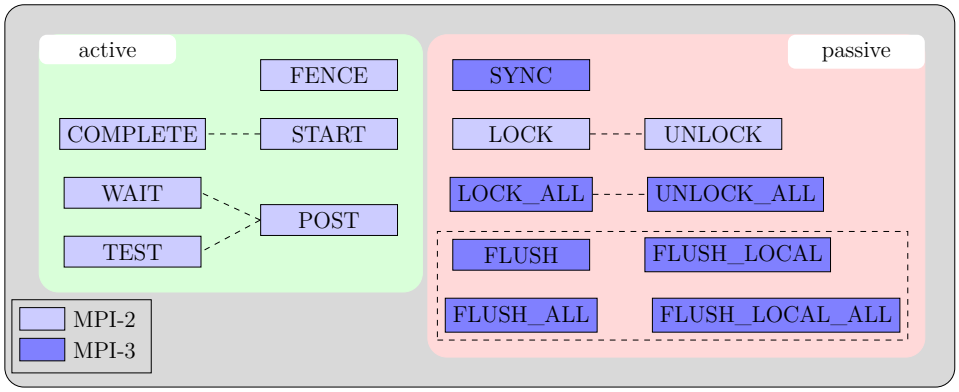


Figure 2.15: RMA synchronisation operations

For every presented communication operation exists a version with the same semantic and a request handle in addition: `MPI_RPUT`, `MPI_RGET`, `MPI_RACCUMULATE` and `MPI_RGET_ACCUMULATE`. Those operations may only be called during *passive target communication* (cf. **Synchronisation** (p.34)). With the request object a user may test with `MPI_TEST`, if the associated operation had already completed, or wait with `MPI_WAIT` for its completion. In this case the term *completed* only indicates local completion, i.e. that the local buffer may be reused without any side-effect. But at this point the update on the remote memory might still be pending. Using request handles generates additional overhead but might at the same time improve the performance by offering a higher ratio of overlapping functions: Instead of delaying a computation phase until every process finished its remote operation, every process might already compute with a buffer, whose associated operation had been tested as completed. See [MPI12, Ch. 11.3.5] for more information and [HDT+13, Listing 1] for an example code for the mentioned use case.

RMA can only show its strength if the MPI implementation is tuned for performance and the underlaying hardware supports RMA (cf. **RDMA** (p.12)). However the MPI standard does not impose such optimisations. A RMA implementation may reproduce the behaviour of one-sided communication operations with hidden point-to-point communication as well.

As already mentioned before, `target == origin` is valid and allows a process to access and update its own window[1].

2.3.3 Synchronisation

MPI RMA introduces the concept of the *access epoch* and *exposure epoch* to administer the synchronisation of MPI RMA communication operations. An epoch describes the region between two synchronisation calls on a window and is also assigned to the process, on whose window the epoch had been created. Every communication operation, which happens during an epoch, has not completed for sure till the epoch is closed. A

[1] Only relevant in combination with the separate memory model (cf. **Memory consistency model** (p.14))

communication operation may complete locally as well as remotely before, but only closing the epoch can guarantee the communication's completion. As mentioned there are two different kinds of epoch: An access epoch signals, that the associated process may access remote memory; the exposure epoch signals, that the local window of the associated process may be accessed by another process, which had entered an access epoch for its part. A process may be in both different epochs simultaneously and therefore access remote memory while being a target of one-sided communication itself.

During an epoch any number of RMA operations[1] may be invoked - it is recommended to use as many RMA communication operations during an epoch as possible, to minimise the amount of RMA synchronisation calls [MPI12, 437]. The reason for this attitude is the fact, that RMA synchronisation operations are crucial for the final performance [TGT05]: The more RMA communication operations are executed during one single epoch the better RMA performs compared to message passing. An epoch is created and closed by invoking RMA synchronisation operations. Depending on the target process' behaviour during a RMA synchronisation call, there are two different communication categories: *active target communication* and *passive target communication*. During the execution of a RMA synchronisation operation the target process is either actively participating (active target) or remains passive (passive target). The choice of category has no effect on the RMA communication operations, it only changes the set of available RMA synchronisation operations.

The following three sections will discuss these categories and how to use them: Sections **Active target** (p.36) and **Passive target** (p.37) list the most important functions to close an epoch with or without the target's cooperation. Because there are relative many synchronisation possibilities compared to the common two-sided communication, section **Synchronisation application** (p.39) gives a review and hints in using RMA synchronisation operations correctly.

Active target communication is similar to common message passing: The target process must participate actively to complete a RMA communication operation although the RMA communication operation itself is solely executed by the origin. Because of the need for cooperation between origin and target, this scheme usually fits best to a static communication pattern with bulk-synchronisation (cf. *Division of labour* in section **Use cases** (p.20)). Passive target communication matches well with the concept of shared memory: The origin accesses or updates data on a target process' local memory, without the necessity of a target's reaction. Another domain, where passive target communication is usually preferred to active target communication, is a dynamic communication pattern. If the pattern becomes irregular or is even unpredictable, the processes would need to agree frequently on a mutual approach relating to synchronisation calls if active target communication was used - this would generate additional overhead, which passive target communication can avoid. Since the active target communication is similar to message passing and passive target communication allows one-sided synchronisation, the latter one is most suitable for mimicking thread parallelisation with MPI RMA operations.

[1] An epoch has only an effect on RMA operations, every other operation - like a local or two-sided communication operation - remains unaffected

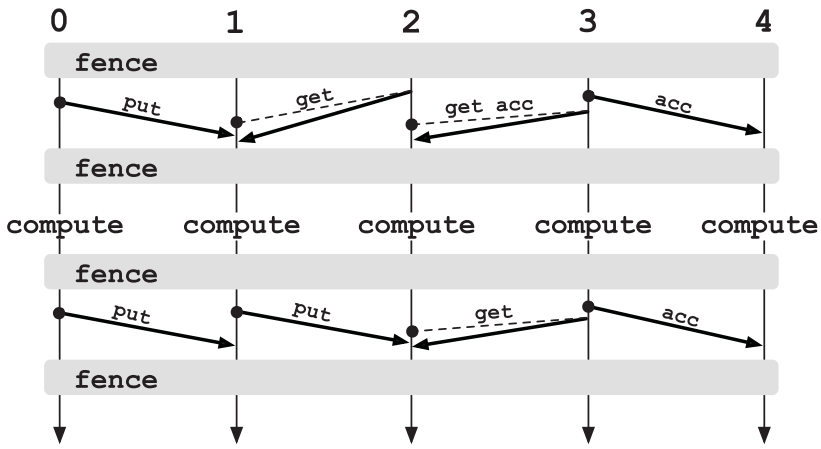


Figure 2.16: Collective access and exposure epoch through fence synchronisation [HDT+15, 9]

Section **TrackerSim** (p.61) includes diverse examples to implement replacement.

The following synchronisation operations for both communication categories may be used in combination in an application. However it is erroneous to lock a window, which already actively exposed its memory (and vice versa), which is why it is recommended by [MPI12, Ch. 11.5.3] to stick to one RMA communication category; otherwise additional synchronisation becomes necessary to guarantee the mutual exclusion respective the categories.

Active target

There are two different kinds of active target synchronisation operations: Fence synchronisation and general active target synchronisation - the latter is sometimes referred to as *GATS*. In both cases the origin as well as the target process(es) need to invoke RMA synchronisation operations.

`MPI_WIN_FENCE(..., win)`

This is a collective synchronisation call (so every process, which is associated with the window object `win`, needs to participate, c.f. figure 2.16) and is frequently used, if every process alternates between a computation phase and a communication phase. Thereby it is irrelevant, if a process was target or origin of a RMA communication operation during this epoch - or did not participate at all. A Fence induces always an access and exposure epoch for the affected processes. Because of its collective approach `MPI_WIN_FENCE` is suitable for bulk-synchronisation or if each process needs to communicate with many other processes.

GATS

An alternative for active target synchronisation, which is not a collective operation, is a combination of four operations: `MPI_WIN_START(group, ..., win)` and the corresponding `MPI_WIN_COMPLETE(win)` as well as `MPI_WIN_POST(group, ..., win)` and the corresponding `MPI_WIN_WAIT(win)`. The

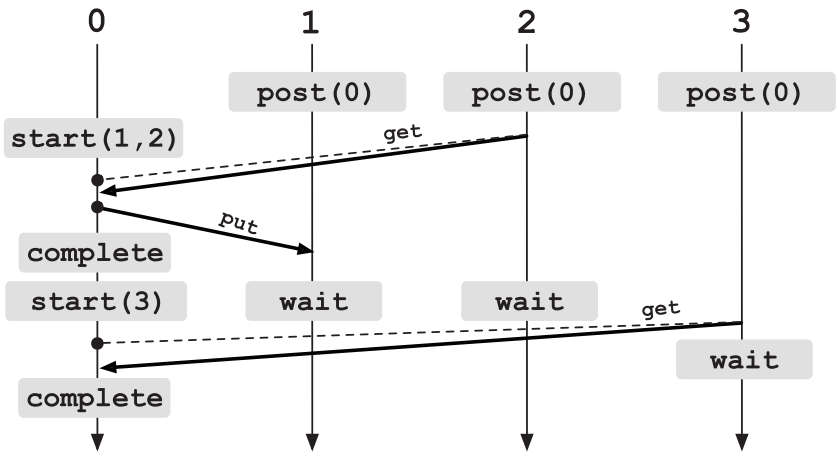


Figure 2.17: Selective access and exposure epoch through active GATS synchronisation [HDT+15, 9]

advantage of GATS over fence synchronisation is, that these operations are not collective and therefore minimize the amount of expensive synchronisation. Instead of entering an access and exposure epoch automatically with `MPI_WIN_FENCE`, each process can choose if it wants to enter an epoch and if so, whether it is an exposure or access epoch (or both). An access epoch exists between the invocation of `MPI_WIN_START` and `MPI_WIN_COMPLETE`, an exposure epoch between `MPI_WIN_POST` and `MPI_WIN_WAIT`[1]. The argument `group` contains every process, which is participated with `win` and shall be accessed (calling process is in access epoch) or is allowed to access (calling process is in expose epoch)[2]. Remote access by an origin process can be executed as soon as the target process has exposed its memory - as long as it is not exposed, nothing happens. The MPI standard does not state, that `MPI_WIN_START` must block until the corresponding `MPI_WIN_POST` has been executed, but then the origin's first put operation on the target's memory must block at the latest[3].

Figure 2.17 shows an example, where a few processes access or update remote memory with GATS.

Passive target

In passive target communication a target process does not invoke a RMA synchronisation operation, to allow or finalise an exposure epoch. In fact the exposure epoch loses its relevance in passive target communication, as every process is supposed to expose each

[1] `MPI_WIN_TEST(win, flag)` is a non-blocking alternative to `MPI_WIN_WAIT` and returns with `flag`, if all remote accessed have already been completed. If not, the exposure epoch can not be completed yet

[2] I.e. the target processes must acknowledge, to which processes they will actually expose their memory

[3] Every MPI implementation is free to choose its own approach, as long as no remote access happens before the target exposes its memory

local window. Therefore the synchronisation becomes one-sided, too. One advantage is a better performance, which can be expected because of less synchronisation. But this comes with a disadvantage, too: two-sided synchronisation calls provides a higher level of safety, because the target needs to allow to be accessed explicitly. Now every process with the same window object may access or update remote memory at any time, so the user must pay more attention to prevent erroneous behaviour.

`MPI_WIN_LOCK(lock_type,rank,...,win)`

A process, which invokes this operation, initiates an access epoch and prepares the local window (which belongs to the window object `win`) of the process with rank `rank` to be accessed or updated by it[1]. To specify if other processes may access the locked window concurrently too, there are two possible assignments to `lock_type`:

- `MPI_LOCK_SHARED`: Every process, which locked the same window with `lock_type == MPI_LOCK_SHARED`, may access it concurrently. Without further synchronisation the order of concurrent access and update operations is undefined. Therefore this lock-type should only be used for access operations. If a process locked the same window with `lock_type == MPI_LOCK_EXCLUSIVE`, its operations are not executed concurrently to the operations of the processes with shared locks.
- `MPI_LOCK_EXCLUSIVE`: This lock guarantees, that no other process may access or update the locked window concurrently.

If every local window, which is associated with `win`, shall be locked, a process may invoke `MPI_WIN_LOCK_ALL(...,win)`, which sets a lock with `lock_type == MPI_LOCK_SHARED` on the local window of every process (including its own). Executing operations without proper or even no locks may have an undefined outcome. A general approach is to set a shared lock before an access operation and an exclusive lock before update operations.

`MPI_WIN_UNLOCK(rank,win)`

The access epoch initiated by executing `MPI_WIN_LOCK` on a process' local window is completed by this operation. To remove the lock set by `MPI_WIN_LOCK_ALL`, `MPI_WIN_UNLOCK_ALL(win)` needs to be invoked.

`MPI_WIN_FLUSH(rank,win)`

This operation completes every RMA communication operation invoked by the calling process, which affects the local window associated with `win` of the process with rank `rank`. This call considers the completion at the origin as well as at the target. If a process only wants to make sure, that every of its invoked operation is completed locally and buffers are ready to be reused[2], `MPI_WIN_FLUSH_LOCAL` is a valid alternative. For both operations exist versions, to affect every process

[1] The lock itself only affects the local window but not the associated process

[2] Any update operation might still be pending at the target after returning from this call

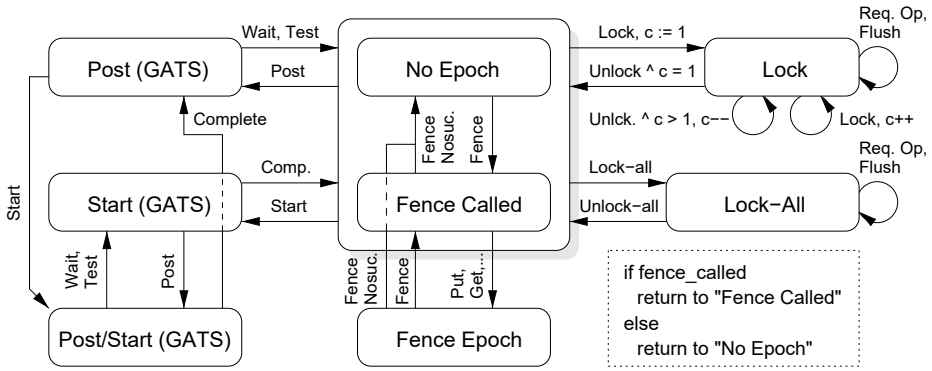


Figure 2.18: State diagram for correct MPI RMA synchronisation

associated with `win`: `MPI_WIN_FLUSH_ALL` and `MPI_WIN_FLUSH_LOCAL_-ALL`.

`MPI_WIN_SYNC(win)`

As already discussed in section **Memory consistency model** (p.14) there exist a private and public copy of a process' local window, which might be diverse if the separate model is used. In this case updates on one copy are applied to the other copy only eventually, calling this operation guarantees, that both copies are synchronized.

2.3.4 Synchronisation application

Due to the large amount of available RMA synchronisation operations it demands additional attention, to not only create a valid application but also an application without breach of specification. Compiling an application, which makes use of RMA synchronisation operations in a wrong way, might be possible but could have an undefined outcome. It must be emphasized, that `MPI_BARRIER`, which is commonly used in MPI applications only accomplished process synchronisation but not memory synchronisation - so it is no alternative to RMA synchronisation operations.

It is indeed possible, to guarantee the correct use of RMA synchronisation operations. Figure 2.18 is a synchronisation tracking diagram, which allows to ensure the correct use of synchronisation calls. If it is applied to a concrete application and tracking down the operation calls leads to a deadlock (i.e. the state "No Epoch" is unreachable from the current state), the application is erroneous. But this should also be reported by the compiler respectively the MPI implementation [DBB+16, Ch. 4.4]. However using this diagram offers no statement about the quality of the application's result. To prove the correct use of not only RMA synchronisation operations but every occurring RMA operation, another approach is to write down the sequence of the application' operations using formal semantics, which are defined in [HDT+13]. With this formal definitions of valid actions[1] and their execution the user may derive a formal specification based on

[1]Possible actions: memory action, synchronisation action

the their application. Thereby it is possible to prove (assuming that the application had been translated correctly into the semantic model) the application's correctness and that its result is well-defined and deterministic.

If both approaches are too complex, [GHTL14, Ch. 3.7.3] has a list of a few rules regarding the correct use of RMA synchronisation operations. They are stricter than necessary to keep them simple.

1. Do not overlap concurrent RMA access operations on the same local window. RMA accumulate operations with the same MPI datatype and reduce operation are an exception[1]. But concurrent accumulate operations on the same array might still result in a combination of both calls - only the basic datatype is protected against concurrent updates.

2. Separate RMA operations from non-RMA operations.

3. In an epoch without any store or update operations, load and access operations may be invoked in any order without further synchronisation.

[MPI12, Ch. 11.7] offers lists of rules, too. Although these rules are more complex, they distinguish between more possible circumstances and consider the used memory model. Applying them might therefore result in a better performance.

[1] Though concurrent accumulate operations are executed in an arbitrary order without further RMA synchronisation

3 Phoenix

Because of their distance it is very difficult to examine stars. Simulations are required to interpret data, which has been collected by the common observation methods: analysing their spectrum or even collecting emitted particles like neutrinos with suitable satellites or ground-based detectors. The radiation itself only allows direct sight into the stellar atmosphere, because this is the location from where it is essentially emitted. Due to this fact the inner region of a star cannot be probed directly. One possibility to handle this situation is to simulate a region of the star while assuming its inner composition.

`PHOENIX` is under continuous development by P. Hauschildt and E. Baron and their research groups. It is able to simulate the atmosphere and its spectrum for a wide variety of objects, ranging from main sequence stars and giants, brown dwarfs and planets, via novae and supernovae to accretion disks. To do so, `PHOENIX` is executed amongst others with assumptions of the composition of an object, the radiation transport equation is numerically solved and a synthetic spectrum of the object of interest is generated subsequently. Comparing the synthetic spectrum calculated by `PHOENIX` with the actual observed spectrum allows an assessment and enhancements of the parameters of the star.

Since `PHOENIX` offers a wide range of models and methods, which can be used, this chapter focuses on one special case and leaves everything else out for the sake of brevity. Therefore, the following chapters only describe one possible case and do not make a claim to be exhaustive.

3.1 3D radiative transfer

The 3D radiative transfer describes the spatial propagation of electro-magnetic radiation through a medium, which might absorb, scatter or emit radiation itself. To generate the synthetic spectrum it is necessary to determine the radiation field. A full description of the radiation field at every point in space needs the specific intensity I_λ at wavelength λ. From I_λ the mean intensity J_λ and the radiative flux $\vec{F}_\lambda$ can then be derived.

3.1.1 Theory

The following derivations are made accordingly to [See08, 9–22], [HB06] and [Aqu], which offer a more in-depth explanation.

A description of the radiative field requires knowledge about the changes in energy of a wavelength interval $[\lambda, \lambda + d\lambda]$ during time dt through a cross section $d\sigma$ in direction $d\vec{\sigma}$ into the solid angle $d\vec{\Omega}$ and is given as a function of the specific intensity I_λ through:

$$I(\vec{r}, d\vec{\sigma}, \lambda, t) = \frac{dE(\lambda)}{d\vec{\sigma} d\vec{\Omega} dt d\lambda} \tag{3.1}$$

The static 3D radiative transfer equation may then be written as:

$$d\vec{\sigma} \nabla I(\vec{r}, d\vec{\sigma}, \lambda) = \eta(\vec{r}, \lambda) - \xi(\vec{r}, \lambda) I(\vec{r}, d\vec{\sigma}, \lambda) \tag{3.2}$$

$\eta(\vec{r}, \lambda)$ is the local emissivity and $\xi(\vec{r}, \lambda)$ the local extinction, their fraction is defined as the source function S.

$$S = \frac{\eta}{\xi} \tag{3.3}$$

The origin of the source function is the medium, through which the radiation propagates. The medium has an inner energy and will therefore, with a distinct rate, emit radiation itself. In the same way the medium may change the radiation, as it absorbs or scatters it. Integrating the specific intensity over all solid angles results in the mean intensity $J(\vec{r}, \lambda)$ of a specific wavelength; considering the spatial direction in addition, gives the radiative flux $\vec{F}(\vec{r}, \lambda)$:

$$J(\vec{r}, \lambda) = \frac{1}{4\pi} \oint_{4\pi} I(\vec{r}, \lambda) d\vec{\Omega} \tag{3.4}$$

$$\vec{F}(\vec{r}, \lambda) = \oint_{4\pi} I(\vec{r}, \lambda) d\vec{\sigma} d\vec{\Omega} \tag{3.5}$$

The correlation between the mean intensity J and the source function S can be derived from a formal solution of the radiative transfer equation (3.2) and is specified by the Schwarzschild-Milne equation. The mean intensity is defined as the influence of every specific intensity over every solid angle and is, therefore, obtained by integration. This solution can then be substituted through the introduction of the so-called Λ *operator* and is then written as:

$$J = \Lambda[S] \tag{3.6}$$

As already noted before the source function describes the ratio of the medium's emission and extinction. Therefore it can be expressed as the sum of the scatter of the bypassing radiation with scattering coefficient σ (not to be mistaken with the cross section) and the emission of energy, which had been absorbed before with absorption coefficient κ:

$$S = \frac{\sigma}{\kappa + \sigma} J + \frac{\kappa}{\kappa + \sigma} B \tag{3.7}$$

$$= (1 - \epsilon) J + \epsilon B \tag{3.8}$$

with the thermal coupling parameter $\epsilon = \frac{\kappa}{\kappa+\sigma}$. Inserting this relation in 3.6 produces:

$$J = \Lambda[(1-\epsilon)J + \epsilon B] \tag{3.9}$$

To solve this equation for J an iteration scheme is used, which allows to determine the source function sufficiently well:

$$J^{(n+1)} = \Lambda[S^n], \qquad S^n = (1-\epsilon)J^n + \epsilon B \tag{3.10}$$

This is the so-called Λ-iteration method. One possible initial guess to start the iteration with is $J^0 = B$, which considers the atmosphere as a black body in first approximation and makes the use of the Planck function imperative. The convergence rate of this iteration scheme depends on ϵ, if ϵ becomes too small the convergence rate is unusable. To approach this problem, the iteration scheme must be adapted to the *operator splitting method*. Using the identity, Λ can be extended:

$$\Lambda = \Lambda^* + (\Lambda - \Lambda^*) \tag{3.11}$$

Insert this extension into equation 3.6:

$$J = \Lambda^* S + (\Lambda - \Lambda^*)S \tag{3.12}$$

Let $J_{FS} = \Lambda[S^n]$ be the formal solution with the source function of the last iteration. To fit 3.12 to the form of a common iteration scheme $Mx^{k+1} = Nx^k + b$, it can then be rewritten as suggested by [Ham87]:

$$J^{n+1} = [1 - \Lambda^*(1-\epsilon)]^{-1}[J_{FS} - \Lambda^*(1-\epsilon)J^n] \tag{3.13}$$

Supposing that Λ^* is well chosen, computing the mean intensity alternately with 3.13 and 3.8 will converge much faster than with 3.10. The subsequent section gives an example for the specific choice of Λ^*.

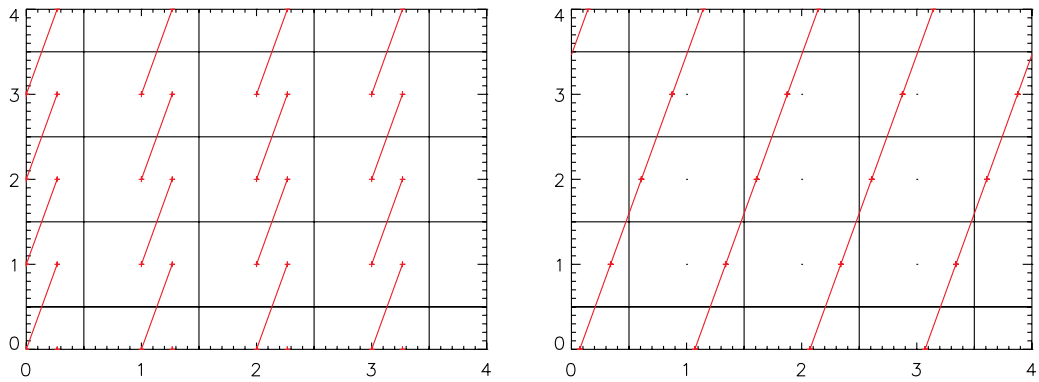
3.1.2 Implementation in Phoenix

The purpose of PHOENIX is to generate synthetic spectra of atmospheres. For this thesis' topic the focus is on the 3D-mode, so the atmosphere needs to be described as a volume. This volume is represented as a voxel grid and, disregarding possible limitations because of numerical effects, its resolution is freely adjustable. Since it is impossible to calculate the synthetic spectrum analytically, numerical approximations need to be made. It is important to choose a suitable implementation for Λ^* to perform the iteration scheme with an optimal convergence rate. The following construction is only a summary, [HB99] gives an in-depth explanation:

$$I_i^k \equiv I_{i-1}^k exp(-\Delta\tau_{i-1}^k) + \alpha_i^k \hat{S}_{i-1} + \beta_i^k \hat{S}_i + \gamma_i^k \hat{S}_{i+1} \tag{3.14}$$

These values are calculated along a so-called *characteristic*, denoted with k, which describes a specific spatial direction. Following these characteristics the source function S and the optical depth τ are interpolated, the values of α, β and γ depend on the chosen kind of interpolation.

The characteristics spread from the current voxel j in many uniformly distributed spatial directions (specified with (θ, ϕ), which describes a solid angle with polar angle θ and azimuthal angle ϕ), the resolution is again freely adjustable. Every voxel which is passed as the characteristic propagates through the grid, is denoted with i. As seen in figure 3.1 there are two different kinds of characteristics: *short characteristics* and *long characteristics*, hereafter *SC* and *LC*. The only difference is their way of propagation:



(a) Short characteristic (b) Long characteristic

Figure 3.1: Two different kinds of characteristics

SC: The start of this kind of characteristic is always set to the centre of a voxel, runs in the direction determined by (θ, ϕ) and stops at the point closest to the centre of the neighbouring voxel.

LC: This characteristic starts at the centre of its voxel and runs without any predefined breakpoint through neighbouring voxels.

In PHOENIX the calculations alongside a certain characteristic, defined by (θ, ϕ), are carried out by a *tracker*, which comes with a multitude of variations. One of these variations is the LC-tracker *LC_Lstar_tracker_PBC_zmap*, which applies periodic boundary conditions: Because the grid is finite in every dimension, characteristics will hit a boundary eventually. If a characteristic hits a boundary in x- or y-dimension, it continues on the opposite side of the grid. To avoid, that this happens too often before it reaches the boundary in z-dimension (at a specific amount more repetitions do not improve the accuracy significantly anymore), a maximum amount of applying boundary conditions is set. After the tracker has been executed for every voxel and its

characteristics, Λ^* is successfully calculated for every voxel and, therefore, one iteration step can be performed. After reaching the desired convergence and stopping the iteration scheme the mean intensity for every voxel is known and therefore the synthetic spectrum, generated by the grid, can be derived.

At this juncture only some LC-tracker make use of OpenMP.

3.2 Program flow

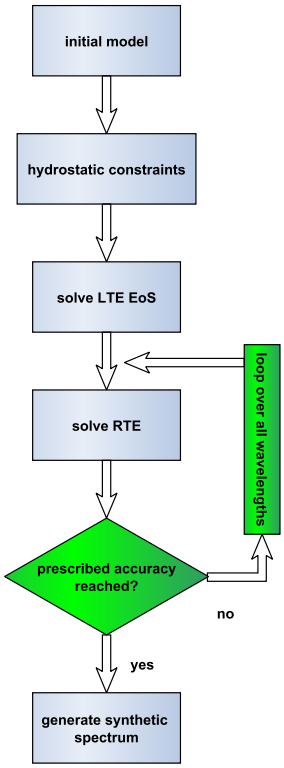


Figure 3.2: Concurrent wavelength clusters (based on [See11, 24])

Figure 3.2 displays the general workflow of PHOENIX. The user's input consists amongst other parameters of the atmosphere's luminosity, mass, temperature, pressure and density as well as the velocity field and present elements. Depending on the chosen mode the *fundamental variables* (temperature, gas pressure, population numbers) are calculated for every voxel at first to fulfil constraints set by the hydrostatic and thermodynamic equilibrium [HB99, 3–6]. Then for every wavelength, which shall be part of the synthetic spectrum, the tracker calculates the mean intensity J_λ and the corresponding Flux F_λ for every voxel of the grid. Concluding this, the operator splitting method is used, to calculate a better guess for the mean intensity. This cycle, which is denoted as the *wavelength loop*, is done repeatedly until the desired accuracy has been reached.

In case that only the functionality of a tracker shall be tested, PHOENIX offers the possibility to execute only the segment, where the 3D radiative transfer equations are solved and constructs for this purpose a so-called *toy atmosphere*, which provides every initial value with that the wavelength-loop is executed. The toy atmosphere assumes that only one kind of atom with two energy levels is present [See08, 19]. Figure 3.3 displays the actual program flow of PHOENIX for this case. After basic initialisations, the model parameter are read and the parallelisation scheme is built (cf. figure 3.4). After creating the grid and generate fictitious input data, which would be calculated or read in by PHOENIX otherwise, the actual tracker is executed to calculate the mean intensities.

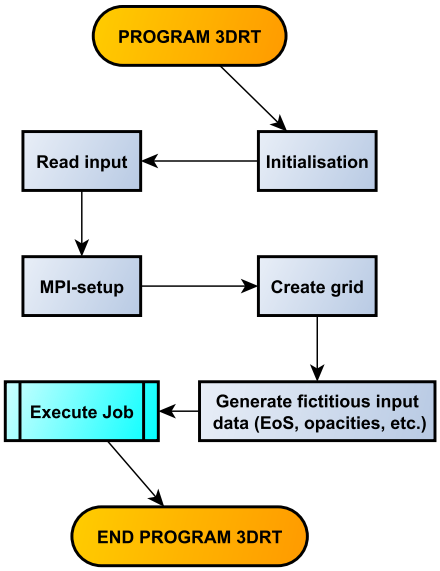


Figure 3.3: Program flow of PHOENIX's test program to check the 3DRT mode

3.3 Parallelisation pattern

```
phoenix:
  nvoxel = number of voxels in total (e.g from structure)
  proces_per_wl = DDS (choose this)
  nwl = number of wavelength points in total
  nwlc = n/DDS = number of wavelength clusters
-------------------------------------------------------

! Initialise MPI structure and communication. Sets e.g wl_cluster_id for each processing unit

init_mpi()

! Split tasks: Each wavelength cluster only calculates a set of wavelength points < nwl
do i=0, nwl-1, nwlc :

  phx_wl_cluster_tasks(wl=i+wl_cluster_id)(nvoxel)

    ==>
    ! Split tasks: Each member of the wavelength cluster calculates and holds data only for a fraction of 1/DDS of all voxels, communicating whenever necessary

    do j=0, DDS-1, 1 :
```

```
21
22        phx_wl_cluster_tasks(wl=i+wl_cluster_id)(j*DDS : (j+1)*DDS-1)
23
24          ==>
25          ! Split tasks: SMP supported routines/sections will fork ↙
    ↳ and join thread-teams
26          [ ... ]
```

Listing 3.1: General parallelisation scheme in PHOENIX [AWH16]

The parallelisation scheme, which is currently used in Phoenix, is depicted by listing 3.1. There are several code segments, which are parallelised, but PHOENIX can basically be divided into three different parallelisation stages using MPI or OpenMP respectively. Figure 3.4 exhibits these crucial three parallelisation stages, which are discussed below:

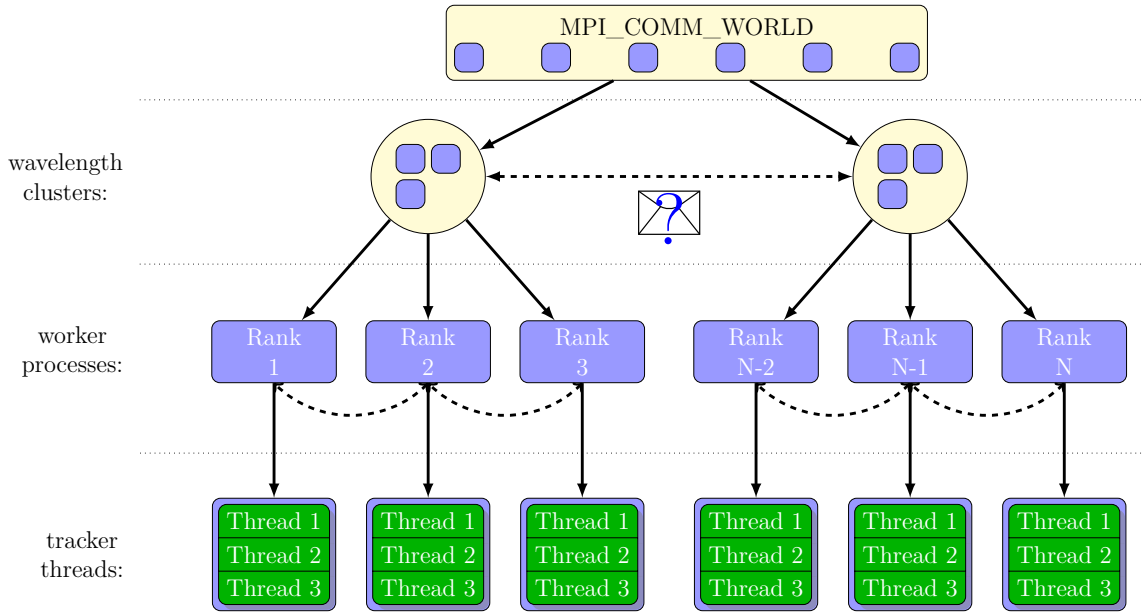


Figure 3.4: Visualisation of the three parallelisation stages of PHOENIX

1. Wavelengths are distributed among groups of processes into so-called wavelength clusters. Every cluster holds the whole data set so the maximum number of processes per cluster is limited by the memory size.

   Given a static model these wavelengths are physically uncoupled, therefore, this provides an easy possibility for data parallelisation, which is initialised in line 13 of listing 3.1. If a dynamic model with a global velocity field in a Lagrangian system is used, the wavelength clusters are dependent on the result of the prior wavelength - it depends on the velocity field which wavelength is the prior one. To execute the calculations on each wavelength cluster still concurrently they are added to a pipeline using a simple round robin scheduler: Wavelength cluster i of a total of n is responsible for the wavelengths $k \cdot n + i, k \in \mathbb{N}$. Figure 3.5 shows a pipeline with

three wavelength clusters, which can be executed concurrently and allow optimal speedup. To keep it simple, a wavelength cluster can be supposed to consist of two phases: one calculation phase `Calculate` λ_i, where the radiative transfer for the current wavelength is computed. This phase cannot be executed concurrently to another calculation phase of another wavelength cluster, because every calculation phase relies on the prior wavelength value. The other phase is the `overlap`. Anything that happens in this phase is independent of any other wavelength cluster and can therefore be executed concurrently to any other phase. The bigger the overlap portion is compared to the calculation phase, the more wavelength clusters can be added to the pipeline without any disadvantages regarding the speed-up. The pipeline is considered full, if adding one more wavelength cluster would result in idling respectively in a non-optimal speedup; figure 3.6 gives an example, where adding one more wavelength cluster impairs the speedup, because wavelength cluster 0 must wait for the result of the prior wavelength, which is calculated by wavelength cluster 3, by what every process becomes delayed - idling phases appear. The exact amount of wavelength clusters, which fit into a pipeline, depends on the underlying hardware and the ratio between the calculation and overlap phase [HB99, 20–21].

After all the calculations have been finished, the results of every wavelength cluster are brought together again to form the synthetic spectrum.

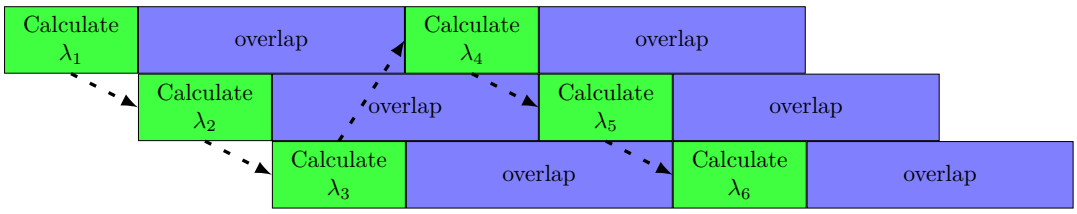


Figure 3.5: Concurrent wavelength clusters with optimal speed-up

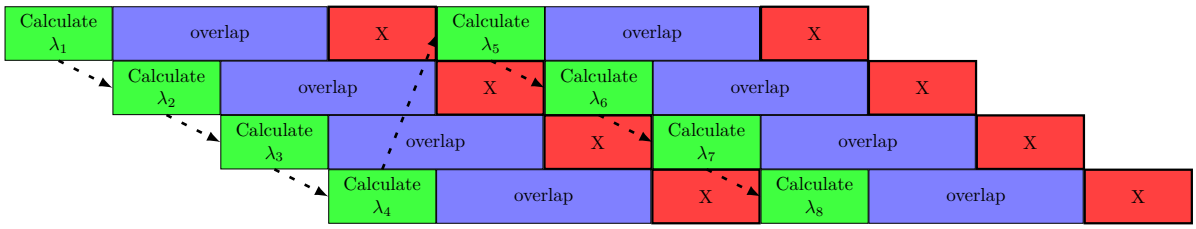


Figure 3.6: Too many wavelength clusters in the pipeline induce idling

2. The voxels of the grid are divided evenly into sections in line 20 of listing 3.1, which are then distributed among all processes of one wavelength cluster. To perform the necessary calculations, the processes might need to communicate with each other sometimes.

3. At the time, when this last parallelisation phase is executed, every process calculates the mean intensity on a selection of solid angles by invoking an appropriate tracker.

To speed this procedure up, some LC-trackers contain OpenMP-sections, which spawn threads to parallelise calculations alongside the long characteristics in line 25 of listing 3.1.

The actual tasks, which are assigned during each phase may differ from the prior description, as they depend on the mode, in which PHOENIX is executed. But for every mode the general parallelisation pattern remains the same [HB99, 17–21].

An example for a LC-tracker, which utilises the last parallelisation step, will be presented in **Design approach** (p.74) and **TrackerSim** (p.61). In these sections the thread parallelisation with OpenMP will be compared and finally substituted with an MPI-3 RMA approach. This will change the hybrid character of PHOENIX into a MPI-only parallelisation model.

3.3.1 3D radiative transfer

To test a LC-tracker it is not necessary to perform a complete run of PHOENIX; instead it is possible to just execute a small test application, which invokes a selected tracker. As the LC-tracker are the only modules in PHOENIX, which utilise OpenMP, it is sufficient to make use of the 3DRT-test application. The parallelisation scheme of the 3DRT-test deviates slightly from the parallelisation scheme of PHOENIX, and shall therefore be explained separately. Again the user input determines how the program flow precisely ends up, so the following enumeration is only one possible sequence and includes the execution of `LC_Lstar_tracker_PBC_zmap` LC-tracker.

1. Assign every process to the same wavelength cluster
2. generate fictitious data
3. Job execution: Iterate over all solid angles
   a) Invoke LC-Tracker with current solid angle
      i. Iterate over every voxel to determine LC's
      ii. Iterate over all LC's to calculate mean intensities, flux components, etc.
4. Check the correctness of the results
5. Collect the results of every process
6. Output the finished results

```
DO i_theta=1,ntheta
  DO i_phi=1,nphi
    [ ... ]
    MPI_counter = MPI_counter+1
    IF( mod(MPI_counter, numProcs) .ne. myRank) cycle
    [ ... ]
```

```
7     END DO
8   END DO
```

Listing 3.2: Load balancing of the solid angles executed by every process

To balance the load every process is provided with different solid angles, listing 3.2 shows the idea behind this distribution. Every process increments its `MPI_counter` and only performs calculations on a solid angle, if its index is a multiple of the process' rank - the solid angles are distributed in fixed turns. Additional parallelisation can be performed through using threads: `LC_Lstar_tracker_PBC_zmap` uses OpenMP to parallelise the loops over all voxel and LC's.

3.4 Profiling

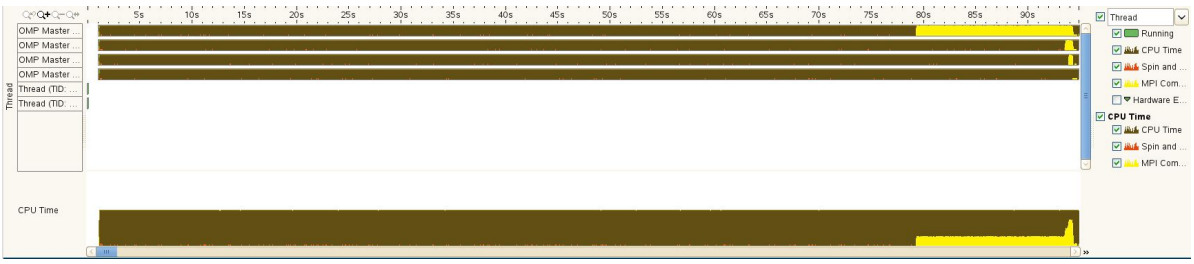


Figure 3.7: Vtune parallelisation visualisation

Examining 3DRT through a explorative data analysis shows, that it makes use of parallelisation very well and reaches a good performance on HPC-systems. A reason for this is, among other things, that the effective CPU time depends primarily on the calculation of mean intensities and how often the Λ^*-iteration scheme needs to be repeated to reach a specified accuracy. Aside from the latter, every item can be parallelised very well (cf. prior section Parallelisation pattern). Figure 3.7 presents a run with four processes and no OpenMP-threads on `Linux-PC` (cf. **Evaluation** (p.79)). Yellow identifies waiting phases, in which MPI communication is executed and no computations are performed, brown marks (desirable) computation. It is quite evident, that the hardware is used with high efficiency and barely any idling occurs due to locks or sequential executions. Only in the end some overhead is visible which might be induced by the collection of the results. This conforms to [HB99, 18–20], where the actual speedup of `PHOENIX`, which was executed with a static model of an atmosphere, achieved about 80% of the theoretical maximum speedup.

[HBA97] examined the performance of `PHOENIX` regarding strong scaling (cf. **Scaling** (p.83)). For that purpose a small test case was constructed with about $60 \times 60 \times 300$ voxels and 64×64 spatial directions per voxel and then executed on the same hardware with up to 2048 processes. Even though the scaling efficiency sometimes decreased to about 80%, it performed quite well. The authors even suppose that the scaling efficiency will be still satisfactory, if the test case is executed on up to 256000 processes and the

underlying hardware can handle so many processes adequately. From this it follows, that provided with a suitable problem size, PHOENIX can be deployed on nearly any HPC system.

Beside these advantages regarding parallelisation and scaling, PHOENIX has one major disadvantage: Many problems, which are solved with PHOENIX, are very reliant on memory. The numerical calculation of the synthetic spectrum requires to construct Λ^* for every voxel of the grid. Constructing one Λ^* makes it necessary to store a so-called Λ^* element for itself and all adjacent voxels, a total of 27 elements. The demand for storage by Λ^* could be reduced to only itself plus six adjacent elements, though this would result in a slower convergence rate. Therefore it scales like $\mathcal{O}(n^3)$, in which n denotes the total number of voxels [HB06, 4–5].

In the prior section **Parallelisation pattern** (p.46) the communication pattern was described, which occurs in the case of a dynamic model and limits the possible amount of wavelength clusters, that fit into the pipeline at once. This limit does not occur in 3DRT but in PHOENIX; there it can be bypassed if more processes are spawned during the second parallelisation phase (cf. figure 3.4), because the increased amount of processes enlarges the content size of each wavelength cluster, which do not require the result of the prior wavelength. But even if the underlying hardware may support additional processes, the number of processes per node increases [HB99, 17–21]. More processes per node lead to a worse $\frac{memory}{process}$ ratio; as a consequence the problem size is even more limited, that fits into a process' memory [AWH16]. One relaxation of this restriction is to lower the number of processes (which causes the problem of a full pipeline again) or to reduce redundancy. Because one wavelength cluster may contain more than one node, this approach would require message passing, so OpenMP cannot be used. RMA could create a global memory instead, which involves every node of a wavelength cluster and therefore allows to dispose of redundant data at the price of an increased amount of communication. Section **Use cases** (p.20) gives an outline of reducing redundancy with RMA.

As already mentioned some LC-tracker make use of OpenMP. This leads to a hybrid-model for better or for worse. The next section **Applying MPI** (p.52) describes a general approach to remove all OpenMP sections and replace them with equivalent RMA operations. An in-depth discussion is made in chapter **Using MPI-3 RMA** (p.60). The purpose of this refactoring is, that it might improve PHOENIX in two ways:

1. Performance: The advantage of OpenMP, that it is very easy to integrate in already existing code, comes at the price, that compared to other parallelisation paradigms an user has only little influence on how the parallelisation is performed. This might lead to synchronisation or locks, which are for some applications more conservative than necessary. Using MPI-3 RMA the user needs to take care of right synchronisation himself; while this leads to a major amount of additional work, the user has now the possibility to omit synchronisation, which is in a concrete case not necessary. Given a suitable application, this could result in a better performance. Section **TrackerSim** (p.61) deals with the question, whether PHOENIX is such a suitable application.

2. Expandability: Another advantage of MPI RMA over OpenMP is, that OpenMP is limited to shared memory; in most cases OpenMP cannot be used for internode communication. As RMA is not dependent on shared memory, it is able to perform both internode and intranode communication. In this regard and at this juncture OpenMP is sufficient for PHOENIX but the introduction of RMA prepares it for internode communication, if it is required some day.

3.5 Applying MPI

PHOENIX makes use of many derived types; one of those is a large derived type denoted as pve_data. Among many other things it stores the Λ^* elements for every voxel and has therefore an eminent impact on the performance. To make use of task parallelism many LC-tracker of PHOENIX share variables through OpenMP. For this thesis the tracker LC_Lstar_tracker_PBC_zmap was chosen to demonstrate the potential of RMA to substitute OpenMP. To create a MPI-only version of this tracker, every OpenMP-section is replaced with a RMA approach - instead of tracker-threads *tracker-processes* are now spawned dynamically and it would also be possible to use shared memory again. Because memory is a valuable asset and pve_data might rely heavily on it, it must not be stored redundantly on every process' local memory by all means.

After sharing the pve_data variable the tracker-processes need to make use of with their access and update operations in the next step. There are some directives (e.g. atomic or critical) in OpenMP which announce operations on memory, which need to be executed with particular attention. Obviously, these operations need to be treated individually by tracker-processes. Indeed there are more operations, which are not explicitly announced to such an extent, but still need individual treatment by MPI RMA: OpenMP automatically takes care, that every load or store operation by a thread on a variable, that was part of a shared-clause in an OpenMP-directive, is executed concurrently in the right way - the user does not need to take action. This behaviour needs to be imitated by the user explicitly, if MPI RMA is used. Therefore every operation on a variable with RMA communication operations or local store and load operations need to be protected by the proper invocation of RMA synchronisation operations, if the variable is part of a shared window, i.e. the associated window has been created with MPI_WIN_ALLOCATE_SHARED.

Section **TrackerSim** (p.61) analysis different RMA approaches to substitute the OpenMP-sections and determines which approach has probably the best chance, that PHOENIX might benefit from the substitution. After actually choosing an approach in section **Concluding discussion** (p.73) the following section **Substitution of OpenMP** (p.77) describes how exactly this design is implemented in PHOENIX.

4 Intel Xeon Phi

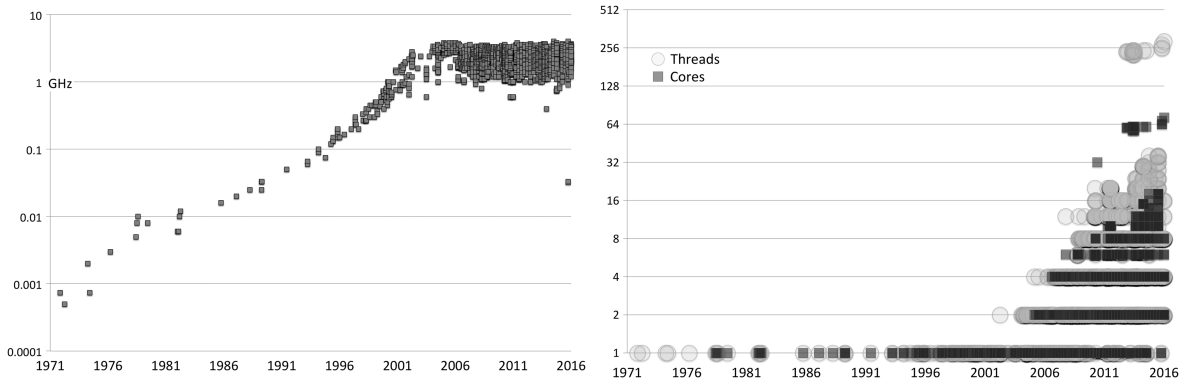
Except for a few CPUs, which reach clock rates of 8GHz and above by overclocking [CPU] but are not intended for an actual deployment, the maximum clock rate of a CPU remains relatively static since the last decade, figure 4.1a visualises this development. This might be explained by the fact, that increasing the clock rate above the current status quo leads to especially three problems:

1. The fraction $\frac{\text{Cost}}{\Delta \text{ clock rate}}$ increases for higher clock rates because higher clock rates entail a disproportionately higher power consumption.

2. The higher the clock rate of a CPU is, the higher becomes the amount of heat, which is emitted and needs an adequate handling in terms of cooling.

3. In order to increase the clock rate, among other things, the density of transistors on the CPU's surface needs to be enhanced. Aside from the advanced complexity this will lead to quantum effects if the structure size goes below a specific value (about $10nm$), which requires a special treatment.

These problems do not imply that it is not possible to further increase the clock rate, but it will not get any easier either and there are more economic possibilities to achieve the same performance. In answer to the general stagnation of the maximum clock rates of CPUs, the portion of core and thread parallelism has increased to compensate this trend, see figure 4.1b. Intels enlarged its product range and released the first generation of the *Intel Xeon Phi Coprocessor* in 2012, which offers massive core parallelism.



(a) ... CPU clock rate [JRS16]

(b) ... core and thread parallelism [JRS16]

Figure 4.1: Development of ...

The next section describes the functionality of the first generation of the Intel Xeon Phi in general terms; the current generation and its improvements are then presented in the succeeding section **Knights Landing** (p.54). In the end the possibility to build a cluster out of Intel Xeon Phis is displayed in section **Xeon Phi cluster** (p.57) and section **Suitability for Phoenix** (p.58) discusses, if it is suitable to run `PHOENIX` on the Intel Xeon Phi.

4.1 Overview

The first generation of the Intel Xeon Phi, which is refered to as *Knights Corner* and was available as a PCIe card, allowed to make use of up to 61 cores with four threads and one large 512bit vector processing unit (VPU) per core. The Xeon Phi has been designed to offer the advantages of an accelerator (like GPUs) without adopting its disadvantages: Usually a program needs to be partly rewritten before it may be executed on an accelerator. Only minimal or even no changes are needed to execute a program on the Xeon Phi, which has three different operating modes:

Native: The application is executed completely on the coprocessor. Every library, that the application makes use of, needs to be crosscompiled for the Xeon Phi and uploaded beforehand.

Offload: The application can be extended with offload pragmas and is executed on the host. Every code section, which is enclosed by offload pragmas, is automatically loaded and executed on the Xeon Phi. The results are loaded back to the host, the execution on the host halts as long as the offload section is executed.

Symmetric: The application needs to be compiled for the host as well as for the Xeon Phi and may then be executed concurrently.

In order to max out the potential capacity of the Intel Xeon Phi an application fulfils ideally two characteristics: On the one hand it scales with hundreds of threads and on the other hand it benefits from a large vector unit. Each HPC application should at least satisfy one criterion. In principle each application, which benefits from an execution on a GPU, should also benefit from an execution on a Xeon Phi. The reason to prefer the Xeon Phi to a GPU is that a Xeon Phi has a broader applicability: It supports amongst others C/C++, Fortran, OpenMP and MPI as well as any library that can be crosscompiled for the Xeon Phi.

A more elaborate description of the first generation of the Intel Xeon Phi is available in my prior thesis [Squ14, Ch. 2 – 3] and shall, therefore, not be repeated in this chapter.

4.2 Knights Landing

The second and current generation of the Intel Xeon Phi is referred to as *Knights Landing* (hereafter: KNL) and is a backwards-compatible further development of Knights Corner.

This section is confined to the new features of KNL and is mainly based on [JRS16], which offers additional in-depth descriptions and explanations.

Figure 4.2: Intel Xeon Phi Knights Landing [Sau16]

The main difference between Knights Corner and KNL is the fact that KNL is available again as a PCIe card (Knights Landing coprocessor) but also as a compatible x86 chip (Knights Landing processor). The socket version does not require a host system like the coprocessor. Using the KNL processor offers directly following advantages:

- No application or library must be ported to be executed on the Xeon Phi.
- The PCIe connection drops out as a bottleneck

In addition KNL offers a possibly higher performance than Knights Corner. Depending on the actual model, Knights Corner offered a maximum performance of about 1 TFLOPS double precision whereas KNL is theoretically capable of providing a maximum performance of about 3.5 TFLOPS double precision. Knights Corner needs to execute multiple threads per core to reach its maximum capacity but Knights Landing can because of hyper-threading already reach its maximum performance with one thread per core.

For the benchmarks in section **Evaluation** (p.79) the socket version of the KNL was used, therefore the following description focuses rather on this variant than on the PCIe card variant.

4.2.1 Architecture

KNL introduces a new vector unit with a capacity of 512 bits, which is called *Intel AVX-512* - each core is assigned with two AVX-512 units. An application may only utilise the KNL at full potential performance if it is able to operate the vector units of

each core at full capacity. Two cores together with both their AVX-512 vector units and a shared 1MB L2 cache are combined in a so-called *tile*, which is shown in figure 4.3. Tiles are connected with each other through the CHA (caching/home agent), which is also part of a a cache-coherency protocol.

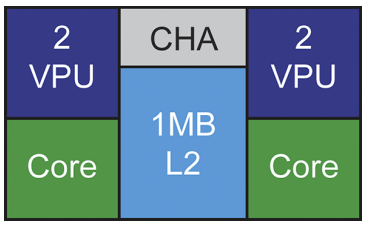


Figure 4.3: Composition of a tile [JRS16]

Up to 36 tiles[1] are combined in a grid like shown in the schematic 4.4 to build a KNL. In contrast to Knights Corner, which uses a bidirectional 1D ring interconnect, a 2D mesh interconnect is used for KNL to provide a higher bandwidth and lower latency.

Knights Corner has only 8GB internal memory available in total, which has to provide memory for both the RAM and file system storage. This limits the possible problem size, which can be computed on Knights Corner and makes the offload mode preferable. The KNL processor lifts this restriction and makes use of two different types of memory:

DDR: Common DDR memory allows to connect the KNL over six DDR4 channels with up to 384GB.

MCDRAM: [2] 16GB high-bandwidth memory, which is distributed over eight devices around the chip, which surround in figure 4.2 visibly the chip. MCDRAM can be used in three different setups, which can be chosen in BIOS.

Cache: MCDRAM works as L3 cache between the tiles and DDR memory.

Flat: MCDRAM expands the common DDR memory by 16GB and blends into the DDR address space, so the MCDRAM memory becomes visible for applications. A KNL coprocessor can only choose this mode because as a PCIe card it has no DDR support and therefore the MCDRAM must act as the main memory.

Hybrid: The available MCDRAM memory is split up into two sections, which are set to cache respectively to flat mode.

In addition the three different MCDRAM modes there are in addition five cluster modes, which control the view of tiles on any memory address of DDR or MCDRAM. It does not restrict the accessibility of tiles to memory or its cache coherency, but influence the address affinity of memory regarding certain tiles respectively if the access to memory is uniform or non-uniform. Since *hemisphere* and *SNC-2* are only variations of *quadrant* and *SNC-4* and subdivide the mesh into only two regions instead of four, they are therefore not discussed separately.

- quadrant: This is the default mode and sets the memory type of DDR and MCDRAM to UMA. Applications which only execute one process per KNL and than make use of OpenMP threads to fully load the local cores, should use this mode.

[1] The number of enabled tiles varies
[2] Multi-channel DRAM

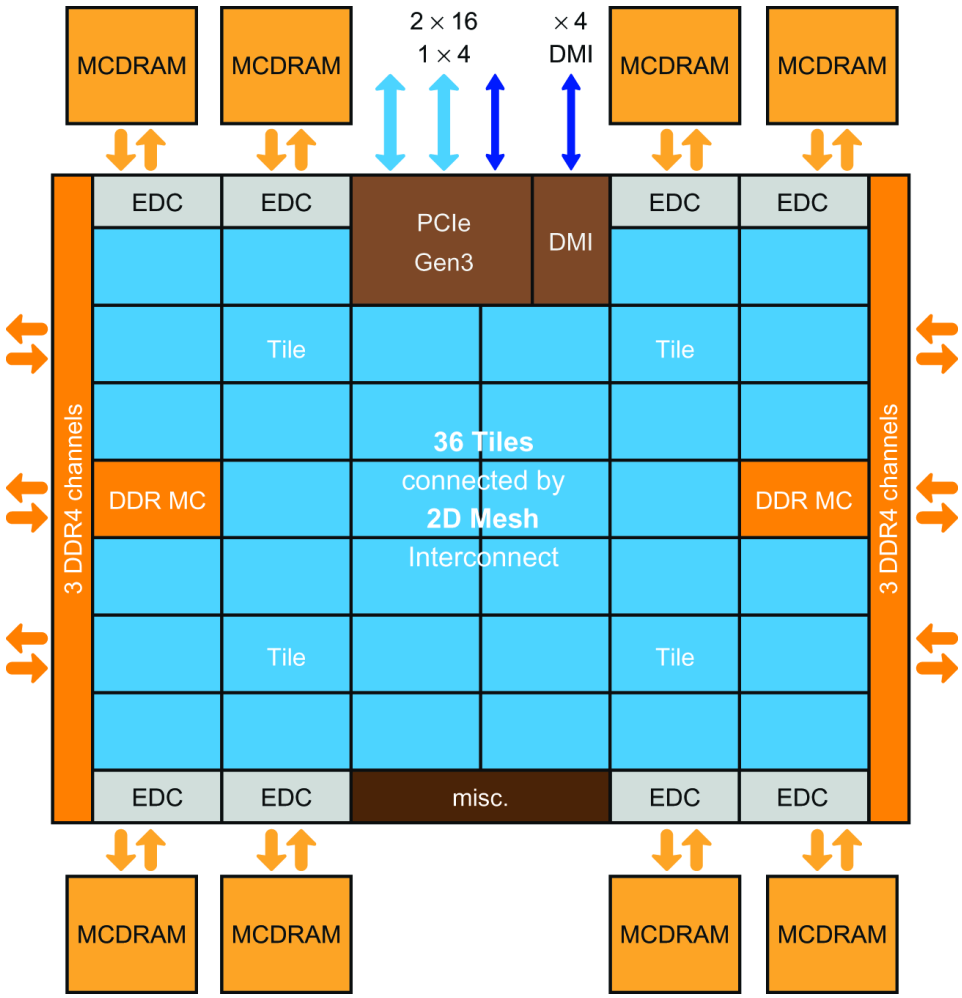


Figure 4.4: Schematic representation of the Knights Landing architecture [JRS16]

- SNC-4: The memory type is set to NUMA, the tile mesh is subdivided into four quadrants. Tiles access memory of tiles in the same quadrant faster than if the other tile is in another quadrant. This type is recommended to choose if four or multiple of four many processes shall be executed on KNL.
- all-to-all: If both prior modes cannot be chosen because the memory modules do not have an identical capacity, this mode is used. It does not set any affinity between tiles and memory and offers, therefore, the worst performance of all modes in general.

See [JRS16, Ch. 3 – Ch. 4] for more detailed explanations of cluster modes.

4.3 Xeon Phi cluster

Every HPC application should be able to make use of HPC architecture with massive parallelism. So using accelerators, which offer high performance through large vector

units and/or the possibility to execute many processing elements concurrently, is a common strategy. Achieving the same performance without accelerators usually results in higher costs and power consumption. Because Intel Xeon Phi offered the performance of an accelerator without many of their disadvantages it was a self-evident choice to use for HPC systems, but Knights Corner had some flaws, which impeded a wide distribution. The KNL processor tackles the most important flaws and is, therefore, short-listed to be used in HPC systems.

Using a coprocessor card requires a host system and raises automatically the question, what task the host system should undertake during the execution of a application. It could be used for I/O only but it could also participate in the computations, which makes load balancing necessary to achieve optimal performance. Building a cluster out of KNL processors means that no additional host CPUs are needed and one gets a homogeneous system; therefore, load balancing between the Xeon Phi processor and a host processor is rendered unnecessary.

Already 75 systems of the Top500 list[1] (November 2014) list use accelerators (GPUs, Xeon Phis, etc.) [LHL+15] and could benefit from KNL processors. For example Stampede[2] makes use of Knights Corner and it was recently announced to deploy Stampede 2, which shall make use of Knights Landing [SV16]. Another example for a cluster, which makes use of Knights Landing, is NERSC-8[3]

Additional optimisation for a cluster of Xeon Phis may be achieved through the usage of MPI RMA. With a single Xeon Phi one gets along with using threads to use it at full capacity. But a cluster of several Xeon Phis makes internode communication necessary, which can for example be realised with PGAS or MPI. [BL15] shows an approach to use MPI-3 RMA on a Xeon Phi cluster, which automatically determines if the target requires internode or intranode communication and makes then use of common two-sided MPI communication or MPI RMA communication on shared memory. Instead of two-sided MPI communication one could of course use one-sided MPI RMA communication, too. With this approach they gained a speedup of up to 4.7 on one Xeon Phi node and a speedup of up to 1.8 on two Xeon Phi nodes in comparison to an approach, which only makes use of MPI two-sided communication. Another benchmark on a Xeon Phi cluster by [LHL+15], which applied MPI-3 RMA to the used benchmark Graph500[4], achieved an improvement of 25% in performance.

4.4 Suitability for Phoenix

Using VTune (cf. figure 4.5) to analyse a run of `PHOENIX` with four processes and two threads per process on Linux-PC shows that it is about half latency bound and only one-fifth bandwidth bound, so `PHOENIX` needs very frequently to reload few data into the cache and does not use the available memory bandwidth to its full capacity. To

[1] `https://www.top500.org/`
[2] `https://www.top500.org/system/177931`
[3] `http://www.nersc.gov/users/computational-systems/cori/`
[4] `http://www.graph500.org/`

make use of this situation it is recommended to spawn more threads, which allows a core to utilise the time, one thread must wait for the arrival of the requested data, for computations with another thread. The target of spawning more threads is to pipeline enough data requests to use the memory bandwidth at full capacity; thereby the amount of computations per time interval and consequentially the resulting performance increases.

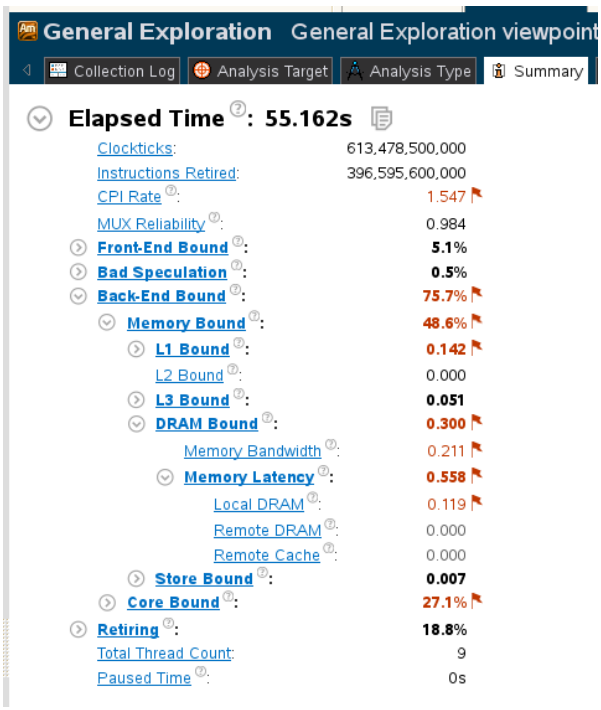


Figure 4.5: VTune: Memory bandwidth and latency

KNL has eight blocks of MCDRAM, which offer about four times more bandwidth than common DDR4 memory [P.16]. The combination of a high bandwidth and the many available cores of KNL allow to utilise that PHOENIX is primarily memory latency bound by spawning more threads for its execution - the total performance possibly increases.

If PHOENIX is not capable to make adequately use off additional threads, the amount of computation operations per time interval does not increase. Then the performance does not increase neither. As discussed in section **Profiling** (p.50) PHOENIX scales very good with additional processing elements if it is provided with a proper problem as input. PHOENIX should therefore benefit if it is executed on KNL.

5 Using MPI-3 RMA

After discussing a general approach to make use of MPI-3 RMA in PHOENIX in the former section **Applying MPI** (p.52), this chapter focuses on the practical implementation. For the sake of brevity and to concentrate on the LC-tracker, all necessary changes are applied to 3DRT, which mimics the behaviour of PHOENIX. Every finding allows inferences about PHOENIX which is why PHOENIX and the 3DRT test application may be used synonymously in this chapter.

Since MPI-3 added many new operations to the MPI RMA interface, various concepts will be discussed and also tested in section **TrackerSim** (p.61). Given the large dimension of PHOENIX in general and of the LC-tracker LC_Lstar_tracker_PBC_zmap (hereafter only denoted as the *LC-tracker*) in particular, the potential of each RMA approach is tested in a much smaller test program: TrackerSim. Its purpose is to mimic important parts of the LC-tracker, so that each RMA approach does not need to be fully implemented in the LC-tracker. Instead of that each approach is fully implemented in TrackerSim to estimate whether is is beneficial respectively the performance.

The LC-tracker contains three OpenMP-sections, that are reproduced in TrackerSim too. Instead of adopting every operation, which is invoked in a OpenMP section of the LC-tracker, a selection is made: Only those operations, which are crucial or representative for a OpenMP section are adopted in TrackerSim. Any optimisation, which is only available in TrackerSim but not possible in the LC-tracker, was declined. Thereby the implementation effort is highly reduced but the benchmark results are still reasonably conferable to the LC-tracker.

$\texttt{TrackerSim}_{\text{SMP}}$ denotes the variation of TrackerSim, which still uses OpenMP and is basically a compacted version of the LC-tracker. Any other variation of TrackerSim uses this variation as initial point to implement its selected RMA approach. The correlation between the performance of $\texttt{TrackerSim}_{\text{SMP}}$ and the performance of any other variation of TrackerSim should arise again in the comparison of the performance of the LC-tracker with OpenMP and the LC-tracker with MPI-3 RMA.

The following section describes TrackerSim in detail and investigates the best approach to replace OpenMP with MPI RMA in PHOENIX. Because it is not enough to replace the OpenMP sections with MPI-3 RMA operations only, section **Design approach** (p.74) gives an overview about additional adjustments that need to be integrated into PHOENIX to realise the refactoring. The last section **Substitution of OpenMP** (p.77) gives a concluding discussion about the actual implementation in PHOENIX.

5.1 TrackerSim

Without combining different synchronisation methods there are five reasonable possibilities to rebuild the OpenMP sections with MPI RMA (cf. **Variation 1: fence-synchronisation** (p.65) for a short discussion of the skipped possibility). Since the LC-tracker and 3DRT are quite extensive it would be very time consuming to implement every possibility already in `PHOENIX` and benchmark every version then. Therefore `TrackerSim` was built, which mimics the OpenMP sections with much less code which allows faster implementations and benchmarks. Based on the benchmark results the most promising approach is then chosen to be applied to `PHOENIX`. The next section **Variation 0: SMP** (p.63) builds the basic framework of `TrackerSim` (hereafter: $\texttt{TrackerSim}_{\text{SMP}}$) upon which the other variations base - those will be discussed in the subsequent sections and source code of the most important sections will be shown. Figure 5.2 displays the general programflow of `TrackerSim`, each step may be analysed independently of the other steps.

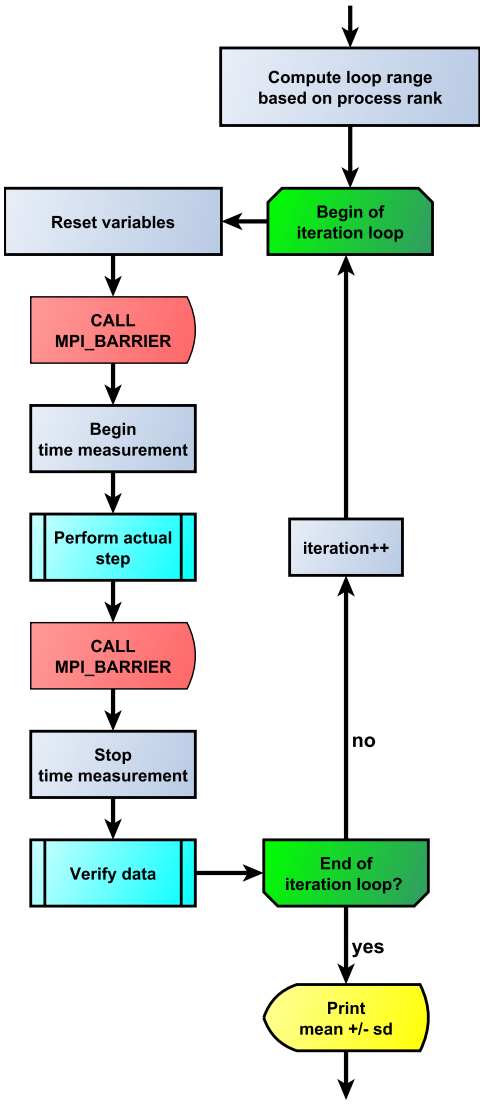


Figure 5.1: General scheme of the execution of one step

In every variation only process 0 (also denoted as `masterProc`) holds memory for a window, every other process invokes the creation operation with `size == 0`. In this way the behaviour of `PHOENIX` can be mimicked as close as possible (in `PHOENIX` each worker process holds the memory and spawns tracker-threads, which use this memory). As the RMA synchronisation operations do not act like normal locks but begin or end epochs (exempt passive locks with `lock_type == MPI_LOCK_EXCLUSIVE`, which are similar to a common lock) the RMA synchronisation operations are placed outside of the loop (cf. section **Scaling** (p.83) to find out, why this positioning should not be chosen for `PHOENIX`). This resulted to be a practice with best performance from benchmarks, which compared both possible placements without additional load: either outside the loop or next to the RMA communication operations nested inside the loop. The latter resulted in loss of performance due to overhead of too many invocations.

Every variation reproduces the behaviour of OpenMP in $\texttt{TrackerSim}_{\text{SMP}}$ true to the original with MPI RMA; therefore every optimisation was omitted, which would only be possible in `TrackerSim` but not `PHOENIX`, because this would have interfered

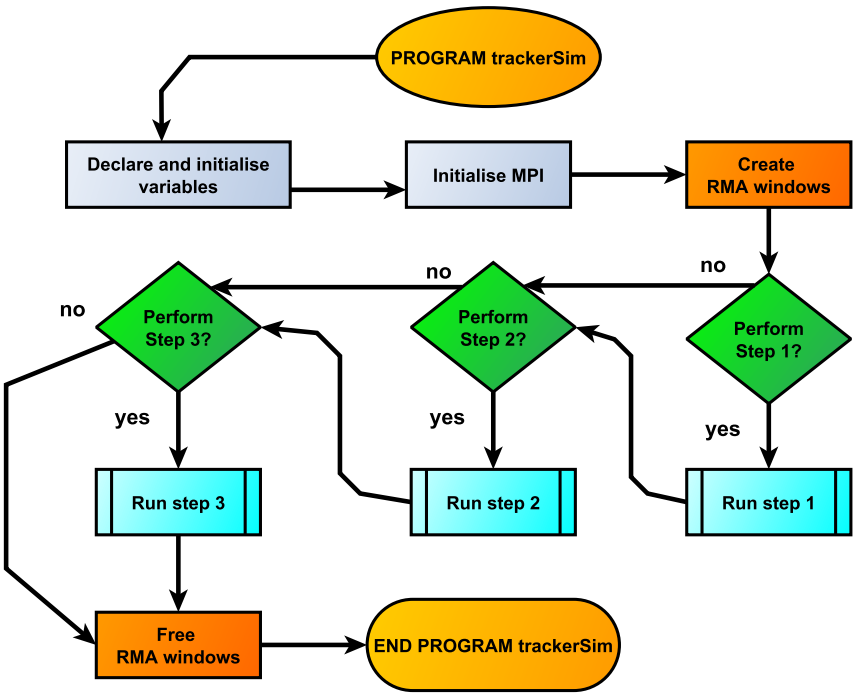


Figure 5.2: General workflow of `TrackerSim` irrespective of the chosen variation

with the assignability of the deductions drawn from the benchmarks of `TrackerSim`. Also no assertions or hints were used in `TrackerSim`, which depend heavily on the underlaying hardware and chosen MPI implementation - this simplifies the comparison of the benchmarks but leaves room for further optimisations.

To assess the performance of every variation, they are compared with $\texttt{TrackerSim}_{\text{SMP}}$ in the end. The following items describe the used methodology in general and are partly displayed in figure 5.1:

- Process 0 is responsible to take the time of every step. To prevent that other processes are already computing, a barrier is set directly before the start and the end of taking the time.
- To firm the statistical validity of the time measurements every timing is repeated five times. To calculate the mean and standard deviation the fastest and slowest time measurement are neglected.
- The creation of windows is excluded from the time measurement, as it happens only once and the focus lays on the communication operations. If the window creation takes a significant amount of time its proportion can easily be reduced be increasing the amount of communication.
- Since the chosen operations, which represent the content of each step, are not the only operations in this step of the LC-tracker, they might be attached with too

much importance. To lessen this effect each step contains a subroutine to generate extra load (`compLast`) - the amount can be chosen by the user. This function generates load while operating on local memory only and may therefore be executed concurrently. The bigger the amount of its execution compared to the rest of the loop is, the bigger is the overlapping portion of one iteration.

- The measured time interval may be extended by increasing the amount of loops to lessen random effects, which might distort the time measurement.
- Any `TrackerSim` variation is executed with as many processes as threads are used in $\texttt{TrackerSim}_{\text{SMP}}$.
- To make sure, that the computations are correct, the results are automatically verified after each iteration: Process 0, which has the whole memory of each shared window, recomputes the results of every without MPI RMA; afterwards it verifies that the newly computed check values conform with the results, which have been calculated with MPI RMA.

There is an alternative to this trial and error approach which relies on a more theoretical construct, that calculates which RMA approach should be fastest. It will be outlined in section **Related work** (p.93).

5.1.1 Variation 0: SMP

Like the LC-tracker $\texttt{TrackerSim}_{\text{SMP}}$ contains three OpenMP sections, which parallelise (nested) loops and allow access and update operations on shared variables. Every section does not necessarily match perfectly its counterpart in `PHOENIX` but examines a special functionality of OpenMP. The operands of the update operations have been simplified, too: either the value "1" or an value is used, which allows to test easily if it is correct (e.g. every array entry could be set to its absolute position).

Step 1: Contains a reduction clause with the +-operator and a critical section from which the result of the prior update operation shall be fetched and stored in `idx`.

```
!$omp parallel do default(none) collapse(3)      &
!$omp   shared(nx,ny,nz,n_chars)                 &
!$omp   [...]
!$omp   reduction(+:reduction)
DO iz=-nz,nz
  DO iy=-ny,ny
    DO ix=-nx,nx
      CALL compLast(lasteinstellung(1))
!$omp critical
      n_chars = n_chars + 1
      idx = n_chars
!$omp end critical
      reduction = reduction + 1
```

```
    END DO
  END DO
END DO
```

Listing 5.1: Abstraction of Phoenix' access pattern with OpenMP (Step 1)

Step 2: Contains the common use case of an atomic counter increment without the need to fetch immediately the result.

```
!$omp parallel do default(none) &
!$omp   [...]                   &
!$omp   shared(zaehler)
DO i=1,n_chars
    CALL compLast(lasteinstellung(2))
!$omp atomic
    zaehler = zaehler + 1
END DO
```

Listing 5.2: Abstraction of Phoenix' access pattern with OpenMP (Step 2)

Step 3: Similar to the prior step a variable shall be updated atomically; this time a function and the variable's original value are used. Furthermore a one-dimensional and a three-dimensional array are accessed

```
!$omp parallel do default(none) collapse(3)     &
!$omp   shared(nx,ny,nz,pve_grid,stelle_max)    &
!$omp   [...]
DO iz=-nz,nz
  DO iy=-ny,ny
    DO ix=-nx,nx
      stelle = 1+(iz+nz)*(2*nz+1)**2+(iy+ny)*(2*ny+1)+(ix+nx)
      CALL compLast(lasteinstellung(3))
!$omp atomic
      pve_grid%speicher3D(ix,iy,iz) = pve_grid%speicher3D(ix,iy,iz)+dble(stelle)
!$omp atomic
      pve_grid%speicher1D(stelle)   = pve_grid%speicher1D(stelle) + dble(stelle)
!$omp atomic
      stelle_max = max(pve_grid%speicher1D(stelle), stelle_max)
    END DO
  END DO
END DO
```

Listing 5.3: Abstraction of Phoenix' access pattern with OpenMP (Step 3)

5.1.2 Variation 1: fence-synchronisation

The first variation is using "common" windows (i.e. created by invoking MPI_WIN_-CREATE) and active, collective fence synchronisation. The variables, which had been accessed and updated directly in $\texttt{TrackerSim}_{\text{SMP}}$ are now covered behind a window handle. This is why RMA communication operations need to be used now.

Step 1: Since the result of the increment operation shall be fetched from only a single datatype, MPI_FETCH_AND_OP may be used. But because a value is returned to idx, which was current before ONE had been added, this operation on idx needs to be repeated afterwards. During the ongoing loop every process updates the reduction variable (here: schritt1Array(2)) locally[1] and is then accumulated, after the loop has finished

```
CALL MPI_WIN_FENCE(0, s1Win, ierror)
  DO iz=startIndex,endIndex
    DO iy=-ny,ny
      DO ix=-nx,nx
        CALL compLast(lasteinstellung(1))
        CALL MPI_FETCH_AND_OP(ONE, idx, MPI_INTEGER, masterProc, 0_MPI_ADDRESS_KIND, MPI_SUM, s1Win, ierror)
        idx              = idx              + ONE
        schritt1Array(2) = schritt1Array(2) + ONE
      END DO
    END DO
  END DO
  IF(myRank .NE. masterProc) CALL MPI_ACCUMULATE(schritt1Array(2), 1, MPI_INTEGER, masterProc, 1_MPI_ADDRESS_KIND, 1, MPI_INTEGER, MPI_SUM, s1Win, ierror)
  CALL MPI_WIN_FENCE(0, s1Win, ierror)
```

Listing 5.4: TrackerSim with active, collective synchronisation (Step 1)

Step 2: Similar to step 1 a variable's value shall be increased. But since in this case the immediate result is not of interest MPI_ACCUMULATE is sufficient.

```
CALL MPI_WIN_FENCE(0,s2Win,ierror)
DO i=startIndex,endIndex
  CALL compLast(lasteinstellung(2))
  CALL MPI_ACCUMULATE(ONE, 1, MPI_INTEGER, masterProc, 0_MPI_ADDRESS_KIND, 1, MPI_INTEGER, MPI_SUM,s2Win, ierror)
```

[1] Even though schritt1Array is associated with a window it may be used like a local variable, as long as no shared window or RMA communication operation is used. Every local store operation only affects the local memory

```
5  END DO
6  CALL MPI_WIN_FENCE(0,s2Win,ierror)
```

Listing 5.5: TrackerSim with active, collective synchronisation (Step 2)

Step 3: In addition to the outer fences every RMA communication operation must be accompanied by a fence inside the loop, too. This is probably inevitable due to the `INTEGER` variable `stelle`: The window handles represent memory, which expects to take variables of type `DOUBLE`. Therefore every usage of `stelle` is preceded by a typecast. The typecast buffer needs presumably to be freed before the next typecast may be performed. This is due to the fact, that even though RMA communication operations write changes to the public window copy, which is synchronised with the private window copy eventually automatically, but all processes, which share the same memory share the same typecast buffer, too - without further RMA synchronisation calls the update operations on the window are still safe but data races happen on the typecast buffer in the background. To ensure, that this buffer is free to be reused, additional fences must be set. Without these fences some values of `stelle` could be omitted. Additional invocations of fences in the end arise from the fact, that fences must be executed collectively: Due to a primitive load balancing the last process needs to resume the last iterations, which could not be distributed evenly. To match the last surplus fences, every other process needs to invoke additional fences in the end (cf. lines 23 – 30)

```
1  CALL MPI_WIN_FENCE(0,s3Win1D, ierror)
2  CALL MPI_WIN_FENCE(0,s3Win3D, ierror)
3  CALL MPI_WIN_FENCE(0,s3WinMax,ierror)
4  DO iz=startIndex, endIndex
5    DO iy=-ny,ny
6      DO ix=-nx,nx
7        stelle = 1 + (iz+nz)*(2*nz+1)**2 + (iy+ny)*(2*ny+1) + ↙
             ↳ (ix+nx)
8        CALL compLast(lasteinstellung(3))
9        s3targetDisp = stelle-1
10 !1D
11       CALL MPI_ACCUMULATE(REAL(stelle,kind(prec)), 1, ↙
             ↳ prec_mpi, masterProc, s3targetDisp,1, prec_mpi, ↙
             ↳ MPI_SUM, s3Win1D, ierror)
12       CALL MPI_WIN_FENCE(0,s3Win1D, ierror)
13 !3D
14       CALL MPI_ACCUMULATE(REAL(stelle,kind(prec)), 1, ↙
             ↳ prec_mpi, masterProc, s3targetDisp,1, prec_mpi, ↙
             ↳ MPI_SUM, s3Win3D, ierror)
15       CALL MPI_WIN_FENCE(0,s3Win3D, ierror)
16 !Maximum
17       CALL MPI_ACCUMULATE(REAL(stelle,kind(prec)), 1, ↙
```

```
            ↳ prec_mpi, masterProc,0_MPI_ADDRESS_KIND,1, ↙
            ↳ prec_mpi, MPI_MAX,s3WinMax, ierror)
18        CALL MPI_WIN_FENCE(0,s3WinMax, ierror)
19      END DO
20    END DO
21  END DO !omp
22  !Last proc uses more Fences, so everyone else needs to match ↙
            ↳ them
23  IF(myRank .LT. numProcs-1) THEN
24    fenceIndex=MODULO((nz*2+1),numProcs)*(2*ny+1)*(2*nx+1)
25    DO i = 1,fenceIndex
26      CALL MPI_WIN_FENCE(0,s3Win1D, ierror)
27      CALL MPI_WIN_FENCE(0,s3Win3D, ierror)
28      CALL MPI_WIN_FENCE(0,s3WinMax, ierror)
29    END DO
30  END IF
31  CALL MPI_WIN_FENCE(0,s3Win1D, ierror)
32  CALL MPI_WIN_FENCE(0,s3Win3D, ierror)
33  CALL MPI_WIN_FENCE(0,s3WinMax,ierror)
```

Listing 5.6: TrackerSim with active, collective synchronisation (Step 3)

Another approach using active target communication is GATS (cf. section **Active target** (p.36)). These RMA synchronisation operations were skipped due to the fact, that their only advantage over fences is that they do not need to be called collectively. As it became apparent during testing that passive target communication offered a higher performance than active target communication, GATS had been omitted.

5.1.3 Variation 2: passive synchronisation

Passive synchronisation with RMA locks allow to realise real one-sided communication (besides the collective window creation). Since every RMA communication operation in every step of `TrackerSim` is an atomic operation, which makes concurrent executions safe, it is sufficient to use `lock_type = MPI_LOCK_SHARED`. To emphasise that RMA locks may not be mistaken for normal locks, figure 5.3 shows an execution of `TrackerSim` with passive synchronisation with shared locks. See figure 6.8b for an example of passive synchronisation with an exclusive lock.

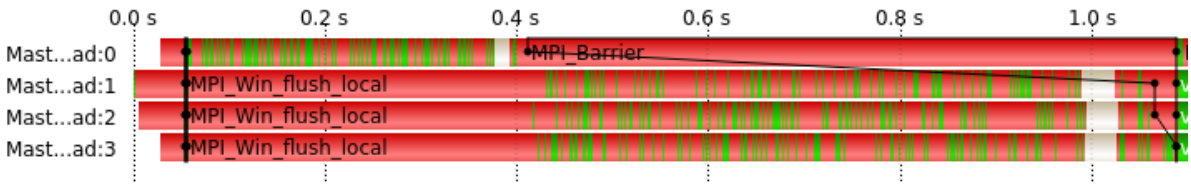


Figure 5.3: Passive target communication with shared locks visualised with `Vampir`

It is clearly visible that although a "lock" is set in front and after the loops, no process is blocked from accessing or updating the window. However the figure shows one disadvantage, too: Despite using `lock_type = MPI_LOCK_SHARED` the invocation of a RMA synchronisation lock by a process functions always as a "real" lock, if the target is the process itself (here: `masterproc == 0`). That is to say that in this case, every time when process 0 (which holds the memory for every window) invokes `MPI_WIN_LOCK(MPI_LOCK_SHARED, masterproc,...)` no other process may access or update these windows. This problem is superfluous regarding `PHOENIX` because the worker process, which spawns the tracker-processes and holds the memory for each window, does not compute concurrently to its child processes on these windows (cf. section **Design approach** (p.74) and listing 5.15).

Step 1: Besides the change to passive target communication the actual RMA communication operations stay the same and are therefore omitted in this listing.

```
CALL MPI_WIN_LOCK(MPI_LOCK_SHARED, masterProc, 0, s1Win, ↙
          ↳ ierror)
DO iz=startIndex,endIndex
  DO iy=-ny,ny
    DO ix=-nx,nx
      [ ...increment and fetch counter... ]
    END DO
  END DO
END DO
[ ... reduction ... ]
CALL MPI_WIN_UNLOCK(masterProc, s1Win, ierror)
```

Listing 5.7: TrackerSim with passive synchronisation (Step 1)

Step 2: Same case like in step 1.

```
CALL MPI_WIN_LOCK(MPI_LOCK_SHARED, masterProc,0, s2Win, ierror)
DO i=startIndex,endIndex
  [ ... increment counter ... ]
END DO
CALL MPI_WIN_UNLOCK(masterProc, s2Win, ierror)
```

Listing 5.8: TrackerSim with passive synchronisation (Step 2)

Step 3: Similar to step 3 of section **Variation 1: fence-synchronisation** (p.65) it must be guaranteed that the typecast buffer may be used again. For this case RMA offers special operations for passive target communication to ensure reusability of used buffers: `MPI_WIN_FLUSH...`. As in this case every RMA communication operation only targets one specific process (process 0) and it must only be guaranteed, that the operation is concluded locally, `MPI_WIN_FLUSH_LOCAL` is an adequate choice.

```
CALL MPI_WIN_LOCK(MPI_LOCK_SHARED,masterProc,0,s3Win1D,  ierror)
CALL MPI_WIN_LOCK(MPI_LOCK_SHARED,masterProc,0,s3Win3D,  ierror)
CALL MPI_WIN_LOCK(MPI_LOCK_SHARED,masterProc,0,s3WinMax, ierror)
DO iz=startIndex, endIndex
  DO iy=-ny,ny
    DO ix=-nx,nx
      stelle = 1 + (iz+nz)*(2*nz+1)**2 + (iy+ny)*(2*ny+1) + (ix+nx)
      CALL compLast(lasteinstellung(3))
      s3targetDisp = stelle-1
!1D
      CALL MPI_ACCUMULATE(REAL(stelle,kind(prec)), 1, prec_mpi, masterProc, s3targetDisp,1, prec_mpi, MPI_SUM, s3Win1D, ierror)
!3D
      CALL MPI_ACCUMULATE(REAL(stelle,kind(prec)), 1, prec_mpi, masterProc, s3targetDisp,1, prec_mpi, MPI_SUM, s3Win3D, ierror)
!Maximum
      CALL MPI_ACCUMULATE(REAL(stelle,kind(prec)), 1, prec_mpi, masterProc,0_MPI_ADDRESS_KIND,1, prec_mpi, MPI_MAX,s3WinMax, ierror)
!Free local buffer
      CALL MPI_WIN_FLUSH_LOCAL(masterProc, s3Win1D, ierror)
      CALL MPI_WIN_FLUSH_LOCAL(masterProc, s3Win3D, ierror)
      CALL MPI_WIN_FLUSH_LOCAL(masterProc, s3WinMax,ierror)
    END DO
  END DO
END DO
CALL MPI_WIN_UNLOCK(masterProc, s3Win1D, ierror)
CALL MPI_WIN_UNLOCK(masterProc, s3Win3D, ierror)
CALL MPI_WIN_UNLOCK(masterProc, s3WinMax, ierror)
```

Listing 5.9: TrackerSim with passive synchronisation (Step 3)

5.1.4 Variation 3: shared memory with RMA communication

Other than both previous variations this variation does not use "normal" but shared windows. Since `TrackerSim` is executed on only one node one may assume the existence of shared memory. Listing 5.10 displays the most important sections to create shared

memory[1]. Only the main process creates shared memory, that will be accessed and updated by every other process, which is why every other process needs to invoke MPI_WIN_SHARED_QUERY to fetch the right memory address (see the description of MPI_WIN_ALLOCATE_SHARED in section **Initialisation** (p.29)). In the last line every process needs to assign the received C-pointer to a Fortran-pointer.

```
USE, INTRINSIC :: ISO_C_BINDING, ONLY : C_PTR, C_F_POINTER
INTEGER                          :: prec_mpi
INTEGER                          :: sizeOfprec_mpi
INTEGER                          :: s3Win3D
INTEGER(KIND=MPI_ADDRESS_KIND)   :: sizeOfSchritt3D3
TYPE(C_PTR)                      :: s3D3Ptr
REAL(kind(prec)), DIMENSION(:,:,:), POINTER :: speicher3DPtr

[ ... MPI-initialisation ... ]

!create new type and get size
CALL MPI_TYPE_CREATE_F90_REAL(KIND(prec), MPI_UNDEFINED, ↙
          ↳ prec_mpi,ierror)
CALL MPI_TYPE_SIZE(prec_mpi, sizeOfprec_mpi,ierror)

IF(myRank .EQ. masterProc) THEN
    sizeOfSchritt3D3 = sizeOfprec_mpi*(container%nxyz*2+1)**3
ELSE
    sizeOfSchritt3D3 = 0_MPI_ADDRESS_KIND
END IF

!create shared RMA window
CALL MPI_WIN_ALLOCATE_SHARED(sizeOfSchritt3D3,sizeOfprec_mpi, ↙
          ↳ MPI_INFO_NULL, MPI_COM_WORLD, s3D3Ptr,s3Win3D, ierror)
IF(myRank .NE. masterProc) CALL MPI_WIN_SHARED_QUERY(s3Win3D, ↙
          ↳ masterProc, sizeOfSchritt3D3, sizeOfprec_mpi, ↙
          ↳ s3D3Ptr, ierror)
CALL C_F_POINTER(s3D3Ptr, speicher3DPtr, (/2*nx+1,2*ny+1,2*nz+1/))
```

Listing 5.10: Example of the creation of a shared window in TrackerSim

In this variation RMA communication operations are still used, even though they are not necessary anymore. Nevertheless they still are atomic operations, so lock_type = MPI_LOCK_SHARED is sufficient again.

Step 1/2: Because only the underlaying window type has changed but RMA communications operations are still used in combination with a RMA passive lock, step 1 and step 2 look exactly like step 1 and step 2 in the prior section **Variation 2: passive synchronisation** (p.67).

[1] It is not necessary to create a new datatype; this was only done to conform with the definition of prec in PHOENIX

Step 3: Like in both previous steps this step uses the same RMA communication operations like step 3 in the prior section **Variation 2: passive synchronisation** (p.67). The only difference between both variations is that because of using a shared memory the flush-operations may be omitted.

5.1.5 Variation 4: shared memory with local communication

Again a shared window is used but this time instead of RMA communication operations local load and store operations are utilised. To make local operations safe, which are executed concurrently to a store operations, `lock_type = MPI_LOCK_EXCLUSIVE` must be used this time. In this case it should be advantageous to pull the locks inside the loops. Even though this entails additional overhead because the locks are more frequently invoked, otherwise no concurrent execution of the step would be possible - the loops are executed sequentially. Since the inner loop offers only little opportunity for concurrent executions (too many `atomic` or `critical` sections) anyway, this variation of `TrackerSim` still performed better than $\texttt{TrackerSim}_{\text{SMP}}$ without any additional load and outer locks(cf. section **Concluding discussion** (p.73)).

Step 1: Every variable, that is associated with a window (here: `schritt1Array`) may be accessed and updated with local load and store operations. Other than the local operations in step 1 of **Variation 1: fence-synchronisation** (p.65) this time these operations affect the window and are therefore visible for every process, which has this window object. It is not necessary to use a temporary variable `reductionTmp` to perform the reduction locally at first, because the exclusive locks are outside. By the time the locks are put inside the nested loops using a local variable before the actual "reduction" in line 12 would make a difference again.

```
1  CALL MPI_WIN_LOCK(MPI_LOCK_EXCLUSIVE, masterProc, 0, s1Win, ↙
          ↳ ierror)
2  DO iz=startIndex,endIndex
3    DO iy=-ny,ny
4      DO ix=-nx,nx
5        CALL compLast(lasteinstellung(1))
6        schritt1Array(1) = schritt1Array(1) + 1
7        idx              = schritt1Array(1)
8        reductionTmp     = reductionTmp + 1
9      END DO
10   END DO
11 END DO
12 schritt1Array(2) = schritt1Array(2) + reductionTmp
13 CALL MPI_WIN_UNLOCK(masterProc, s1Win, ierror)
```

Listing 5.11: TrackerSim with passive synchronisation on shared Memory with local load/store (Step 1)

Step 2: Like mentioned before the window's memory (`zaehlerPtr`) may now be updated with a local assignment.

```
CALL MPI_WIN_LOCK(MPI_LOCK_EXCLUSIVE, masterProc,0, s2Win, ↙
          ↳ ierror)
DO i=startIndex,endIndex
  CALL compLast(lasteinstellung(2))
  zaehlerPtr = zaehlerPtr + 1
END DO
CALL MPI_WIN_UNLOCK(masterProc, s2Win, ierror)
```

Listing 5.12: TrackerSim with passive synchronisation on shared Memory with local load/store (Step 2)

Step 3: The window arrays may accessed and updated locally, too. The fact that the array indices of `speicher3DPtr` look somehow strange in this listing is based on the fact that the corresponding array in `PHOENIX` is allocated with a partial negative range, which could not be reproduced truly to original with MPI RMA otherwise.

```
CALL MPI_WIN_LOCK(MPI_LOCK_EXCLUSIVE,masterProc,0, s3Win1D, ↙
          ↳ ierror)
CALL MPI_WIN_LOCK(MPI_LOCK_EXCLUSIVE,masterProc,0, s3Win3D, ↙
          ↳ ierror)
CALL MPI_WIN_LOCK(MPI_LOCK_EXCLUSIVE,masterProc,0, s3WinMax, ↙
          ↳ ierror)
DO iz=startIndex, endIndex
  DO iy=-ny,ny
    DO ix=-nx,nx
      stelle = 1 + (iz+nz)*(2*nz+1)**2 + (iy+ny)*(2*ny+1) + ↙
          ↳ (ix+nx)
      CALL compLast(lasteinstellung(3))
      s3targetDisp = stelle-1
!1D
      speicher1DPtr(stelle) = speicher1DPtr(stelle) + ↙
          ↳ REAL(stelle,kind(prec))
!3D
      speicher3DPtr(ix+nx+1,iy+ny+1,iz+nz+1) = ↙
          ↳ speicher3DPtr(ix+nx+1,iy+ny+1,iz+nz+1) + ↙
          ↳ REAL(stelle,kind(prec))
!Maximum
      stelle_maxPtr = max(stelle_maxPtr, ↙
          ↳ REAL(stelle,kind(prec)))
    END DO
  END DO
END DO !omp
CALL MPI_WIN_UNLOCK(masterProc, s3Win1D, ierror)
```

```
20 CALL MPI_WIN_UNLOCK(masterProc, s3Win3D, ierror)
21 CALL MPI_WIN_UNLOCK(masterProc, s3WinMax, ierror)
```

Listing 5.13: TrackerSim with passive synchronisation on shared Memory with local load/store (Step 3)

5.1.6 Concluding discussion

A detailed analysis of every variation of `TrackerSim` is made in section **TrackerSim** (p.80). It is necessary to anticipate this section in its results since the following sections **Design approach** (p.74) and **Substitution of OpenMP** (p.77) already discuss a concrete approach to apply MPI RMA to `PHOENIX`.

The results hypothesise that an approach with passive target communication might provide good performance. Another aspect which encourages passive target communication is the fact that `TrackerSim` displayed the single steps in a worst case scenario: Not every operation which is protected through a special OpenMP directive is invoked in every iteration of the LC-tracker. The decision, if it is invoked, is made during runtime which is why fence synchronisation affects performance adversely as it requires

a) making additional arrangements between parent and child processes to avoid unnecessary collective RMA synchronisation or

b) follow a worst case pattern and perform the collective RMA synchronisation (needlessly) during every iteration

if the invocation pattern is irregular. The same applies to GATS synchronisation, because these operations require concerted actions, too. Passive target communication allows to react reasonably to the actual need during runtime.

Since MPI RMA shall be used to replace OpenMP it is certain that shared memory for shared windows is available, too. Section **TrackerSim** (p.80) exposed the advantage of a shared window over a normal window, therefore a combination of passive synchronisation and shared windows will be applied to `PHOENIX`.

In a final step one must decide between using RMA communication or local operations. The analysis of `TrackerSim` hypothesises that a combination of a shared memory with outer passive locks and `lock_type == MPI_LOCK_SHARED` and atomic RMA communication operations could provide the best performance for `PHOENIX`. Nevertheless local operations were used for `PHOENIX` instead of RMA communication operations.

On the one hand in each loop iteration of `TrackerSim` very few operations are executed which generates much overhead and less overlapping sections. `PHOENIX` in return has a much bigger amount of additional operations that are not part of critical or atomic sections and may be therefore executed concurrently - local operations encased by passive locks inside the loops profit by this.

On the other hand using RMA communication operations results in additional work to substitute every operation in a step, that accesses shared variables. Using local

operations instead of RMA communication operations allows to reuse more existing code, if the affected variables are substituted by RMA memory beforehand.

If the locks should be placed inside or outside the loops depends on the fact if update operations are used, which must be protected with an exclusive lock. If no exclusive lock is necessary the performance should benefit from locks, which are placed around the loop and have `lock_type == MPI_LOCK_SHARED`: the access epoch is initialised but the loop content may still be executed concurrently. The passive locks should be placed inside the loop as soon as the outer lock must use `lock_type == MPI_LOCK_EXCLUSIVE`, otherwise no parallelism is available. Depending on the actual code some of the inner locks might still be executed with `lock_type == MPI_LOCK_SHARED`.

5.2 Design approach

Since the substitution of the OpenMP section entails more additional work than apparent at first sight, this section describes necessary adaptions which do not deal with OpenMP for the most part. Still these adaptions are essential if RMA shall replace OpenMP in an optimal way - neglecting them would impair the performance most likely.

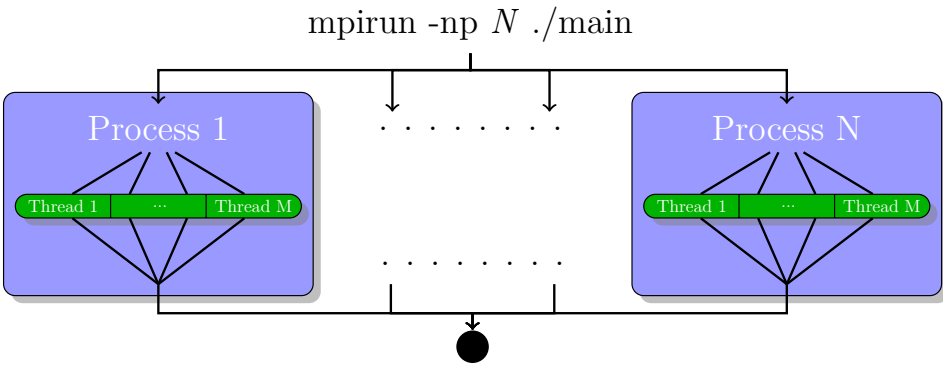


Figure 5.4: Current load distribution in `PHOENIX` with OpenMP

As seen in figure 3.4 `PHOENIX` subdivides the solid angles and assigns each part to the processes of a wavelength cluster. Each process then spawns up to M threads, which use one part of the grid respectively of the characteristics for their calculations. Figure 5.4 illustrates this situation in general. One problem, which arises from this situation, is that `TrackerSim` is executed with N processes concurrently through invoking `mpirun -np N`. But this procedure cannot be performed for `PHOENIX`, as it spawns already N processes for the wavelength clusters - every worker process must then be provided with M new tracker-processes (cf. **Applying MPI** (p.52)). This situation is displayed in figure 5.5.

A total of $(N \cdot M) + N$ processes are needed for executing `PHOENIX` with MPI-3 RMA operations. There are two different possible solutions for this problem:

- The additional processes could be spawned by invoking `PHOENIX` with `mpirun -np` $N + (N \cdot M)$ `./main`, but then adjustments must be performed on many

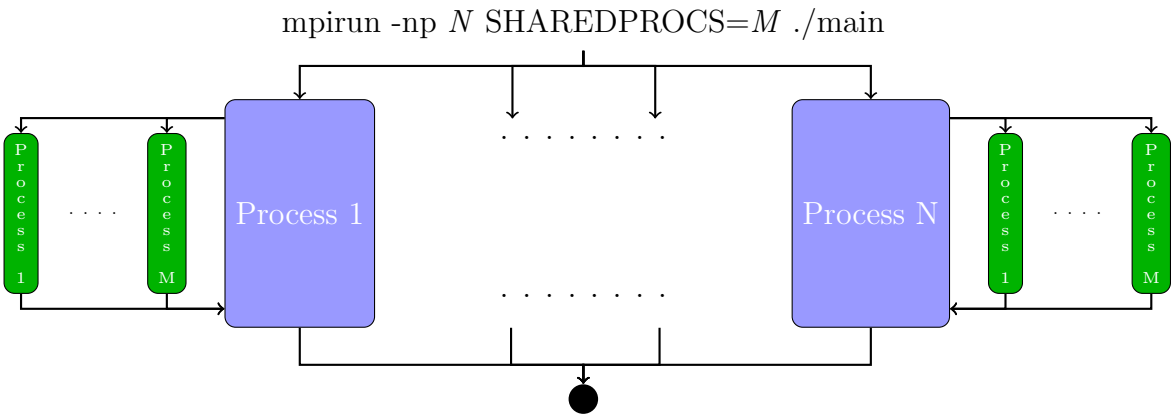


Figure 5.5: Adapted load distribution in PHOENIX without OpenMP

code sections which might spread all over PHOENIX. This is due to the fact, that without adjustments the additional processes would also be distributed to wavelength clusters, by what they would not be available as tracker-processes anymore. To solve this problem one could change the distribution section to exclude $N \cdot M$ processes. But because this is not the only situation where the processes are used explicitly in PHOENIX, so this approach entails a huge amount of necessary work, which has little to do with the real task.

- As showcased in **MPI-2.0** (p.23) processes can be spawned dynamically during runtime. Because these dynamically spawned processes can not be deployed in the application directly, everything, that shall be calculated with the new processes, needs to be outsourced into a new application, i.e. the new application mimics with MPI-3 RMA operations the OpenMP sections. Every primary process (hereafter: *parent process*) spawns new processes (hereafter: *child processes*) on its own by using MPI_COMM_SELF as communicator. Otherwise the new spawned child processes could interact with other parent processes. As the OpenMP threads are assigned to one specific process, this behaviour should imitated by the child processes, too.

Because the first solution makes major adjustments of PHOENIX code necessary[1], the second approach was chosen for implementation. The following steps show the sequence of actions in general, that are necessary to use RMA processes instead of OpenMP threads. The parent process undertakes the task of administrating the child processes, it does not participate in calculations, which were part of any of the three OpenMP sections (hereafter: Step 1, step 2 and step 3).

1. Spawn child processes:
   Each worker process spawns as many child processes for itself, as it would have made use of OpenMP threads. The child processes execute an application, which

[1] Actually necessary adaptions in the 3DRT test application could be kept within a limit. Since replacing OpenMP with RMA in an already existing (and possibly extensive) application could be a common use case, the second approach was still chosen.

contains everything that is needed to mimic every step with MPI-3 RMA. After spawning the child processes they form a group of their own without their parent process, the parent is connected with the child group through an intercommunicator only (cf. figure 2.9). To enable RMA usage between the parent and child processes they must build a mutual intracommunicator first (cf. **MPI-2.0** (p.23)). To avoid unnecessary overhead spawning and initialisation must not be done during the execution of the tracker, because the tracker is executed for all solid angles on every wavelength. Instead of that, the child-processes should be spawned before the first execution of the LC-tracker and killed after the last execution of the LC-tracker.

2. Window creation:
   After being spawned the child processes need at first to determine their parent process via `MPI_COMM_GET_PARENT`. The child processes share the same memory with their parent process, therefore they can create windows with shared memory. Like before this should be performed before the first invocation of the LC-tracker.

3. Data transfer:
   The extent of this step depends on the chosen window-strategy (cf. **Substitution of OpenMP** (p.77)). Generally speaking the parent process provides data through the window, which the child processes require for their calculations in step 1, 2 or 3 and collect the data from the window again, after the child processes finished the calculations for one specific spatial direction.

4. Control the child processes' progress:
   The child processes mainly shall perform, when the OpenMP sections in the LC-tracker would have been executed. To prevent that the child processes already compute the next step before the parent process finished its post-processing, the single steps on each solid angle need to be synchronised. To guarantee a correct sequence of interactions between the parent process and its child processes, MPI barriers are used at crucial checkpoints: Listing 5.14 displays the changes that are necessary in general for the 3DRT main file. The actual required synchronisation pattern is shown in listing 5.15 for the execution of the LC-tracker by the parent and in listing 5.16 for execution of every tracker step by the child processes.

```
Parent - main

Spawn processes
Create RMA windows
Reset pve_grid pointer
[ ... ]
Loop over solid angles
  [ ... ]
  Invoke LC-tracker
  [ ... ]

[ ... ]
```

Listing 5.14: main

```
Parent - LC-tracker
[ ... ]
MPI_BARRIER

MPI_BARRIER
Use step 1 results
MPI_BARRIER

MPI_BARRIER
Use step 2 results
MPI_BARRIER

MPI_BARRIER
Use step 3 results
[ ... ]
```

Listing 5.15: LC-tracker

```
Child
Create RMA windows
While(not condition)

MPI_BARRIER
Compute step 1
MPI_BARRIER

MPI_BARRIER
Compute step 2
MPI_BARRIER

MPI_BARRIER
Compute step 3
MPI_BARRIER
update condition
```

Listing 5.16: Child

5.3 Substitution of OpenMP

Section **Initialisation** (p.29) presented four different possibilities to create a window so that every process could fetch required data from a central memory location. As discussed in section **Concluding discussion** (p.73) passive target communication shall be used in combination with shared RMA windows.

Because the necessary code adjustments are numerous it is refrained from printing the new code in this section. Generally speaking the adjustments in `TrackerSim` were applied to the LC-tracker and other involved program files of `PHOENIX`: After every worker process has spawned its child respectively tracker processes the overall communication pattern is built through `MPI_BARRIER`s. The major task is to create every shared window that is needed to replace memory, which is used in a shared manner during an OpenMP section. `PHOENIX` uses amongst others one large derived datatype `pve_data` to gather every variable, which is somehow associated with the LC-tracker. Most entries of a variable of type `pve_data` (hereafter: `pve_grid`) are pointers. This makes it impossible to create a single shared window, which represents `pve_grid`,

because this would require one contiguous memory stack. Through the use of pointers, the memory, which belongs to `pve_grid`, is distributed over the local memory. Creating a shared window for `pve_grid` only makes the pointer addresses available to every associated process - since the memory, to which the pointers refer to, is not part of a shared memory it is not accessible via RMA.

To solve this problem, every variable's content, which is on the one hand part of `pve_grid` or is referred to by a pointer of `pve_grid` and on the other hand is necessary for the LC-tracker, needs to be copied to a shared window. After every tracker process fetched a pointer to these windows through `MPI_WIN_SHARED_QUERY` these variable may then be accessed and updated through RMA. If every child process creates a `pve_grid` for its own (additional memory requirements are insignificant, because `pve_data` mainly consists of pointers) and redirects the pointers of their `pve_grid` to the newly fetched pointers (referring to the RMA windows), no further adjustments to the original code of every step are necessary.

To guarantee the correct functionality of the LC-tracker, 3DRT compares the results for one specific voxel and its 26 neighbouring voxels with the correct result. As long as the absolute error deviation undermatches the critical value of $1 \cdot 10^{-10}$ the calculations are considered correct. Figure 5.6 shows an example of the execution of 3DRT with `LC_Lstar_tracker_PBC_zmap` on Linux-PC with four processes and four threads each on a $(32 \times 32 \times 128)$ grid with (16×16) solid angles. Because the error deviation is this case is very small it is barely visible in the figure.

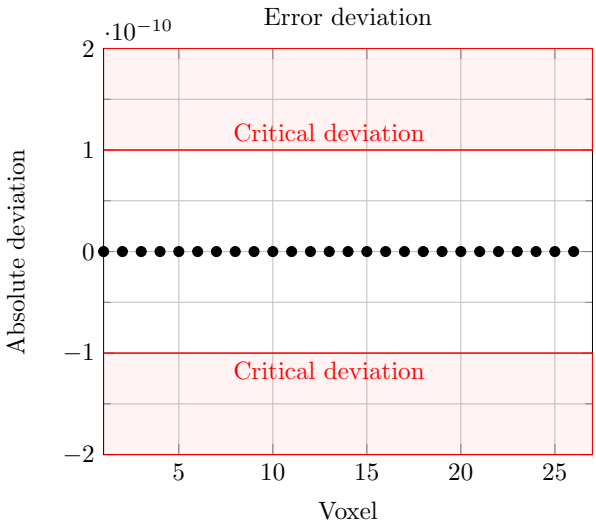


Figure 5.6: Example for a very low error deviation

6 Evaluation

This sections provides benchmarks for `TrackerSim` and `PHOENIX`. Based on the results of benchmarking `TrackerSim` the most promising approach was chosen to implement the necessary adjustments in `PHOENIX` to substitute OpenMP for MPI RMA. The following hardware systems were used to perform these benchmarks and are accordingly marked in the plots' description. The cross compiling for Knights Corner is performed on Linux-PC.

Linux-PC :

- Intel Xeon CPU E5-1620 v2 @ 3.70GHz (4 cores, hyperthreading)
- 64 GB RAM
- SUSE Linux Enterprise Server 11.2
- Intel® Parallel Studio XE 2017

Knights Corner :

- Intel Xeon Phi Coprocessor 3120A @ 1.1GHz (57 cores)
- 6GB RAM
- MPSS 3.4.2

Knights Landing :

- Intel Xeon Phi Processor 7210 @ 1.3GHz (64 cores)
- 80GB RAM + 16GB MCDRAM (cache+quadrant)
- CentOS Linux release 7.2.1511
- Intel® Parallel Studio XE 2017

6.1 Tools

During the development and benchmarking of `TrackerSim` and `PHOENIX` some tools were used to support the whole process and are therefore introduced briefly:

- GNU gprof[1]:
  This tool can be used to create call graphs and was mainly used in the beginning to receive a first impression of the workflow of `PHOENIX`.

- Intel VTune Amplifier[2]:

[1] `https://sourceware.org/binutils/docs/gprof/`
[2] `https://software.intel.com/en-us/intel-vtune-amplifier-xe`

A tool for application performance analysis, which offers a large variety of features (e.g. hotspot detection, concurrency analysis, memory access, etc.). A GUI for a visualisation of the results is available, too.

- Vampir[1]:
  Tool to visualise collected applications' traces. Because VampirTrace does apparently not support MPI-3 operations, the traces were created with SCORE-P[2] instead.

- micsmc: A simple tool, which allows an overview of the capacity utilisation of the Intel Xeon Phi, figure 6.1 shows an example.

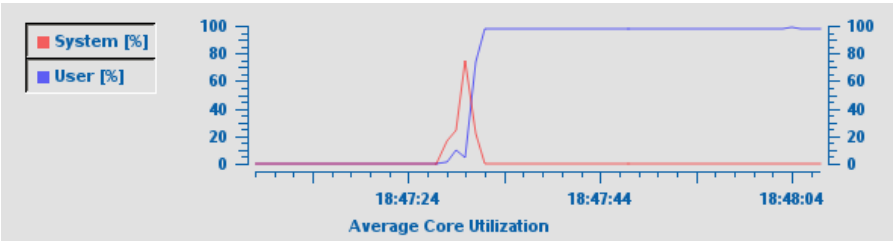


Figure 6.1: Average core utilisation displayed by micsmc, process initialisation through `TrackerSim` becomes apparent

6.2 TrackerSim

In the prior chapter **Using MPI-3 RMA** (p.60) four different variations for `TrackerSim` have been presented. To choose an approach, which shall be used to replace `PHOENIX`'s OpenMP with MPI RMA, each variation has been benchmarked. The results are discussed in the following sections. Step 1 contains three nested loops, which are collapsed in case of OpenMP; in case of MPI RMA the range of indices of the outermost loop are distributed evenly amongst the available processes. If the range of indices can not be distributed evenly, the process with highest rank takes them over. No additional actions for load distribution were taken.

The content of each (nested) loop is executed $32 \cdot 32 \cdot 32 = 32768$ times, which meets a typical grid size of `PHOENIX`. Up to six different versions of `TrackerSim` are used for the following benchmarks (see section **TrackerSim** (p.61) for a describtion of each version in detail):

SMP: Variation 0, which makes use of OpenMP, the processing elements (PE) for this version are therefore threads. For every other version a processing element describes a MPI process.

Fence: `TrackerSim`-variation 1, hereafter called *`TrackerSim`-Fence*

[1] `https://www.vampir.eu/`
[2] `http://www.vi-hps.org/projects/score-p/`

Lock: `TrackerSim`-variation 2, hereafter called *`TrackerSim`-Lock*

Shared: `TrackerSim`-variation 3, hereafter called *`TrackerSim`-Shared*

SharedNoRMA-A: This version accords to the variation 4, which has been presented in **Variation 4: shared memory with local communication** (p.71); the locks are placed outside the loops, hereafter called *`TrackerSim`-SharedNoRMAOuterLocks*.

SharedNoRMA-B: To examine the effect of passive locks, which are executed more frequently, they were placed inside the nested loops - hereafter called *`TrackerSim`-SharedNoRMAInnerLocks*.

6.2.1 Comparison of OpenMP and MPI-3 RMA

For the first benchmark the parallelisation of loops with OpenMP was compared to the parallelisation with MPI-3 RMA. To reduce side effects this comparison is executed with no additional load. The results on the Linux-PC are displayed in figure 6.2. Attention should be paid, that the visualisation of step 3 uses a logarithmic scale because the difference between OpenMP and MPI RMA is very extensive.

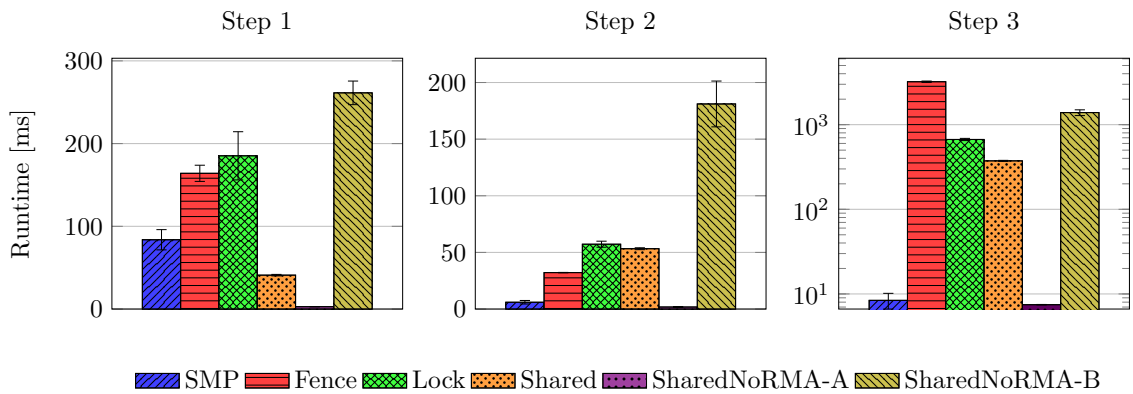


Figure 6.2: Benchmarks of every variation of `TrackerSim` with no load and four processing elements on Linux-PC

In every step the version with active target communication is outperformed by OpenMP: Every process needs to participate in performing fence synchronisation (active, collective operation) which induces too much overhead. The same goes for the variation with shared memory and inner passive locks (*`TrackerSim`-SharedNoRMAInnerLocks*): The locks are set and withdrawn in every loop iteration, the amount of lock management dominates.

Step 1 and step 2 reveal a similar picture: MPI RMA with a normal window (*`TrackerSim`-Fence* and *`TrackerSim`-Lock*) are not able to compete against OpenMP. The direct comparison between *`TrackerSim`-Lock* and *`TrackerSim`-Shared*, which only differ regarding the RMA window type, shows a supposable reason: the benefit of

using shared memory. *TrackerSim-Shared* performs in step 1 even better than TrackerSim$_{SMP}$. Using a shared memory to replace OpenMP in PHOENIX has therefore a lot to commend it.

Figure 6.3 and figure 6.4 show the results of the same run on both generations of Intel Xeon Phi. As expected the runtime on the Knights Landing is faster than on Knights Corner, most variations gain a speedup of about factor 3, which correlates with about the proportion of the FLOPS values of KNC and KNL. However some single measurements deviate slightly from this value, it is for example very noticeable that *TrackerSim-Lock* sticks out from *TrackerSim-Fence* in step 1 on KNL. Because only four processing units were used on the Xeon Phis they could not deliver their maximum performance and the Linux-PC, which has the highest clock rate per core, outperforms them.

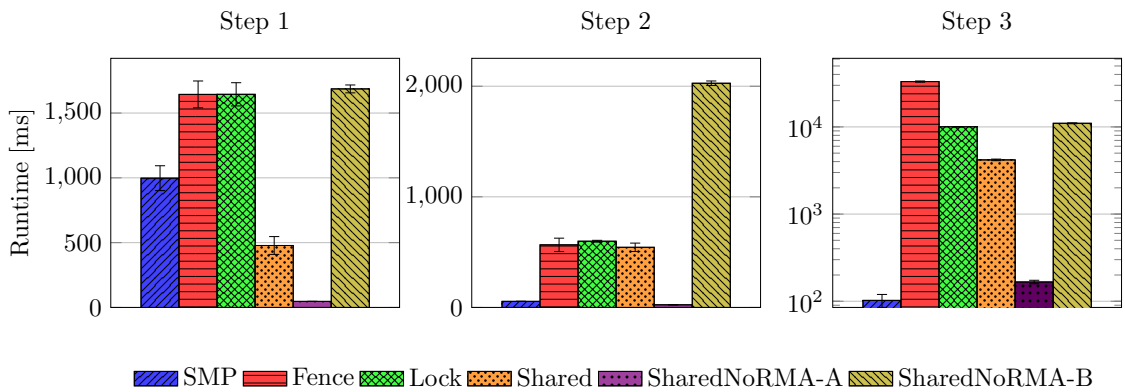


Figure 6.3: Benchmarks of every variation of TrackerSim with no load and four processing elements on Knights Corner

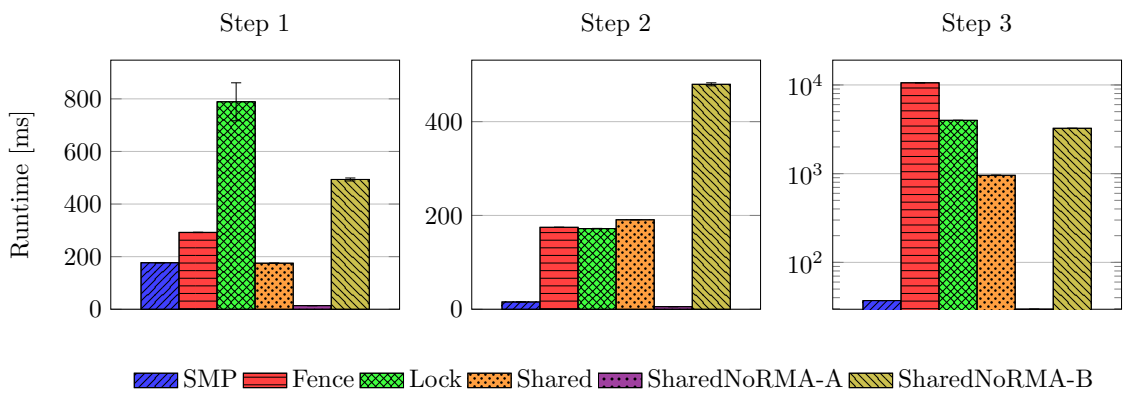


Figure 6.4: Benchmarks of every variation of TrackerSim with no load and four processing elements on Knights Landing

The question remains if a substitution of OpenMP should still make use of RMA communication operations even though using a shared memory allows local load and store operations. Comparing *TrackerSim-SharedNoRMAOuterLocks* with *TrackerSim-Shared*, which uses RMA communication operations, shows clearly in every step, that local

operations on a shared memory are executed much faster than the RMA communication operations. If it is in addition necessary to guarantee the reusability of a buffer, which is not part of a window, the negative impact of RMA communication on the performance emerges from step 3: Every RMA operation, which is executed in step 1 to 3 is atomic and therefore safe regarding concurrently updates and accesses, but every variation with RMA communication operations needs to call additional RMA synchronisation operations because of a used buffer, before the next iteration can be performed (cf. an example with explanation in section **Variation 1: fence-synchronisation** (p.65)). If only local operations on a shared memory are used (like in *`TrackerSim`-SharedNoRMA*) this additional synchronisation calls are unnecessary. Of course this advantage is of no relevance if the locks are placed inside the nested loops: the performance results of *`TrackerSim`-SharedNoRMAInnerLocks* show, that in this case local operations lose their advantage over RMA communication operations

These findings lead to the decision that, as long as no internode communication is needed, which would render RMA communication operations essential, the most promising approach to substitute OpenMP is a combination of shared windows with local operations.

6.2.2 Scaling

After the direct comparison of every variation without additional load, another interesting point is, how well which variation scales with an increase in the number of processing elements and loop size. For weak scaling the initial load is set to 50 on the Linux-PC and to 100 on a Xeon Phi for each step, which increases the total runtime of step 1 by about two up to eight times - the actual factor depends on the chosen variation and arguments. Without the initial load nearly no parallelism would be available which would not mimic `PHOENIX` in a realistic way. The number of processing elements and the (outer) loop size is then increased proportionally to examine the weak scaling. In the best case the runtime remains constant.

For an improved differentiation a logarithmic scale is used in figure 6.5, which illustrates various aspects:

1. The Linux-PC has four cores with hyper-threading enabled. As soon as the amount of threads rises above the amount of physical cores, the standard deviation of $\texttt{TrackerSim}_{\text{SMP}}$ increases significantly[1] and makes it very difficult to make a binding statement. At least for a thread count lower or equal four it still has a good and stable performance.

2. *`TrackerSim`-Fence* and *`TrackerSim`-Lock* scale very similar. *`TrackerSim`-Lock* cannot benefit from its passive locking against the collective, active fence synchronisation, because `TrackerSim` does not provide an opportunity and invokes both synchronisation operations exactly once per iteration.

[1]Because of the logarithmic scale one could by mistake underrate the error deviation

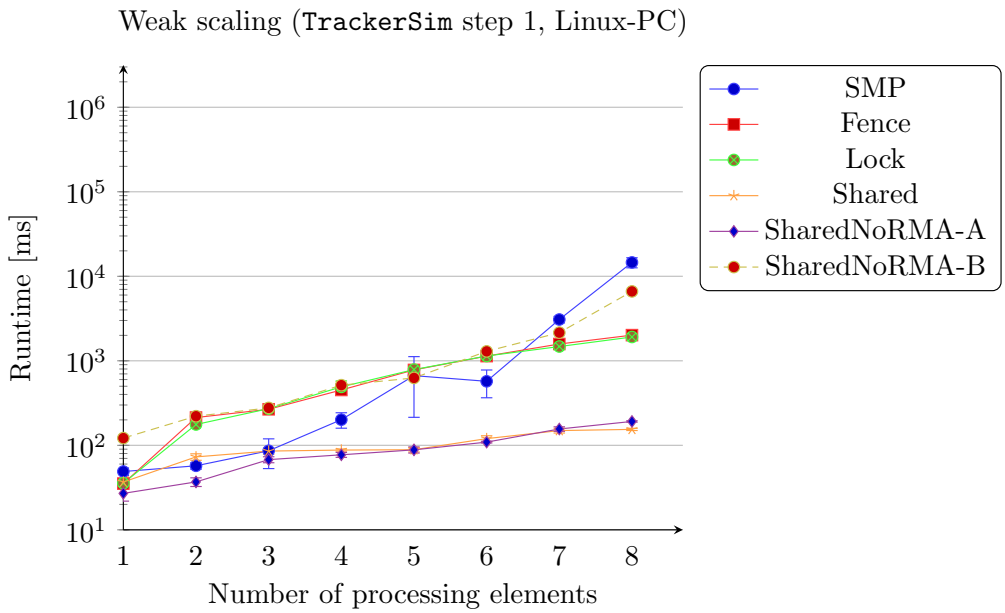


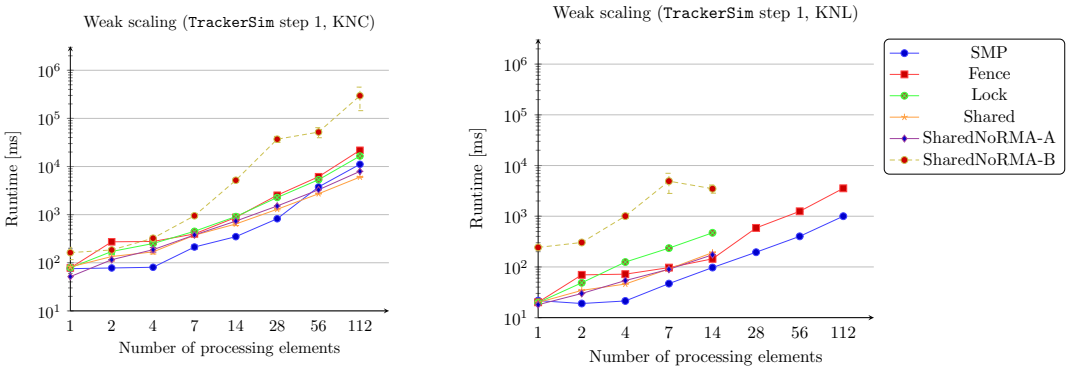
Figure 6.5: Weak scaling of `TrackerSim` on Linux-PC

3. *`TrackerSim`-Shared* uses the same RMA communication operations like *`TrackerSim`-Lock* but still shows a significant better scaling. It is definitely worth it to use shared memory if it is available. Both variations have an extraordinarily increased runtime during the transition of one process to two processes because like mentioned in **Variation 2: passive synchronisation** (p.67) and shown in figure 5.3 a shared lock executed by a process on its own local window becomes an exclusive lock. Therefore the process with rank 0 cannot be executed concurrently to the other processes.

4. *`TrackerSim`-SharedNoRMA* with outer locks has the best scaling behaviour and is only for eight processes marginally outperformed by *`TrackerSim`-Shared*. Since MPI communication should entail more overhead than local operations this indicates that the used MPI implementation *Intel MPI* uses optimisations in the background if RMA operations are used for intranode communication.

5. Even though the additional load should privilege *`TrackerSim`-SharedNoRMA* with inner locks against the variation with outer locks quite the contrary happens. The outer locks with `lock_type == MPI_LOCK_EXCLUSIVE` allow only a sequential execution of the processes whereas the variation with inner locks affords an opportunity to execute the processes concurrently. Apparently the overhead of the many called inner locks makes it impossible to use this advantage.

For comparison figure 6.6 shows the results of the same run executed on each Xeon Phi. In contrast to the benchmark on the Linuc-PC the run of $\texttt{TrackerSim}_{\text{SMP}}$ is more stable and keeps as far as 4 threads the optimal performance. The reason for this might be that the threads are distributed on the same tile: Each core can in a best-case



(a) Knights Corner

(b) Knights Landing

Figure 6.6: Weak sclaing of `TrackerSim` on Intel Xeon Phi

scenario execute up to four threads without loss in performance but only four vector processing units are available. More threads are then distributed among more tiles to guarantee that each process has access to its own VPU. But again the trend of each variation's performance is quite similar to the run on the Linux-PC.

In figure 6.6b the time for *`TrackerSim`-Lock*, *`TrackerSim`-Shared* and *`TrackerSim`-SharedNoRMA* is only measured for executions up to 14 processes. Any execution of these variations with more than 15 processes freezes, therefore no runtime values are available. Since *`TrackerSim`-Fence* is unimpaired by this phenomenon its origin is not MPI-3 RMA in general. In fact only variations are affected, which make use of passive target communication. One possible explanation for this problem might be, that the KNL is used in *quadrant* mode, which logically divides the cores into quarters. Invoking passive locks on a process, which is not executed on a tile in the same quarter, seems to cause this problem. Further investigations would be required to determine the definite cause and a possible solution.

After having a look on weak scaling the performance of each variation is displayed in figure 6.7 with regard to strong scaling: The load is set to 50 for the execution on Linux-PC and to 100 on Intel Xeon Phi, the outer loop is on both platforms set to 80[1] while the amount of executing units is increasing. In the best case the runtime decreases proportional with the increase of processing elements. For a more clear elaboration of the differences again a logarithmic scale is used.

Again $\texttt{TrackerSim}_{\text{SMP}}$ has a high standard deviation for a thread count above four and scales pretty bad. The combination of hyperthreading and thread organisation seems to have a fatal effect on the runtime. *`TrackerSim`-Fence*, *`TrackerSim`-Lock* and *`TrackerSim`-SharedNoRMA* with inner locks scale like before: The amount of overlapping computation phases is not big enough to compensate the sequentially executed code sections, the runtime increases proportionally to the number of processing elements.

`TrackerSim`-Shared performs better than the other variations again: the actual runtime gains a speedup of about $1,5$. Still this is far below the possible maximum

[1]For comparison: The number of iterations of both inner loops are set to 32

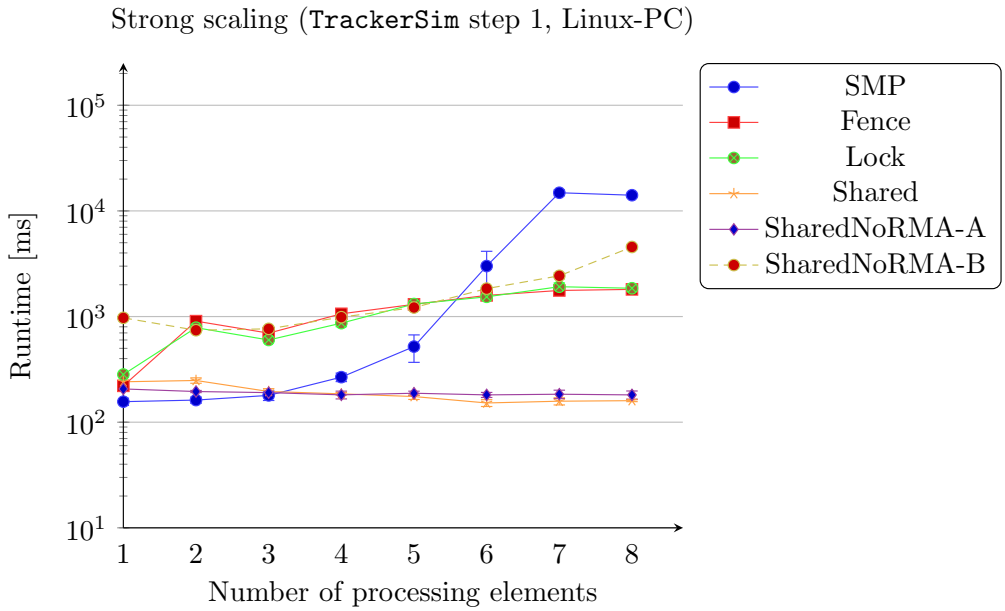
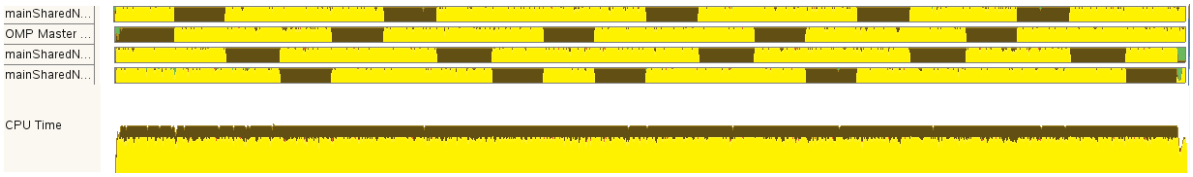
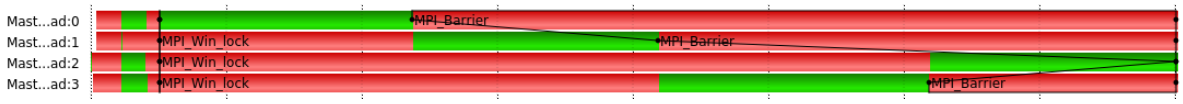


Figure 6.7: Strong scaling of `TrackerSim` on Linux-PC

speedup of 8. Obviously the MPI implementation can optimise the synchronisation of atomic RMA communication operations by what *`TrackerSim`-Shared* outperforms the variations with local operations and explicit synchronisation.



(a) VTune: outer locks, five iterations



(b) Vampir: outer locks, one iteration

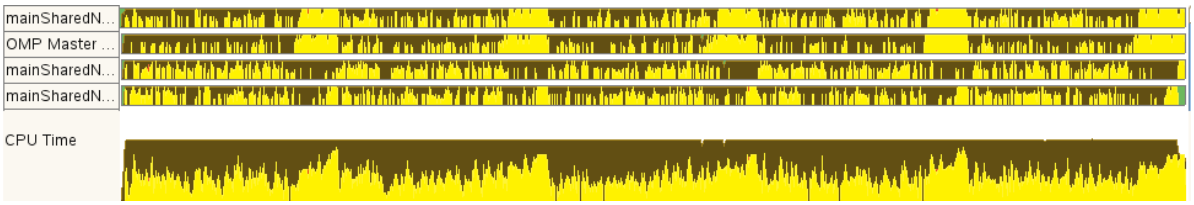
Figure 6.8: *`TrackerSim`-SharedNoRMA* (Step 1) with additional load

The runtime of *`TrackerSim`-SharedNoRMA* with outer locks remains relative stable even though the processes can execute the loops only sequentially, which is evident in the output of VTune[1] in figure 6.8a: the yellow sections, which stand for MPI communication, predominate brown computation phases, which do not appear concurrently. This is the reason that the overall CPU time consists mainly of MPI communication. An analysis with Vampir[2] of one iteration (cf. figure 6.8b) reveals that the MPI communication

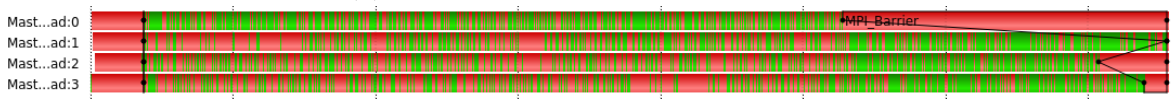
[1]Result of `mpirun -np 4 mainSharedNoRMAInnerLocks -s1 -l1 1000` with five iterations

[2]Result of `mpirun -np 4 mainSharedNoRMAInnerLocks -s1 -l1 1000` with one iteration

consists mainly of the exclusive outer locks, which prevent a concurrent execution of the loop, and barriers to guarantee that the processes finish simultaneously and the time measurement is performed right.



(a) VTune: inner locks, five iterations



(b) Vampir: inner locks, one iteration

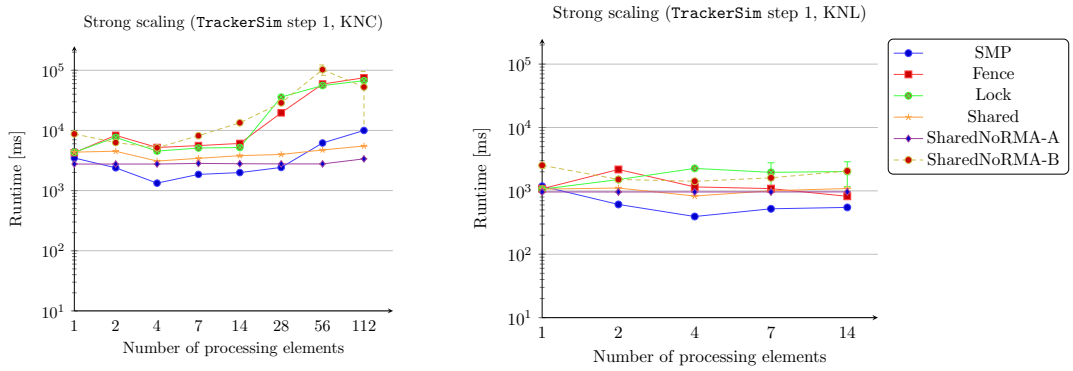
Figure 6.9: `TrackerSim`-*SharedNoRMA* (Step 1) with additional load

Setting the exclusive locks inside the loop enables a concurrent execution again, figure 6.9 shows the the same run like before but now with inner locks. To perform better than the variation with outer locks the additional overhead needs to be compensated; therefore the application must offer overlapping computation sections during one iteration, which are big enough to make this variation cost-efficient.

Despite this disadvantage the variation `TrackerSim`-*SharedNoRMA* with outer locks only has very few additional overhead in comparison to the other variations, because it uses a shared memory, places the locks outside of the loops and makes use of local operations. Therefore the runtime of each process regarding the loops is reduced to the same degree as more processes are used. The processes are indeed executed sequentially but need less time to execute their portion of the loop in return, so the bottom line is that the total runtime of step 1 is roughly constant. With this approach no application can benefit of additional processes and is therefore no wise alternative to `TrackerSim`-*Shared*, even though the runtime looks promising.

Again for comparison figure 6.10 shows the results of the strong scaling benchmark executed on both Xeon Phis. To compensate the fact, that the execution on the Xeon Phi is run with more processing elements in total but the same outer loop size, the amount of additional load was increased to 100. Figure 6.10b displays only results up to 14 processing elements for the same reason like before: executions with passive target communication did not finish with more than 15 processes.

Since OpenMP is not impaired through hyper-threading anymore it outperforms most of the time every MPI-3 RMA alternative. Every alternative begins to lose performance beginning with a number of processing elements between 14 and 28, there might be a sweet spot where the workload per process becomes too little and by what the overhead becomes too important. Apart from that every variation performs in the same way on the Xeon Phi like on the Linux-PC.



(a) Knights Corner

(b) Knights Landing

Figure 6.10: Strong scaling of `TrackerSim` on Intel Xeon Phi

6.3 Phoenix

Following the results from the prior section a combination of a shared window with local operations and passive locks is used to replace OpenMP in `PHOENIX`. As discussed in section **Substitution of OpenMP** (p.77) `PHOENIX` has one major derived datatype, which holds many pointers. For each pointer a new window must be created so that the data can be accessed and updated remotely. Therefore, only step 1 of the LC-tracker has been converted into a MPI-only version.

6.3.1 Comparison of OpenMP and MPI-3 RMA

The converted version of the LC-tracker, where the first OpenMP section has been replaced with MPI-3 RMA operations, still has at this particular time some bugs. Depending on the number of worker processes, tracker processes and grid size the application halts with a segmentation fault. A comparison between the results of both versions of step 1 (MPI-3 RMA and OpenMP) shows that the RMA version does not compute correct results, so additional work is needed to rectify the RMA version. But for the most part every operation is already used, which is theoretically needed to replace the OpenMP section. In case that the error does not have a major effect on the runtime (if for example only a pointer is wrongly used instead of a missing lock), this allows a wary estimation of the suitability of MPI-3 RMA for the LC-tracker.

The benchmark has only be performed for the first solid angle because the erroneous RMA version creates some zero results, from which the subsequent steps would use the reciprocal values. Since it was difficult to determine an executable combination of worker and tracker processes and grid size, the benchmark begins already with four tracker processes. Figure 6.11 displays the results for the runtime of both LC-tracker versions[1]. Because of present errors the expressiveness of this benchmark must be handled with care. Both versions seem to have a problem with six processing elements, which can

[1]This time without calculating the mean value and standard deviation, since this benchmark only provides a first impression.

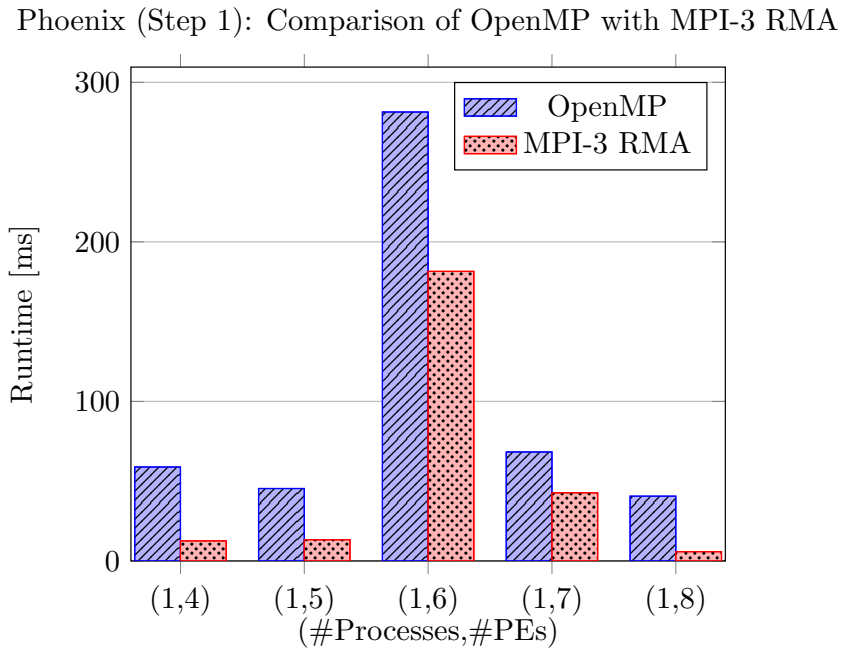


Figure 6.11: Benchmark of PHOENIX on Linux-PC: small grid $(8 \times 8 \times 8)$ and few solid angles (2×2)

probably traced back to how the process affinity distributes the processing elements among the four cores. Apart from that one may suppose that MPI-3 RMA outperforms OpenMP in step 1.

One possible reason might be the fact, that after the critical OpenMP section (cf. listing 5.1) an array is written at position `idx`. Since `idx` is increased every time the critical section is executed, each processing element has a unique value for `idx`. Therefore every array element is written by at most one processing element. Possibly this is knowledge, which only the user but not OpenMP can make use of. Because no element of the array needs to be protected against concurrent accesses, a shared lock, which does not prevent concurrent accesses on the same window, is sufficient.

6.4 Evaluation

`TrackerSim` is a very artificial application, which gives a first impression about how well various combinations of RMA initialisation, communication and synchronisation operations perform. Which combination should be implemented for a concrete application depends mainly on the application itself, section **Concluding discussion** (p.73) gives an example of such a consideration.

6.4.1 Runtime

In direct comparison OpenMP seemed to perform much better than most of the other MPI RMA variations, which have been presented in this thesis. But as soon as the comparison is repeated in a more realistic environment with additional load, which enlarges the amount of possibly overlapping computations, this impression changes and in many cases now MPI-3 RMA performed much better than its OpenMP counterpart even though there is still additional potential for optimisations: A MPI implementation might speed RMA even more up if `hint` or `assert` is used for RMA window creation or RMA synchronisation operations. [GT07] showed the possible impact of the `assert`-parameter on different architectures: In some cases no difference was measurable but in other cases it was noticeable.

In case of too much overhead OpenMP seems to perform better than MPI RMA, if OpenMP is not impaired by the hardware through hyper-threading or the like. This became apparent through the scaling benchmarks on the Xeon Phis, however this situation should anyway be prevented at all necessary cost because in this case the problem size is not suitable for the amount of cores.

[BSCL14] examined the effect of the message size on the performance of MPI-3 RMA and found out that MPI RMA performs better with a bigger message size. Since variable message sizes are not implemented in `TrackerSim`, which only accesses and updates single data with only a few operations per iteration, the potential of MPI-3 RMA regarding a real application is underestimated by trend in this chapter. Probably *`TrackerSim`-Shared* performed fairly well because every RMA communication operation was invoked during the same epoch and decreased therefore the percentage of RMA synchronisation.

Even though shared memory with local operations showed a great performance this result should be handled with care as placing exclusive locks too generously results in an impaired parallelism. However as soon as shared memory is available one should definitely make use of it. Which type of communication and synchronisation is then used, depends on the application: Using atomic RMA communication operations results in less explicit synchronisation calls by the user, using more explicit synchronisation then again allows to omit unnecessary synchronisation, which would be automatically invoked by the system in case of doubt - to guarantee correctness an MPI implementation as well as OpenMP need to lock conservatively even though the actual use case might allow to omit it. The prior section **Comparison of OpenMP and MPI-3 RMA** (p.88) probably (if the better performance is not due to the error) gives an example that using explicit locks might be advantageous: Perhaps OpenMP does not recognize that every thread only accesses unique elements and locks therefore the whole array against concurrent access operations; in contrast using MPI-3 RMA allows to use a shared lock explicitly.

6.4.2 Memory usage

During the replacement of the OpenMP section through MPI RMA operations the focus did not lie on saving memory. Therefore there might be still potential to optimise

PHOENIX in this respect.

It is not possible to assign already allocated memory to a newly created shared window, therefore the only possibility to save memory is to replace every allocation operations through window creation operations. Otherwise one has to copy all data from allocated memory to the windows' memory and deallocate redundant memory afterwards. In the worst case the memory demand increases by the size of the replaced variables between the window creation and the memory deallocation. Even though this is is only a peak, the capacity needs to be available. The same goes for static variables, which shall be shared through a shared window.

As discussed in section **Substitution of OpenMP** (p.77) derived datatypes with pointers makes additional effort necessary as for every memory, which is referenced by a pointer, a new window must be created. In return it is easier to distribute memory segments among nodes. Without RMA each node would create a single segment as a new independent variable, which are only implicitly connected with each other, and requires that every node handles the logical connection itself. Using MPI RMA allows to create one single window object to which each process contributes its memory through its local window. This results in one single window object which covers each memory segment explicitly. To access another segment a process only requires the rank of the segment's owner.

However the user needs to take care to avoid redundancy because spawning a new process implies duplicating every variable, too, if its existence is not associated to a specific rank.

7 Conclusion

7.1 Discussion

In comparison with OpenMP it is more extensive to write equivalent code with MPI RMA, because OpenMP undertakes many tasks of administering necessary steps for using shared variables, which an user of MPI RMA must perform himself. This disadvantage is at the same time the greatest advantage of MPI RMA: the user is for example able to determine whether and how synchronisation is performed. This might increase the performance if the user can utilise optimisations like leaving unnecessary synchronisations out or collecting multiple operations in one synchronisation epoch, which are available for the actual application and are therefore not performed by OpenMP's conservative practice.

The benchmarks showed that in certain circumstances MPI RMA can outperform OpenMP but it is important to chose the right approach. Which kind of window creation and RMA communication should be used can quite easy be determined by looking at the code and choosing an RMA approach which mimics it: If a variable shall be shared one uses a shared window; if memory needs to be allocated dynamically one uses a dynamic window; if an operation should be protected against data races one uses atomic RMA communication operations and so on.

Which RMA synchronisation operations should be used is not as easy to determine and makes more in-depth knowledge about the application necessary. Besides trial and error the following procedure has proven itself:

- Active target synchronisation fits best for applications which have bulk computation and synchronisation stages and static communication patterns. GATS synchronisation has possibly less overhead than collective fence synchronisation and should therefore be preferred, if not every process needs to participate in synchronisation [HDT+13, 8].

- Passive target synchronisation should generally perform better than active target synchronisation because the target is not actively involved in the synchronisation; therefore it is most suitable for irregular communication patterns. Additional attention must be paid to ensure that the outcome is correct.

- To minimize the amount of RMA synchronisation operations one should perform as many RMA communication operations during the same epoch as possible without impairing the concurrent execution [MPI12, 437].

If possible an application should be written with MPI RMA in mind or already using it instead of inserting it subsequently. Section **Design approach** (p.74) and section **Substitution of OpenMP** (p.77) showed a selection of drawbacks which could have been avoided if PHOENIX would have been built with MPI RMA in mind. One major disadvantage of the LC-tracker of PHOENIX is the fact, that most updates on shared variables inside the OpenMP sections are only executed on single data; [BSCL14] showed that MPI RMA performs better with bigger message sizes.

Even though using MPI RMA entails additional effort a suitable application definitely will benefit from it. The results from benchmarking step 1 of PHOENIX with OpenMP respectively MPI-3 RMA operations might be erroneous but still look very promising.

7.2 Related work

[aMT06] examined the potential use of MPI-2 RMA to replace OpenMP in a simple algorithm. The MPI-2 RMA replacement performed really bad in comparison to OpenMP and was tedious to implement. This showed that it is possible to replace OpenMP with MPI RMA in general but additional effort is necessary to lessen the overhead as low as possible. However the expansion of MPI RMA with the third MPI standard provides the user with additional approaches to fulfil the objective, which is why this thesis makes use of MPI-3 RMA.

To determine which approach regarding the RMA synchronisation operations might perform best, TrackerSim was built to mimic an application, where OpenMP should be replaced with MPI-3 RMA. This explorative approach makes additional effort necessary to build TrackerSim, which is especially adapted to PHOENIX; its results cannot just like that be applied to any other application - a large reconstruction must be done first to adapt TrackerSim to the new application. Another possibility is to measure the performance of each MPI RMA operation first and then calculate which combination of MPI RMA operations would take how much time to perform. [GBH13] developed an abstract cost function for each MPI RMA operation, which can be determined for a specific combination of hardware, process number, data size and number of communication targets by simple performance tests. This allows an early estimation which approach is most suitable for a specific situation (e.g. an concrete OpenMP section).

7.3 Future work

Obviously a possible future work is to correct the MPI-3 RMA approach for step 1 of PHOENIX and then implement the remaining two steps also with MPI-3 RMA. But since replacing OpenMP with MPI-3 RMA is a wide topic, there are some open issues in addition whose considerations might lead to an improvement of performance.

In case of PHOENIX MPI-3 RMA has only been used to improve the performance but there are more use cases, from which PHOENIX might benefit. [HB99, Ch. 3.1] states that PHOENIX would benefit from a dynamic load balance, which distributes

wavelength points to wavelength clusters dynamically. To avoid deadlocks additional communication and synchronisation would be necessary, therefore it was refrained from implementing this approach. Using MPI-3 RMA would in contrast to common two-sided communication be able to lessen the overhead of communication and synchronisation if passive target communication is applied. Currently a simple round robin method is used for a static distribution which is quite adequate for a homogeneous cluster. A dynamic method would allow to execute `PHOENIX` satisfactorily on a heterogeneous cluster, too.

In addition using MPI-3 RMA instead of OpenMP could enable a new approach to distribute the memory, which belongs to a task of a worker process, among several nodes: On the one hand this allows to decrease redundancy and increase the possible problem size as a result; on the other hand this would make it possible to work on the task of a worker process with not only a single process but with several processes with distributed memory. [BL15] gives an example to enable a smooth transition of the communication between two processes, which share the same memory or lie on different nodes.

Even though section **Concluding discussion** (p.73) discussed which combination of MPI-3 RMA operations might deliver the best performance for `PHOENIX`, some theoretical considerations were part of this discussion. Therefore it would not be digressive to try out other possible combinations. Section **TrackerSim** (p.80) showed that a shared window in combination with atomic MPI-3 RMA communication operations performed several times very well, too. Since some atomic MPI-3 RMA communication operations are optimised to be applied to a single datum, which is a common use case in `PHOENIX` (e.g. counter variables), they could compete with the actual approach of using local operations with exclusive locks.

Bibliography

[aMT06] Dieter an Mey and Irene Tedjo. Adaptive Integration-Form OpenMP to MPI. 2006. Accessed: 16.10.2016.

[Aqu] Ivan De Gennaro Aquino. 3D NLTE Radiative Transfer Modelling of M-dwarf Atmospheres. Unpublished PHD-thesis.

[AWH16] Mario Arkenberg, Viktoria Wichert, and Peter H. Hauschildt. Parallelisation of Critical Code Passages in PHOENIX/3D. Zenodo, October 2016. Poster No. 185.

[Bal04] Pavan Balaji. Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. In *In RAIT workshop '04*, page 2004, 2004.

[BD04] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004.

[BH14] Maciej Besta and Torsten Hoefler. Fault Tolerance for Remote Memory Access Programming Models. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 37–48, New York, NY, USA, 2014. ACM.

[BL15] Mikhail Brinskiy and Mark Lubin. An Introduction to MPI-3 Shared Memory Programming. *The Parallel Universe Magazine*, (21):36, 2015. Accessed: 14.10.2016.

[BSCL14] Mikhail Brinskiy, Alexander Supalov, Michael Chuvelev, and Evgeny Leksikov. Mastering performance challenges with the new mpi-3 standard. *The Parallel Universe Magazine*, (18):47, 2014. Accessed: 28.09.2016.

[CPU] CPU-Z OC World Records. `http://valid.canardpc.com/records.php`. Accessed: 14.10.2016.

[DBB+16] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. An Implementation and Evaluation of the MPI 3.0 One-Sided Communication Interface. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2016.

[For16] MPI Forum. Status of MPI-3.1 Implementations. `http://mpi-forum.org/mpi31-impl-status-Jun16.pdf`, June 2016. Accessed: 07.10.2016.

[GBH13] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 53:1–53:12. ACM, Nov. 2013.

[GHTL14] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface (Scientific and Engineering Computation)*. The MIT Press, 1 edition, 11 2014.

[GLS14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface (Scientific and Engineering Computation)*. The MIT Press, third edition, 2014.

[GT07] William D. Gropp and Rajeev Thakur. *Revealing the Performance of MPI RMA Implementations*, pages 272–280. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[Ham87] W.-R. Hamann. *Line Formation in Expanding Atmospheres: Multi-Level Calculations using Approximate Lambda Operators*, page 35. 1987.

[HB99] Peter H. Hauschildt and E. Baron. Numerical solution of the expanding stellar atmosphere problem. *Journal of Computational and Applied Mathematics*, 109(1–2):41 – 63, 1999.

[HB06] P. H. Hauschildt and E. Baron. A 3D radiative transfer framework : I. Non-local operator splitting and continuum scattering problems. *Astronomy and Astrophysics*, 451(1):273–284, may 2006.

[HBA97] Peter H. Hauschildt, E. Baron, and France Allard. Parallel Implementation of the PHOENIX Generalized Stellar Atmosphere Program. *The Astrophysical Journal*, 483(1):390, 1997.

[HDB+13] T. Hoefler, J. Dinan, Darius Buntinas, Pavan Balaji, B. Barrett, R. Brightwell, W. D. Gropp, V. Kale, and Rajeev Thakur. MPI + MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory. *Computing*, 95:1121–1136, 2013.

[HDT+13] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 1(1):29, March 2013.

[HDT+15] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26, June 2015.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[JRS16] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition*. Morgan Kaufmann, 2 edition, 7 2016.

[LHL+15] Mingzhe Li, Khaled Hamidouche, Xiaoyi Lu, Jian Lin, and Dhabaleswar K. (DK) Panda. *High-Performance and Scalable Design of MPI-3 RMA on Xeon Phi Clusters*, pages 625–637. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[Mel06] Mellanox Technologies. Mellanox InfiniScale : Eight-Port 10 Gb/s 4X InfiniBand Switch. `https://www.mellanox.com/pdf/products/silicon/InfiniScale_PB_1.0.pdf`, 2006. Accessed: 30.09.2016.

[MPIa] MPI Forum. MPI 4.0. `http://mpi-forum.org/mpi-40/`. Accessed: 28.09.2016.

[MPIb] MPI Forum. MPI Forum. `http://mpi-forum.org/`.

[MPI03] MPI Forum. *MPI-2 : Extensions to the Message-Passing Interface*. MPI Forum, `http://mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf`, November 2003. Accessed: 28.09.2016.

[MPI12] MPI Forum. *MPI : A Message-Passing Interface Standard : Version 3.0*. MPI Forum, `http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, September 2012. Accessed: 28.09.2016.

[MPI15] MPI Forum. *MPI : A Message-Passing Interface Standard : Version 3.1*. MPI Forum, `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, June 2015. Accessed: 28.09.2016.

[P.16] Mike P. An Intro to MCDRAM (High Bandwidth Memory) on Knights Landing. `https://software.intel.com/en-us/blogs/2016/01/20/an-intro-to-mcdram-high-bandwidth-memory-on-knights-landing`, January 2016. Accessed: 14.10.2016.

[Pad11] David Padua, editor. *Encyclopedia of Parallel Computing (Springer Reference)*. Springer, 2011 edition, 9 2011.

[Sau16] Marc Sauter. Intel veröffentlicht Xeon Phi mit bis zu 7 Teraflops. `http://www.golem.de/news/knights-landing-intel-veroeffentlicht-xeon-phi-mit-bis-zu-7-teraflops-1606-121642.html`, June 2016. Accessed: 13.10.2016.

[See08] Andreas Michael Seelmann. 3D Strahlungstransport : Erste Rechnungen. mathesis, Universität Hamburg, February 2008.

[See11] Andreas Michael Seelmann. *3D radiative transfer in arbitrary velocity fields: The Eulerian approach.* phdthesis, Universität Hamburg, 2011.

[Squ14] Jannek Squar. Einsatz von Beschleunigerkarten für das Postprocessing großer Datensätze. Online `http://edoc.sub.uni-hamburg.de/informatik/volltexte/2015/209/pdf/bac_squar.pdf`, 12 2014.

[SV16] Faith Singer-Villalobos. Stampede 2 Drives the Frontiers of Science and Engineering Forward. `https://www.tacc.utexas.edu/-/stampede-2-drives-the-frontiers-of-science-and-engineering-forward`, June 2016. Accessed: 14.10.2016.

[TGR+09] Vinod Tipparaju, William Gropp, Hubert Ritzdorf, Rajeev Thakur, and Jesper L Traff. Investigating high performance rma interfaces for the mpi-3 standard. In *2009 International Conference on Parallel Processing*, pages 293–300. IEEE, 2009.

[TGT05] Rajeev Thakur, William Gropp, and Brian R. Toonen. Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication. *IJHPCA*, 19(2):119–128, 2005.

[TOP] TOP500. Earth-Simulator. `https://www.top500.org/system/167148`. Accessed: 18.10.2016.

[VVH+15] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar (DK) Panda, Darren Kerbyson, and Adolfy Hoisie. A Case for Application-oblivious Energy-efficient MPI Runtime. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 29:1–29:12, New York, NY, USA, 2015. ACM.

[YIU+02] Mitsuo Yokokawa, Ken'ichi Itakura, Atsuya Uno, Takashi Ishihara, and Yukio Kaneda. 16.4Tflops Direct Numerical Simulation of Turbulence by a Fourier Spectral Method on the Earth Simulator. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

Appendices

List of Figures

2.1 Latency of RMA communication operations . . . 14
a Internode put . . . 14
b Intranode put/get . . . 14
2.2 Latency of GATS synchronisation . . . 14
2.3 Comparison between 10-Gigabit Ethernet and InfiniBand regarding performance features . . . 15
a Latency . . . 15
b Throughput . . . 15
c CPU requirement . . . 15
d Memory-network-traffic ratio . . . 15
2.4 RMA memory models . . . 16
a Separate memory model . . . 16
b Unified model . . . 16
2.5 UMA architecture . . . 18
2.6 Cluster architecture with distributed memory . . . 19
2.7 Kinds of shared memory . . . 20
a No shared memory . . . 20
b RMA shared window . . . 20
c Complete shared memory . . . 20
2.8 Development of performance . . . 21
2.9 Spawning new processes during runtime . . . 24
2.10 Child communication . . . 25
a Parent processes manage and redirect child-communication . . . 25
b Direct communication between child-processes . . . 25
2.11 RMA initialisation operations . . . 29
2.12 Maximum shared memory . . . 31
2.13 Accessing window with shared memory . . . 31
2.14 RMA communication operations . . . 32
2.15 RMA synchronisation operations . . . 34
2.16 Fence synchronisation . . . 36
2.17 GATS synchronisation . . . 37
2.18 State diagram for correct MPI RMA synchronisation . . . 39

3.1 Two different kinds of characteristics . . . 44
a Short characteristic . . . 44
b Long characteristic . . . 44
3.2 Concurrent wavelength clusters (based on [See11, 24]) . . . 45

3.3 Program flow of PHOENIX's test program to check the 3DRT mode . . . 46
3.4 Visualisation of the three parallelisation stages of PHOENIX 47
3.5 Concurrent wavelength clusters with optimal speed-up 48
3.6 Too many wavelength clusters in the pipeline induce idling 48
3.7 Vtune parallelisation visualisation 50

4.1 Development of 53
a ... CPU clock rate 53
b ... core and thread parallelism 53
4.2 Intel Xeon Phi Knights Landing 55
4.3 Composition of a tile [JRS16] 56
4.4 Schematic representation of the Knights Landing architecture [JRS16] . . 57
4.5 VTune: Memory bandwidth and latency 59

5.1 General scheme of the execution of one step 61
5.2 General workflow of TrackerSim irrespective of the chosen variation . . 62
5.3 Passive target communication with shared locks visualised with Vampir 67
5.4 Current load distribution in PHOENIX with OpenMP 74
5.5 Adapted load distribution in PHOENIX without OpenMP 75
5.6 Example for a very low error deviation 78

6.1 Average core utilisation displayed by micsmc, process initialisation through TrackerSim becomes apparent 80
6.2 Benchmarks of every variation of TrackerSim with no load and four processing elements on Linux-PC 81
6.3 Benchmarks of every variation of TrackerSim with no load and four processing elements on Knights Corner 82
6.4 Benchmarks of every variation of TrackerSim with no load and four processing elements on Knights Landing 82
6.5 Weak scaling of TrackerSim on Linux-PC 84
6.6 Weak sclaing of TrackerSim on Intel Xeon Phi 85
a Knights Corner 85
b Knights Landing 85
6.7 Strong scaling of TrackerSim on Linux-PC 86
6.8 Execution of the first step of *TrackerSim-SharedNoRMA* 86
a VTune: outer locks, five iterations 86
b Vampir: outer locks, one iteration 86
6.9 Execution of the first step of *TrackerSim-SharedNoRMA* 87
a VTune: inner locks, five iterations 87
b Vampir: inner locks, one iteration 87
6.10 Strong scaling of TrackerSim on Intel Xeon Phi 88
a Knights Corner 88
b Knights Landing 88

6.11 Benchmark of PHOENIX on Linux-PC: small grid $(8 \times 8 \times 8)$ and few solid angles (2×2) 89

List of Listings

3.1 General parallelisation scheme in PHOENIX [AWH16] 46
3.2 Load balancing of the solid angles executed by every process 49

5.1 Abstraction of Phoenix' access pattern with OpenMP (Step 1) 63
5.2 Abstraction of Phoenix' access pattern with OpenMP (Step 2) 64
5.3 Abstraction of Phoenix' access pattern with OpenMP (Step 3) 64
5.4 TrackerSim with active, collective synchronisation (Step 1) 65
5.5 TrackerSim with active, collective synchronisation (Step 2) 65
5.6 TrackerSim with active, collective synchronisation (Step 3) 66
5.7 TrackerSim with passive synchronisation (Step 1) 68
5.8 TrackerSim with passive synchronisation (Step 2) 68
5.9 TrackerSim with passive synchronisation (Step 3) 69
5.10 Example of the creation of a shared window in TrackerSim 70
5.11 TrackerSim with passive synchronisation on shared Memory with local load/store (Step 1) 71
5.12 TrackerSim with passive synchronisation on shared Memory with local load/store (Step 2) 72
5.13 TrackerSim with passive synchronisation on shared Memory with local load/store (Step 3) 72
5.14 main 77
5.15 LC-tracker 77
5.16 Child 77

List of Tables

2.1 RMA memory models 17
a Separate memory model 17
b Unified memory model 17
2.2 Overview over common MPI implementations 27

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik (M.Sc.) selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

Ort, Datum	Unterschrift