TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation

Improving Hybrid Codes Through MPI-Aware OpenMP

David Büttner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Martin Bichler

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Arndt Bode
2. Univ.-Prof. Dr. Michael Bader

Die Dissertation wurde am 18.05.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 01.08.2016 angenommen.

Acknowledgements

I would like to thank Prof. Dr. Arndt Bode for giving me the opportunity to do my doctorate at his chair, the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR), for his continuous support and the freedom he granted me for my research. I would also like to thank Dr. Carsten Trinitis and Dr. Josef Weidendorfer for their supervision of my work and their continuous support.
A lot of thanks also goes to my colleagues for creating an enjoyable work environment and all the great conversations.

Furthermore I would like to thank Prof. Dr. William Jalby for giving me the opportunity to work at the Exascale Computing Research Laboratory in Versailles, France. I would also like to thank Dr. Jean-Thomas Acquaviva for many fruitful discussions and all his help during my time in Versailles.

Finally, I would like to thank my wife Lena Büttner for her continuous support over the years and my good friend Bert Heinrich for all the time he invested in me. Without either of them, this work would not have come to a successful end.

Abstract

With the number of cores growing faster than memory per node, hybrid programming models have become a popular approach for efficient use of high performance computing (HPC) systems. Using threads instead of MPI ranks inside compute nodes replaces intra-node messaging with faster communication through shared memory access. Additionally, the number of parallel inter-node messages can be optimized. In large scale parallel applications, communication often influences the runtime of applications. Achieving good efficiency by overcoming this bottleneck is challenging, even when using asynchronous communication. While hardware support for asynchronous data transfer exists, most MPI implementations can only advance pending asynchronous operations inside library calls.

In this thesis a classification of programming approaches and existing implementations of the MPI and OpenMP standards are used as the basis to discuss this problem. An analysis of asynchronous capabilities of different MPI implementations on different hardware environments shows that in nearly all cases no real asynchronous operations are provided. A strategy using the hybrid MPI-OpenMP programming approach to overcome this problem is developed. It improves asynchronous behavior by introducing MPI-awareness to the OpenMP runtime through a new type of task. Being scheduled together with the regular OpenMP work-pool, it can be used to advance pending asynchronous MPI operations with minimal overhead. At the same time it keeps the advantages of the hybrid programming approach, including easier work balancing as compared to an MPI-Only approach.

For different state of the art HPC systems, the performance of this approach is measured and analyzed. Without complicating the programming interface, the results show perfect overlap of the asynchronous operations with the performed computational work. Additionally it is shown that this approach reduces the influence of communication on performance. This approach has a lot of performance potential for parallel HPC applications which can be parallelized in a hierarchical manner, i.e. hybrid `MPI-OpenMP`. Especially for future exascale systems it reduces communication costs and relaxes synchronization requirements. It can be expected that this also positively influences scalability.

Zusammenfassung

Ausgehend von einer Klassifizierung paralleler Programmieransätze und insbesondere einer Analyse des MPI-Standards wird das Problem der asynchronen Kommunikation eingehend untersucht. Verschiedene Implementierungen des Standards werden untersucht und auf verschiedenen Hardwareplattformen gezeigt, dass asynchrone Kommunikation in der Regel nicht erfolgt. Ein neuartiger hybrider MPI-OpenMP Programmieransatz wird entwickelt, um dieses Problem zu beheben. Mittels geeigneter MPI-Erweiterungen im OpenMP Standard wird ein neuartiger OpenMP-Task vorgestellt. Dieser kann dynamisch von der OpenMP-Laufzeitumgebung verwendet werden, um ausstehende asynchrone MPI-Operationen bei minimalem Mehraufwand anzustoßen.

Contents

CHAPTER 1

Introduction

1.1. Motivation

While earlier trends in modern processor design were able to achieve higher performance through increased processor clock frequencies, this trend came to a halt due to increasing power consumption and heat development. Instead of trying to increase clock frequencies further, multi-/many-core architectures have emerged, providing increased performance potential through multiplication of processor components. Increased performance is provided through multiple, but slower clocked, computational cores, which can be used in parallel. Access from all cores to the same physical main memory provides very fast communication between threads or processes. In order to minimize the memory access bottleneck, a series of fast memory (caches) is integrated. In order to increase the available core counts, multiple sockets - each hosting multiple cores - make up large compute nodes. While single socket systems provide an environment where access to any memory region from any core takes the same amount of time (i.e. uniform memory access (UMA)), multi-socket systems provide a non uniform memory access (NUMA) environment. Different parts of the physical memory are attached to different sockets. While the operating system provides a global address space, the actual access latencies differ depending on the distance between used core and the desired memory region.

One area where performance of programs is very important is high performance computing (HPC). Many applications, both scientific and industrial, cannot be executed efficiently on single processors, either due to the amount of necessary memory, or due to the number of computational steps necessary. These applications can often be split into a set of subproblems which can be executed in

parallel. They can be assigned to processes or threads running on different compute resources, i.e. the application can be parallelized. Seeing as in most cases the subproblems are not independent, some kind of communication between the processing units is necessary. In order to provide hardware environments for these kinds of applications, different high performance computing systems exist. Using multi-/many-core architectures as described above is one of them. An additional level of hardware parallelism is added by connecting compute nodes through communication networks. Multi-/many-core environments are limited by the number of cores and amount of memory which can be placed into a single compute node, but provide very efficient communication between the computational cores. Connecting compute nodes through networks is scalable to very large node counts, but the communication is limited through the used network. While modern networks are very efficient, they cannot stand up to shared memory communication.
Recent developments in high performance computers, as can be seen in the TOP500[1] list, combine both hardware setups, providing large hierarchical hybrid systems. Networks connect large numbers of compute nodes, resulting in a distributed memory environment. Each compute node internally provides a multi-/many-core environment.

In order to use the parallel hardware, different programming paradigms have been created. Based on the used communication method, two major parallelization paradigms are defined: shared memory and distributed memory parallelization. Shared memory parallelization is mostly based on the many-/multi-core architectures described above, using multiple threads to work on a single problem. The most widely used standard for shared memory parallelization is `OpenMP` [62]. It defines a set of compiler directives, library routines and environment variables which can be used to parallelize programs at a thread parallel level. It takes care of thread creation and management, work item definition and distribution, and provides tools for thread synchronization, etc. The `OpenMP` standard especially targets the programming languages `C` and `Fortran` and is usually used in an incremental parallelization approach. Starting with a sequential program, pragmas, library calls, etc., are added step by step, adding parallelism to different parts. Introducing basic parallelism to a program is relatively easy to achieve, but advanced parallelization optimizations are possible.
In regard to distributed memory parallelization, the most widely used approach is the Message Passing Interface (`MPI`) [55]. `MPI` is not a programming language, but a definition of an interface to library routines which offer all necessary functionalities in regard to distributed memory parallelization. The main focus of the standard is point to point communication, but it also covers areas such as remote memory access, process creation and high performance parallel I/O.
With modern HPC systems, as described above, the hardware is not necessarily

[1]www.top500.org

matching one or the other programming paradigm, but incorporates aspects of both approaches. A common way to program parallel systems made of many multi-core compute nodes is to regard every core as a separate compute unit and use `MPI` only (`MPI-Only`). On every core of the system, one `MPI` process, or rank as they are called, is executed and communication is done via message passing, regardless of whether or not the communicating ranks are located on the same node or have to use the communication network. Smart implementations for the used `MPI` library can recognize the fact and optimize the data movement, but direct access to the memory regions of other ranks on the same node is not possible. Nevertheless, this approach can yield good parallelization performance, as will be discussed later. An alternate approach is to combine both programming paradigms in a way which matches the used hardware environment. At a higher level, one (or a few) `MPI` ranks are placed per node, communicating via message passing using the communication network. At a lower level, the ranks internally parallelize their workshare using a shared memory programming paradigm in order to make use of fast intra-node communication.

Especially with distributed memory parallelization, communication can become a real bottleneck, and strategies to remove, reduce or hide communication behind useful workloads are very important. Many parallel applications have two kinds of workloads which are being assigned to the used processes: communication independent work, i.e. computation which can be executed without the need to exchange data with another process, and communication dependent work. The latter is a result of the parallelization and the fact that work items can depend on the values of other work items. Due to the fact that these might be assigned to a different process, some data exchange needs to happen either before or after this work can be done, as those values can change over time.
One way to reduce communication overhead is to look for opportunities to hide communication behind communication independent work. The goal is to make use of the provided resources in such a way that a scheduled communication can be handled without keeping the processor occupied to do so. Therefore, the processor can work on communication independent work at this time.
Communication overlap, as this is called, has been discussed in literature (see Chapter 4 for references), and has been integrated into the `MPI` standard. The standard offers so called nonblocking communication functions, which post a communication request to the `MPI` runtime without blocking the calling process. Nevertheless, no current `MPI` implementation which can be found on HPC systems available to this work was actually able to achieve real communication overlap when using these functions, as will be discussed later. Communication overlap has been discussed in different publications and the general agreement is that overlap can only be achieved either through the use of progress threads or by manually advancing communication while working on communication independent work. Progress threads are threads spawned by each `MPI` rank. They take care of communication related steps, such as programming network hard-

ware. Progress threads are not always possible or feasible. Manual progression, i.e. taking care of calling progression functions in the user program, is difficult to implement. Especially the question of when and how often available advancement functions need to be called is hard to answer efficiently and no automated way to do so exists.

1.2. Contribution

This work addresses the questions discussed in the motivation. Especially automated message progression in the context of the hybrid `MPI-OpenMP` programming approach. The proposed `commtasks` add awareness of the `MPI` parallelization used on the higher level to the `OpenMP` implementation used for parallelization inside each `MPI` rank. Together with a new `OpenMP` schedule, these `commtasks` can achieve perfect communication overlap by calling message progression functions at optimal times. Additionally, the new approach introduces NUMA awareness to the `OpenMP` parallelization while providing work balancing at the same time. Additionally, the approach removes timing constraints in regard to the exact communication times during the execution of a parallel program. Communication dependent steps, both progression and communication dependent work, can automatically be assigned a higher priority in order to make sure that idle times in this regard are minimized.
In order to build a foundation for the proposed `commtask`, the communication overlap capabilities of the available `MPI` libraries are being investigated through a benchmark. The results show that the `MPI` libraries are indeed able to not only achieve synchronization overlap, but also data transfer overlap, which together result in the desired communication overlap. The benchmark also reveals the `MPI` libraries' limitations in regard to communication overlap. A formal definition of the `commtask` is being presented, showing how the automated progression can be included into the `OpenMP` standard with minimal changes. The performance of the proposed approach is being investigated by applying it to a relevant parallel code, namely the Jacobi Relaxation method. This method represents three dimensional stencil codes, which can be found in a large variety of applications. Using different stencils and different approaches to split the used computational domain to the available compute resources, the `commtask` approach is being compared to a large number of traditional implementations, including `MPI-Only` and `MPI-OpenMP` both implemented twice: using blocking and using nonblocking communication functions.

The results show that for real life parallel applications, automated message progression and perfect communication overlap is possible using the `commtask` approach. It is easy to add to hybrid parallel applications (`MPI-OpenMP`) and does not require fundamental changes in the existing parallelization standards. For programs with contiguous memory buffers used in the communication, the

results show perfect overlap and the proposed `commtask` implementations outperform all other implementations. These include `MPI-Only` and `MPI-OpenMP` implementations, both implemented using blocking as well as nonblocking communication functions. As shown with the benchmark mentioned above, limitations in regard to overlap exist in cases where the used memory buffers are noncontiguous. While the proposed approach cannot outperform other approaches like `MPI-Only` in this case, the hybrid implementations can be improved drastically, even to the point where their performance matches the performance of the `MPI-Only` counterparts.
The presented work shows that hybrid parallelization on hybrid hardware has a lot of potential to improve the performance of parallel applications and optimize the usage of HPC resources. For a large set of parallel applications, this can be done using the proposed approaches. For other applications, performance improvements can be seen in comparison to the classic hybrid implementations and possible future extensions promise better results in these cases as well.
The proposed `commtask`, while implemented to achieve overlap using progression function, is designed to also work with progress threads, which might exist in future `MPI` implementations. Its positive impact on performance might even be increased through the removal of the overhead necessary to call progression functions yet keeping its other advantages at the same time: NUMA aware work balancing, prioritizing communication related work steps and minimizing the impact of delayed communication partners.

1.3. Structure of Thesis

Following the introductory chapter, an introduction to High Performance Computing parallel programming models is presented in Chapter 2. Besides an overview of parallelization and the possible programming paradigms, two of the most prominent, and for this work most relevant, programming paradigms are presented in detail: the shared memory parallelization approach `OpenMP` (Section 2.1) and the distributed memory parallelization standard `MPI` (Section 2.2). The chapter is concluded by a closer look at their combination in a hybrid `MPI-OpenMP` fashion (Section 2.3).
Chapter 3 proposes a new `OpenMP` loop parallelization schedule, namely `static-ws`, which addresses the need of `OpenMP` to be aware of non uniform memory access (NUMA) environments while providing efficient work balancing.
The need for asynchronous MPI functionality, such as communication computation overlap, is presented afterwards. The capabilities of modern HPC systems and software environments are benchmarked in detail in Chapter 4. Based on the results a solution is presented. Adding `MPI` awareness to the `OpenMP` runtime in hybrid parallelization approaches through the proposed `commtask` is used to achieve real asynchronous `MPI` functionality. The chapter finishes with a formal definition of the `commtask` consistent with the `OpenMP` standard in Section 4.4.

The impact of the `commtask` on real live applications is then extensively analyzed in Chapter 5. A large variety of parallelization options for three dimensional stencil codes is being compared on a large set of HPC systems. These include `MPI-Only` and `MPI-OpenMP`, both implemented using blocking and nonblocking communication functions, as well as implementations of the proposed `commtask`. For two computational domain decomposition strategies, namely one- and three-dimensional domain decomposition, the implementations are executed on large node counts of modern hybrid HPC hardware and the results are discussed in detail.
Finally, the last chapter concludes the work and presents an overview of related future research topics.

CHAPTER 2

Parallel Programming Models

As discussed in Section 1.1 the use of high performance computing (HPC) systems is widely spread and multi-core environments are becoming a part of desktop computers, notebooks and even mobile phones. Processor frequencies are decreasing in exchange for lower power consumption and multiple instances, creating multi-core compute nodes. Together with high performance networks, these nodes make up large parallel systems. In order to use these efficiently, i.e. dedicate all (or parts of) the systems to work on a single computational problem, the common programming interfaces have to be extended in regard to parallelization.
Using a pool of resources to work on a single problem results in some kind of distribution of work items. These can for example be data blocks which need to be used for some kind of computation or different computational phases which need to be applied to data passed through them. In most cases these work items are not independent from each other and some kind of communication needs to happen between the used computational cores. For these parallel programming approaches, two major forms of parallelization are typically distinguished: shared memory parallelization and distributed memory parallelization.
For environments providing a common (virtual) main memory to all participating computational cores, as is the case in multi- and many-core compute nodes, parallelization can be done by using multi-threading. The threads have a shared memory region and communication can happen through direct access to the shared data buffers. This form of communication is fast, but limited in scalability. Access to the shared data has to be coordinated by the programmer. It can be extended to be used on multiple hardware nodes by providing virtual shared memory, but this removes the advantage of fast access times and is not commonly applied in HPC applications. Common parallelization paradigms for

shared memory parallelization are `OpenMP` and `Pthreads`.
For environments providing computational units distributed to different nodes connected through a network, the distributed memory parallelization approach has to be used. The provided memory spaces on the nodes can not directly be accessed by all used computational cores and communication has to be done by sending data through the available network. One common way to exchange data is through message passing. The de facto standard for message passing parallelization is the Message Passing Interface (`MPI`).
Other parallelization exist, such as the use of graphical processing units (GPUs) for computation, implemented for example using `Cuda` or `OpenCL`.

Independent of the chosen parallelization paradigm, the parallelized applications have useful work, which is directly related to the problem to be solved. When splitting the work or the work steps and assigning these work items to different workers (threads or processes), additional steps need to be taken care of. This overhead, while necessary, is not part of the original algorithm and needs to be minimized where possible. This work includes synchronization between workers, copying and sending data in distributed memory environments, etc.
For this work, the important parallelization paradigms are the Message Passing Interface (`MPI`) for distributed memory parallelization and the use of multithreading (`Pthreads` and `OpenMP`) for shared memory parallelization as well as the combined use for hybrid parallelization. Both parallelization paradigms, `MPI` and `OpenMP`, are widely accepted in the HPC community and it can be expected that they will be relevant in the future. Nevertheless, as can be seen by looking at research done in regard to new and different programming models, new challenges will have to be addressed on the way to even larger systems. In order to face these challenges, the current `MPI` and `OpenMP` standards will have to be adjusted and extended. One possible way to ensure their relevance might be their interaction, as proposed later in this work.

The rest of this Chapter will give an introduction of these programming paradigms, discuss related work and highlight important aspects in regard to the work presented later. The goal of the introduction is to present the concept behind the presented programming paradigms, explaining the different steps which are necessary in order to get a basic parallel program. Additionally, the most relevant extensions for advanced parallelization are presented and more detail is discussed in regard to the aspects used later in this work. The introduction is not discussing detailed semantics of the programming paradigms, as those are explained in detail in the referenced standards, which are available to anyone from the corresponding web pages.

2.1. OpenMP

With the introduction of multi-core architectures, real parallelism was enabled inside a compute node. A program can not only be programmed to have multiple execution paths by the use of threads, but these can truly run in parallel on the available hardware. Through the provided shared memory environment, the threads can access the processes global address space directly and therefore communicate in a highly efficient way. In order to exploit this parallelism, programming models and paradigms have been created to address the different aspects arising through the shared memory parallelization, which include:

- Thread management: This includes the creation of threads, their termination, identification, etc.
- Synchronization: Coordination of possibly concurrent accesses to the same memory region, e.g. to make sure that an expected update of the corresponding value has already been written by another thread.

Many programs which are targets for parallelization have different phases. Often a very compute intensive central part takes up the major part of the execution time. In cases where parallelism can be exploited here, the use of multiple threads has the most impact. The other parts are often program setup and initialization phases in the beginning and the saving of results and termination phases at the end. These latter parts are often neither long in comparison to the central part nor do they exhibit enough useful parallelism to exploit.
For these kinds of programs shared memory parallelization paradigms are needed which are easy to use, provide all the necessary tools and are able to make efficient use of the available resources. While all of this can be done on a lower programming level using for example `Pthreads`, the amount of work necessary on the programming side reserves this approach for programming specialists and usually makes it hard to use for application specialists[1]. A simpler parallel programming interface for shared memory parallelization which has evolved as a widely used standard is `OpenMP`. In order to "standardize directive-based multi-language high-level parallelism that is performant, productive and portable" [58], the `OpenMP` Architecture Review Board (ARB) has been created. While at the beginning of this work, the `OpenMP-3.0` [59] standard was the newest version, versions `OpenMP-3.1` [60], `OpenMP-4.0` [61] and `OpenMP-4.5` [62] have been published since then, adding new features and support for new hardware environments.

The standard describes the three basic parts of `OpenMP`:

- Compiler directive (extending the programming languages `C`, `C++` and `Fortran`)
- Library routines

[1]Specialists in areas such as physics, chemistry, etc. whose programs benefit from parallelization but whose expertise is not computer science/engineering.

- Environment variables

The standard approach for programming `OpenMP` is incremental. Starting with a sequential program, pragmas and library calls are added afterwards. This allows for easy and quick parallelization of core parts of a program, as the original algorithm can be implemented without taking parallelization into consideration from the beginning. Nevertheless, as is the case with any program, adding more specialization (e.g. knowledge about the memory layout of used data structures, access patterns, etc.) to the implementation results in more potential for performance gain.
Different `OpenMP` compilers exist, such as the `GOMP`[2] OpenMP implementation for the `GNU Compiler Collection`[3]. As most of the systems available for this work provide specially tuned software, another implementation used mostly throughout the results presented later is the Intel C/C++ Compiler (ICC).

Research in regard to OpenMP is ongoing and covering a wide range of areas. The authors of [18] propose sub-teams of threads and the option to assign work to them via worksharing constructs, which is now available in `OpenMP`. Other work proposes extensions in regard to the tasking construct [85]. The concept of places has been added to `OpenMP` as well, which has been the topic of [27], whose authors also proposed affinity policies. Affinity in regard to memory placement in NUMA domains or `OpenMP` places is still missing but has been addressed by the authors of [76]. They propose an OpenMP extension which offers memory initialization in regard to used places and the option of migrating memory, either automatically or through proposed runtime library functions. The use of `OpenMP` in non uniform memory access (NUMA) environments is also the topic of [8]. Extending OpenMP for NUMA architectures, the authors of this paper implement methods taken from the parallel programming paradigm High Performance Fortran in an OpenMP compiler. They propose a user directed approach for page migration, i.e. memory movement, and user directed data layout mechanisms. The authors of [84] examine the behavior of task-parallel codes on NUMA systems, comparing different approaches in regard to the task creation. Due to the fact that the behavior of task scheduling for OpenMP is less defined, offering a more dynamic work distribution, the efficient use of tasks in NUMA environments is not guaranteed.
Another area, which is important in high performance computing is efficient parallel I/O, which has not been addressed in the `OpenMP` standard. The authors of [52] introduce a parallel I/O interface for `OpenMP` in order to close this gap.
The authors of [1] address the increasingly important question of power management in the proposed energy extension OpenMPE. Other work examines the energy consumption of Haswell processors running OpenMP programs, by applying and evaluating different energy saving strategies [92]. Classic performance

[2]`https://gcc.gnu.org/projects/gomp/`
[3]`https://gcc.gnu.org/`

evaluation is also part of ongoing work. The authors of [73] presents results of the SPEC OMP2001 Benchmarks on a Hitachi SR8000 node, showing that it can achieve a high performance for applications with high demands on the memory bandwidth. The Jacobi method, used in the results presented in Chapter 5, is part of research as well, including the work presented in [6] where it has been implemented using different paradigms, including `MPI` and `OpenMP`, executed on Cray XE6.

2.1.1. Parallelization and Work Distribution

Compiler Directives

The most basic compiler directive is the `parallel` construct. It defines a parallel region and creates a team of threads which executes the code inside the constructs range. Thread creation does not have to be implemented manually. The pragma can be supplied with parameters defining the behavior in regard to used variables, i.e. which variables are being used by each thread privately, which variables are being accessed by all threads and how to define the values of the variables at the beginning and after the parallel region. In order to distribute the work to the created threads, different directives exist in regard to worksharing. These include, but are not limited to, the `for` directive for loop parallelization, the `sections` and the `single` constructs. Another way to define structured code blocks which can be distributed to available threads, being executed in parallel, is the tasking construct. While the former construct has been part of `OpenMP` from the beginning, the `OpenMP tasks` have been added in later versions.
In order to control the execution flow of the threads and synchronize them, `OpenMP` offers additional directives. Code inside the `parallel` region can be defined to be executed by a single thread using the `single` directive (any thread) or the `master` directive (execution only by the thread who encountered the current `parallel` region). Access to shared variables can be coordinated with constructs such as the `critical` or `atomic` directives.
As threads have a local view on shared variables, the programmer has to take care of making changes public to other threads by the use of the `flush` directive.

Runtime Library

The runtime environment of `OpenMP` parallelized programs can be controlled through a series of runtime library functions. These include functions to control the number of threads used in `parallel` regions, query the state of the runtime, change settings such as scheduling decisions made at runtime and routines controlling locking mechanisms, timing functionality and device memory management.

Later in this work, these runtime library functions are being used to do manual worksharing in respect to the thread identifier assigned to the used `OpenMP` threads.

Environment Variables

The defined environmental variables can be used to influence the behavior of the programs globally by either setting them before the program execution or using the corresponding runtime library functions in the code. An important variable is `OMP_NUM_THREADS`, which defines the number of threads used for a defined `parallel` region.

2.1.2. Work Scheduling and Balancing

As described above, `OpenMP` offers different ways to distribute work to the available threads inside a given `parallel` region. In regard to this work, the two important approaches are the loop parallelization directive `for` and the `task` directive.

The `#pragma omp for` construct can be used to split the iteration space of a given loop. Depending on the parameters and the defined scheduling the mapping of the indices to the available threads is done. The used loops must be conforming to the `canonical loop form`, defined in the `OpenMP` standard [62, 2.6 Canonical Loop Form]. The behavior when applying this to loops with dependencies between the different iterations is undefined.
The possible scheduling clauses which can be supplied to the `for` construct are:

- `static` (Syntax: `schedule(static,` *`chunk_size`*`)`)
- `dynamic` (Syntax: `schedule(dynamic,` *`chunk_size`*`)`)
- `guided` (Syntax: `schedule(guided,` *`chunk_size`*`)`)
- `auto` (Syntax: `schedule(auto)`)
- `runtime` (Syntax: `schedule(runtime)`)

Using the `static` schedule, the iteration space is distributed to all available threads in equal sized blocks (no *chunk_size* is specified) or by distributing blocks of size *chunk_size* in a round-robin fashion. For the schedule clauses `dynamic` and `guided` threads request chunks whenever they executed the last received chunk. For `dynamic`, the amount of iterations per chunk are fixed and defined through *chunk_size*. For `guided`, "the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team" [62] decreasing to one (default) or to the minimum size defined through *chunk_size*. Schedule `auto` delegates the question of which schedule to choose to the runtime and compiler. With `runtime` the schedule is chosen at runtime

depending on the internal control variable (ICV) `run-sched-var`, which can be set either using environment variables or the runtime library. Depending on the parallelized loop, different advantages are offered through the different schedules. For loops which are entered by all threads at the same time and which exhibit a uniform execution time for each iteration, the use of schedule `static` is most efficient. In cases where the threads might reach the at varying points in time or where the duration of each iteration is variable, the schedules `dynamic` and `guided` offer work balancing at the cost of overhead for the chunk assignment.
Important for this work is the question of iteration assignment across different loops, different encounters of the same loop and ac-cross different `parallel` regions, as the question of which thread on which core accesses which memory region is influencing the efficiency of code when using `OpenMP` in NUMA environments.

2.1.3. OpenMP in NUMA Environments

Modern compute nodes providing multiple computational cores are split into different sockets. Each socket hosts a number of cores. The main memory is partitioned and the individual partitions are connected to one of the available sockets. While a thread on a given core can access any memory partition available in the node through a global address space, not every access is the same. The latency and bandwidth between the used core and the desired data in memory is dependent on the question where the hosting memory partition is located. This kind of environment is called a non-uniform memory access (NUMA) environment [8, 84].
For efficient parallelization with shared-memory paradigms like `OpenMP` different aspects have to be taken care of in regard to memory access. These include:

- Thread pinning: As in HPC applications the best practice is to use one thread per available core, the threads are usually pinned to one core each.
- Data pinning: Across multiple recurring regions accessing the same data, the data to thread distribution is fixed in order to allow the efficient use of caches and, in case of NUMA environments, the minimized access latency and bandwidth.

Thread pinning can be done using different methods, which are often supplied through the scheduling system on the HPC systems or through the use of available libraries. Making sure that the data a thread is being assigned is located on a memory partition close to the core the thread is pinned to, is more difficult. On modern operating systems in combination with the hardware, a widely used approach in regard of memory allocation is the `first-touch policy`. At memory allocation points, these systems make a reservation of the virtual memory only and defer the allocation of physical memory to the time it is being accessed for the first time [8, 32, 68, 71–73, 94].

For `OpenMP` parallelized applications, this can be a problem. First of all, as mentioned above, the typical incremental way of adding `OpenMP` directives to the sequential program for compute intensive parts of the code does not usually include memory initialization and initialization. This has to be considered by the programmer and parallelization of the initialization phase in regard to the `first-touch policy` must be done. Even when using the same loop logic for initialization and the computational phase, the problem using `OpenMP` is the question whether or not the same iterations are assigned to the same threads for both parts. Only for the schedule `static` does the `OpenMP` standard define a behavior in this regard. The assignment of logical loop iterations to the available threads must be the same for two loop regions, when the following criteria is met [62]:

1. Both loop regions have the same number of loop iterations
2. Both loop regions have the same value of `chunk_size` or no specified `chunk_size`
3. Both loop regions bind to the same parallel region
4. Neither loop region is associated with a `SIMD` construct

For all other schedules, the iterations assignment is unspecified.
Especially criteria number 3 can be a problem. The corresponding parts of the code and therefore also the used `parallel` regions are usually not close together and not easily consolidated. Nevertheless, for the compilers used for this work (GNU gcc and Intel ICC), tests showed that the same iteration assignment is being done in cases where the respective loops do not define a `chunk_size`, are enclosed in `parallel` regions using the same threads and have the same number of loop iterations. (No `SIMD` construct has been used)
For cases where work balancing is desired, i.e. `dynamic` or `guided` should be used, no support for NUMA environments exist. To address this, a new scheduling clause, namely `static-ws` is proposed in Chapter 3 and applied in the implementations done for Chapter 5.

2.2. MPI

While multi processor compute nodes and parallel programming paradigms like `OpenMP` provide programmers with the tools and environments to parallelize their code, this approach is mostly limited by hardware constraints. This can be the limited main memory available to the cores on a compute node or the amount of computational cores itself. To overcome these limitations, distributed memory parallelization offers the possibility to make use of hardware (computational cores and memory) which are distributed across multiple compute nodes which are connected via a communication network. Numerical simulation codes, which

often communicate in a “bulk-synchronous” fashion, have strong communication requirements and are the reason for the need of fast networks in HPC systems and highly efficient ways to use them [33]. Processes running on the cores of these compute nodes do not have direct access to all available memory. This can be simulated through virtual shared memory implemented in software [90], which is also the idea behind the Partitioned Global Address Space (PGAS) programming paradigm. Nevertheless, the most commonly used approach to program for distributed memory parallelization is message passing.
Processes sharing the work for one parallel program communicate through explicit messages, exchanging data between the corresponding virtual and physical memory spaces. With the Message Passing Interface (`MPI`) standard the Message Passing Interface Forum created and develops a “message-passing library interface specification” [55]. MPI is not a programming language but a definition of an interface to library routines which offer functionality in regard to everything concerning distributed memory parallelization. In addition to point to point communication functions, functions in many different groups are defined such as collective operations, remote-memory access operations, functions in regard to process creation and areas such as high performance parallel I/O operations. The standard defines all operations in a generic manner, providing language bindings for the programming languages C (used for the implementations in this work) and Fortran.
The goals of the standard are standardization, portability of the code and the definition of a library which can be optimized for each available system without having to change the user programs.

Environmental Management

Programs using an `MPI` implementation and the provided library will be executed in a parallel environment. These environments can be set up in different ways and run different operating systems. `MPI` does not define how an `MPI` program is to be started, how the environment is to be setup or what needs to be done on the system side in order to execute an `MPI` parallelized program. Nevertheless, code must be runnable on all systems without a change and the environment needs to be provided with a possible setup phase. Therefore the initialization function `MPI_Init` is required to be called before any other calls to the `MPI` library is done.
In later versions of the `MPI` standard, the need for dynamic process management was addressed and an interface between `MPI` and the environment, such as the process management systems, has been added. Nevertheless, the standard emphasizes that this must not interfere with a consistent definition of `communicators`, which are described below.
Analog to the `MPI_Init` function, the `MPI_Finalize` function must be called in order to finalize the `MPI` program and no call to the library can be done afterwards.

Communication Contexts and Topologies

The created processes of an `MPI` program are called `ranks`. Each rank can exist in a series of contexts. With the start of the program, after the use of `MPI_Init`, all ranks are part of the global context, defined through the global `communicator` `MPI_COMM_WORLD`. `Communicators`, which among other things encapsulate the ideas of communication contexts, groups and virtual topologies, have been added to the standard in order to make `MPI` compatible with the creation of support libraries. They allow a subset of processes (possibly all) to be associated with a given task and the corresponding communication. Within the scope of this work, implementations either use `MPI_COMM_WORLD` only or create a new `communicator` including all ranks in order to virtually place them into a used `topology` (s. Section 5.1). Inside a `communicator`, each rank is assigned a rank identification, which will be referred to simply as rank.
Process topologies can be used to virtually map the available ranks in order to facilitate the naming of ranks and selection of corresponding communication partners. It also provides information about the expected communication partners to the `MPI` environment. This allows the remapping of processes to the available hardware in order to match the communication pattern to the available communication network pattern. Functions are provided which help programmers to select communication partners in the topology without having to manually match the (one-dimensional) rank numbers from the `communicator` `MPI_COMM_WORLD` to the (possibly multi-dimensional) communication topology.

In theory every `MPI` process can be provided with its own source code, which has to match the same context, communication, topology, etc. patterns as for all other ranks. In most cases all processes are provided with the same source code, which executes the same code, making distinctions in execution paths in regard to the corresponding rank and communicator information. In order to parallelize programs using this approach, different functionality must be provided, as is the case for shared-memory parallelization paradigms like `OpenMP` as discussed above. Communication between the ranks and synchronization functionality are part of these. Different types of communication defined in `MPI` are “Point-to-Point Communication”, “Collective Communication” and “One-Sided Communication”, which will be discussed below.

MPI and Threads

For hybrid parallelization, such as the combination of `MPI` with a shared memory parallelization paradigm like `OpenMP`, `MPI` defines different `thread safety levels`. These address the question whether or not and when multiple threads can access the same `MPI` function. Creating thread safe implementations for a library such as `MPI` is not only more difficult, but also requires mechanisms which take care of thread synchronization inside the library. This might influence performance as well, which is not desired in the case where no threads are used. In order to satisfy all needs, four different thread levels exist, which can be supported by an MPI implementation, probably through the use of multiple implemented versions of the same:

1. MPI_THREAD_SINGLE
2. MPI_THREAD_FUNNELED
3. MPI_THREAD_SERIALIZED
4. MPI_THREAD_MULTIPLE

Depending on the requirements of the program, the thread level must be confirmed using MPI_Init_thread instead of MPI_Init especially if another level than MPI_THREAD_SINGLE is desired. This gives the MPI implementation the option to setup anything necessary to support threads from its implementation aspects.
While MPI_THREAD_SINGLE is used for MPI-Only implementations, hybrid approaches can request increasing thread support with the other three levels. MPI_THREAD_FUNNELED assumes that threads exist and will be used, but only the main thread makes calls to MPI functions. The main thread is the thread who calls MPI_Init/MPI_Init_thread and MPI_Finalize. This might be the case when MPI is used in combination with OpenMP and calls to MPI are made outside OpenMP parallel regions only. In cases where multiple threads need to make calls into the MPI library functions, but do not need to make these at the same time, the necessary thread safety level is MPI_THREAD_SERIALIZED. Any thread can call MPI functions, but the programmer has to take care that no two threads do so at the same time. Finally, in cases where all threads must be able to call MPI functions, even at the same time, the necessary thread safety level is MPI_THREAD_MULTIPLE. Nevertheless, some restrictions still apply, such as the fact that no two threads can complete the same request (see the following Section for more information on requests). The authors of [32] give an extended overview in regard to the thread safety levels.

MPI Datatypes

In order to exchange data between processes all communication functions need to be provided with a description of the data which is to be exchanged. MPI offers some basic datatypes, describing elements like integers (MPI_INT) or doubles (MPI_DOUBLE). When calling a communication function, the data is defined by a type and a repetition count together with a buffer, which is expected to contain the corresponding number of elements of the requested type (or has enough space to store them) sequentially. While the standard requires the matching of MPI types, it guarantees representation conversion in cases where messages are exchanged in a possibly non homogeneous environment.

Many scenarios are covered by the basic datatypes, as the data to be sent is a series of elements located sequentially in a memory buffer. Nevertheless, these kind of message buffers are too restrictive as different algorithms require the option to communicate different kinds of elements or noncontiguous data in a single message. The latter is the case in the implementations presented in Section 5.3. While it would be possible to send two messages with two kinds of data for the first case or to locally pack the required data into a sequential buffer in the second case, this includes overhead which `MPI` library implementations can try to minimize. One option for example would be avoiding unnecessary memory to memory copies. Implementations optimized for a special HPC environment can also make use of specialized hardware, such as processors located on the network interface cards (NIC), if information about the data layout of message buffers is provided. To solve this, `MPI` defines derived datatypes, through which advanced data layouts can be described and made public to the `MPI` environment. A derived datatype uses either basic datatypes or previously created derived datatypes together with a layout description in order to define a memory pattern. One example used later in this work is `MPI_Type_create_subarray`. For a memory buffer describing a three dimensional array of elements, it allows the definition of the surface planes of the array. Through the information provided to the function call, the `MPI` implementation can then decide whether it is more efficient to pack the data into a sequential buffer locally and send/receive one message, send/receive multiple small messages or to apply other optimizations. In order to use a derived datatype in any kind of communication function, it needs to be committed using `MPI_Type_commit`.

Point-to-Point Communication

The basic form of communication described in the `MPI` standard is point-to-point communication. Two ranks communicate with each other with one rank being the source of the data (the sender) and one rank being the destination of the data (the receiver). For both the send and the receive operation used, the communication partners are identified through the used `communicator` and the corresponding rank numbers in regard to this `communicator`. The data to be communicated is defined through a (send or receive) buffer, the `MPI` datatype used in the buffer and the repetition count of the datatype. In order to be able to match messages in cases where two multiple messages are sent from one sender to the same receiver, it is possible to label the message using a message `tag`. There are two basic types of point-to-point communication functions: Blocking and nonblocking versions. The details on how these types work and what their definitions imply will be discussed in the following Section 2.2.1, as these functions are at the center of the work presented later in Chapters 4 and 5.

Collective Operations

Collective operations are communication functions involving a group of ranks. They include synchronization functions such as `MPI_Barrier`, data distribution functions such as `MPI_Bcast` and data collection function functions such as `MPI_Reduce`. The corresponding group of ranks which is part of the collective operation, and therefore has to take part in it, is defined through the chosen communicator.
Those collective functions which are used to transfer data have to be provided with additional information. This includes information about the source of the data or the target destination. Also, consistent with the syntax of the point-to-point communication, message buffers have to be provided, which are further defined using the datatypes described above. Function such as `MPI_Reduce` apply an operation to the data which has to be either a function provided through `MPI` or a user defined operation. Predefined operations include `MPI_MAX` and `MPI_SUM` in order to select the largest value of all values supplied by the ranks or receive the sum of all elements respectively.

One-Sided Communication

While the communication approaches described above facilitate the use of communication by combining all necessary aspects into the respective communication functions, this is not always desired. Especially the fact that communication and synchronization are both combined can be a restriction in some cases. In order to overcome this, `MPI` defines one-sided communication. Through the definition of so called windows, a region in one ranks memory can be made visible and accessible to another rank. Using the one-sided communication functions, this region can then be accessed (read, write) by the other rank without having to post a matching communication function call on the memory hosting ranks side. The necessary synchronization between the ranks, e.g. making sure that the desired (updated) data resides in the memory before reading it locally, needs to be implemented by the programmer.
While this works well in theory, `MPI` libraries do not provide perfect implementations for this, as they internally rely on synchronization mechanisms [75]. This results in similar problems as encountered when trying to use nonblocking communication functions to overlap communication and computation, as described later in this work.

2.2.1. Theory of Asynchronous Communication

As described above, point-to-point communication defines the transfer of data from a source rank to a target rank. While the information provided to the chosen communication functions describe the communicating ranks and the data together with corresponding message buffers, the time of transmission is defined

through the chosen communication function. The two basic groups of point-to-point communication functions are divided into blocking and nonblocking communication functions. Blocking communication functions, such as `MPI_Send` and `MPI_Recv`, return when the used message buffer is save to be accessed again. For the sending operation this means that changing the content of the buffer after the blocking send returns does not change the data which is received on the other end. For the receiving operation, this implies that the message buffer contains the communicated data. The details of when the functions return is dependent on the `MPI` implementation and the internally used communication approach.
For the send operation, different communication modes exist and it is up to the `MPI` environment to choose which one is used, unless it is specified further through the use of specialized send operations. The system may or may not choose to copy the message envelope (information and message buffer) into a temporary buffer in order to free the send buffer. The additional memory copy operation might slow down performance and the available buffer space can be limited. In order to provide portable programs, implementations of the standard `MPI_Send` operation must not rely on buffer space. If message buffering is desired, message buffers can be defined manually and their use be enforced with `MPI_Bsend`.
Depending on the implementation and the communicated data, the behavior of `MPI_Send` can vary in another aspect also: For messages which are buffered, the send operation can return before a matching receive operation has been posted. For cases where the message is copied directly into the receiving buffer in the target ranks memory, the send operation blocks until all data has been transferred. In order to provide information about the status of the receiving rank, `MPI` offers two additional send operations. `MPI_Ssend`, or synchronous send, will only return when a matching receive operation has been posted and has started to receive data. `MPI_Rsend`, or ready send, may only start when a matching receive operation has been posted, or the behavior of the program is erroneous. This function may be used when this precondition can be guaranteed in order to remove possible rendezvous points, possibly improving performance.

In addition to these blocking communication functions, the `MPI` standard offers nonblocking communication functions. Overlapping communication with computation is possible in theory when the computation can be split into two parts, namely communication dependent and communication independent work. The communication independent work can be done at the same time as communication in order to hide synchronization and data transfer. Using specialized hardware, this overlap is possible in theory through splitting the definition of the communication envelope and the completion confirmation. Nonblocking, or immediate, send and receive initialization functions (e.g. `MPI_Isend` and `MPI_Irecv`) are used to provide information about the message buffers and the communication partners early. They return immediately without guarantees

about the status of the desired communication. The corresponding message buffers may not be used before a matching communication termination function has successfully confirmed that the message buffers are no longer needed in order to finish the communication. These termination functions include `MPI_Wait`, which returns only when the message buffers are free to be used, or `MPI_Test`, which returns immediately together with information about the communication status. The same modes as for the blocking send operation (i.e. standard, buffered, synchronous and ready) can be applied to the nonblocking send initialization functions.
In order to match nonblocking send initialization calls to the used termination functions, each communication is identified through an `MPI_Request`. The request object is needed internally to match the corresponding library calls to the used communication mode and the respective message buffers. In cases where multiple communications need to be initialized, the `MPI` standard offers variations of the `MPI_Test` and `MPI_Wait` calls which can be supplied with an array of requests. These extensions include versions waiting or testing for the completion of all supplied requests (`MPI_Testall` or `MPI_Waitall`) and versions checking or waiting for the completion of (at least) one of the supplied requests (`MPI_Testany` or `MPI_Waitany`).
Instead of using `MPI_Wait` or `MPI_Waitall`, it is also possible to wait for completion by regularly calling the respective test function, i.e. `MPI_Test` or `MPI_Testall`. This is important in regard to the work presented in later chapters. The standard allows for the implementations to not guarantee the synchronization and data transfer overlap. It might be the case that any steps related to the communication is done during the used communication termination functions. This implies that calling `MPI_Test` repeatedly will eventually take care of all necessary steps. While this is the case, no implementation on the high performance computing (HPC) systems used for this thesis provided real data transfer overlap but take care of the message progression during the calls to `MPI_Wait`. Benchmarks and measurement results will be presented in Chapter 4.

Two communication partners do not necessarily have to use the same kind of point-to-point communication. I.e. nonblocking send operations can be matched by blocking receive calls and blocking send operations by nonblocking receive functions. Nevertheless, the order of functions is important. The first send operation executed by the sending rank will be matched with the first receive operation by the receiving rank. This is important in regard to matching message buffer sizes and used datatypes. Nevertheless, the order of the corresponding communication termination functions is independent of this order.

2.3. Hybrid MPI-OpenMP

The hierarchical design of current HPC systems combines the environments described in the previous Sections [32]. Large HPC systems are built using large numbers of compute nodes, which are being connected through high performance networks, providing a distributed memory environment. The compute nodes consist of multi- or many-core systems, internally providing a shared memory to the available cores. While it is possible to simulate a shared memory environment throughout the entire system or to place `MPI` ranks on every core, a combination of both approaches matches the hardware setup. As mentioned in the previous Section, it is possible to combine the use of `MPI` with multi-threading approaches such as `OpenMP`. This can be used to assign multiple cores, e.g. an entire node or a single socket in a NUMA environment, to an `MPI` rank and parallelize the ranks work using a shared memory parallelization paradigm. Applying the necessary thread safety level through the corresponding call to `MPI_Init_thread` sets up the `MPI` environment accordingly.

This kind of hybrid approach is part of ongoing research and different trends emerge when looking at the results. A common opinion is that `MPI-Only` is the best approach, but different examples exist [32]. `MPI-Only` is compared to `MPI-OpenMP` using the NAS parallel benchmarks on an IBM SP system by the authors of [15]. They conclude that in most cases MPI only is performing better than the hybrid approach. The tendencies necessary for the hybrid approach to be attractive are said to be very fast processor cores reducing the amount of work in relation to the communication and sufficient potential for multi-level parallelization. This is also discussed by the authors of [79]. They demonstrate that a hybrid approach is not always better than an `MPI-Only` approach on clusters of SMP nodes, but neither is it always worse. They conclude that the hybrid approach can be beneficial in cases where pure `MPI` has problems with scaling. Reasons for this include load balance problems or memory limitations when too much data has to be replicated inside each rank. Also, limitations on the number of possible `MPI` ranks can exist.
Nevertheless, the question which approach is better can even come down to the chosen input sizes, input data, etc. for a fixed given code, as demonstrated by the authors of [49].

One aspect important to the work presented in later chapters is the interaction of the `MPI` libraries with the `OpenMP` runtime. Both standards are developed independently and while aspects from each are taken into consideration in the other, the interaction is not seamless. One runtime/compiler framework, namely MPC, addresses this problem by providing a low overhead environment for hybrid programming [16]. While this would be ideal for this work, it is not available on any HPC system used later and not widely used in general.
Nevertheless it is an important step in the direction of understanding, facilitating

and optimizing hybrid parallelization approaches. Similar issues are discussed by the authors of [32], stressing the fact that a mismatch between the hybrid hardware and the parallel programming models exists. They conclude that machine topology awareness is important for efficient (hybrid) parallelization on such hardware. This is also supported by the results presented by the authors of [72], where `MPI-Only` and `MPI-OpenMP` are compared in regard to latency and bandwidth aspects. One piece of hardware information useful here is the fact that a single core on modern HPC compute nodes might not be able to saturate the available inter-node bandwidth, which is important when using a single `MPI` rank per node.

Finally, hybrid implementations are also part of research working with stencil codes as done later in this work. The authors of [26] implement a multilevel parallelization framework using a sixth order stencil code. Their intra-node performance and optimization results show that for a 6th point stencil, the use of 8 threads on an 8 core system is better than using the hyper threading cores and using 16 threads. This is due to the memory requirements of the application. Further related work on stencil codes and different implementation options can be found in Chapter 5.

2.3.1. Potential of Hybrid MPI-OpenMP Parallelization

When looking at the question of efficient and scalable `MPI` implementations and the mismatch between the distributed memory parallelization approach and the hierarchical hardware, a set of limitations is being discussed [32, 71, 79]. The first problem of `MPI-Only` approaches can be poor scalability. Reasons for this include large load imbalances between the ranks or restrictions in regard to fine grained parallelization. Both can be removed or at least minimized when using multiple threads inside the rank, working on a large grain subproblem. The fine grained distribution is done in regards to threads and work balancing can be implemented at this level as well.
Another problem is data replication on the rank level, e.g. for ghost cells as defined later in Chapter 5, and therefore in the context of a cores main memory. With newer HPC systems, where memory per core is increasing at smaller rates as the core per node counts, this will be a definite problem in the future [66]. Using `MPI` on a node level or a socket level reduces the replication to the context of the nodes or sockets memory, respectively.
Also, restrictions on the number of ranks which can be chosen due to the nature of the chosen algorithm might exist. This can include limitation to use powers of 2 for the rank count due to efficiency reasons with the `MPI` collective communication. The number of usable cores can be increased through hybrid implementations, loosening up the restrictions while keeping the same rank count.
Finally, the ease of programming is an aspect of interest. Programming `OpenMP` is relatively easy and the overhead of adding `OpenMP` to `MPI` codes is relatively

small.

These problems are not only theoretical, but have been discussed for applications such as weather modeling [49]. The three main advantages expected here are the reduction of message latencies through message consolidation, network contention prevention and load balancing. A single rank using multiple threads sends the same amount of data as in an `MPI-Only` implementation, but does so in a single large message instead of multiple small ones. Also, by reducing the amount of concurrent messages, congestion of the network interface is reported to be avoided. This is an interesting point, as the authors of [32] reported that on modern HPC systems a single core is able to saturate the network, but stress the fact that this might not always stay true. In the future, the use of a subset of cores, i.e. reducing the number of messages to a few instead of only one, might be the optimal solution. Finally, the weather simulation code was expected to benefit from the fact that dynamic work balancing can be done on the thread level. Nevertheless, as discussed above, the results of the work done in [49] showed that the question which approach (`MPI-Only` or `MPI-OpenMP`) is better comes down to the chosen problem sizes and input parameters. Positive results through the use of threading inside the `MPI` ranks is reported by the authors of [26], while the authors of [34] conclude that the hybrid approach with the existing runtimes and libraries can not outperform `MPI-Only` to publication date.

CHAPTER 3

NUMA Aware Work-Scheduling and Work-Balancing for OpenMP

As described in Section 2.1, using OpenMP in non-uniform memory access (NUMA) environments adds additional optimization requirements on `OpenMP` parallelized applications. Especially when looking at the `OpenMP for` pragma for parallelizing the execution of loops, these include the placement of memory in relation to the core on which the corresponding thread is running. While parallel environments usually offer thread pinning to prevent the operating system to move threads to different cores, memory placement has to be considered by the programmer. There is no direct support in `OpenMP` in regard to memory placement, which is partially due to the fact that memory placement is part of the underlying operating system and hardware environment. As discussed above, the usual approach is the so called `first-touch policy`. On allocation, only the virtual address space is reserved and actual memory placement on physical memory is done when the corresponding memory regions are accessed the first time.
As `OpenMP` parallelization is usually done in an incremental approach, adding `OpenMP` pragmas to compute intensive parts of a sequential program only, the memory initialization phase is not parallelized. The main thread, which is executing all the non parallelized parts of the program, takes care of the memory initialization and, as a result, all memory regions are placed physically close to the core this thread is pinned to. In order to overcome this limitation, parallelizing the initialization phase can be done by applying the same parallelization to it in regard to memory accesses as to the computation phase. Nevertheless, further limitations exist. Only for the criteria discussed in Section 2.1 does the `OpenMP` standard guarantee the same logical loop iteration assignment between two loops. As the initialization phase and the computation phase are usually not inside the same part of the program, these are not easily met.

It has been observed that different compilers do relax these criteria. The same schedule is applied in cases where the same number of threads execute different parallelized loops associated with different `parallel` regions in case they have the same number of logical iterations, but only when the schedule `static` is used. For codes which are suited for such a schedule, the memory to core mapping can be solved. For codes which require one of the other available schedules, i.e. `dynamic` or `guided`, this is not the case and no guarantees in regard to core to memory distance can be done. The first-touch and then reuse approach is not achieved in this case.

The authors of [8] propose using extensions found in High Performance Fortran for `OpenMP` Fortran, adding optimization strategies to the `OpenMP` compiler. The authors of [76] address memory affinity and an extension for memory migration in the context of the new `OpenMP` places. Another area where memory accesses in NUMA environment can have a large impact is `OpenMP` tasks. The distribution of tasks to available `OpenMP` threads is even less predictable than logical iterations is for loops. In order to better understand memory placement for the tasking model, the authors of [84] investigate different tasking strategies on NUMA systems.
While all these approaches show benefits in regard to the current `OpenMP` standard for NUMA environments, they all include additional steps from the programmer or complicated extensions to the runtime and compilers. For the work presented later, a simpler approach will be proposed in the remainder of this chapter.

3.1. OpenMP Schedule `static-ws`: Introduction

In order to extend the `OpenMP` schedules `dynamic` and `guided` to work well in NUMA environments, the logical iteration distribution to the available threads has to be addressed. Assuming that the threads are being pinned to the available NUMA cores, as is also necessary when using the schedule `static` in a NUMA environment, these two schedules have to be investigated in regard to possible memory reuse options. As the `OpenMP` standard does not specify the order in which the chunks are distributed to available threads, it may not influence the outcome of a program. This can be used to combine the advantages of schedule `static` with those of the schedules `dynamic` and `guided`.
Let `static-ws` ("static work stealing") be a schedule which can be used in the same manner as the other three mentioned schedules. This proposed schedule will be described here and more formally defined later in this chapter. Distributing the logical iteration space the same way as the existing schedule `static`, it can be used in the context of a `first-touch policy` aware memory initialization. Each thread is assigned a block of logical iterations, for which the accessed data regions are therefore located physically close to the core it is running on.

In a hierarchical manner, each iteration block is split into chunks in the style of either the `dynamic` or `guided` schedule. Threads, either when entering the loop or after finishing an assigned chunk, will request new chunks as is the case when using these schedules originally and the runtime will assign the next thread following these rules:

- The next available chunk from the thread's higher level static block, if one is available.
- The next available chunk from another thread, if one is available.

Using this approach makes sure that each thread works on memory physically close to the core it is pinned to as long as possible while preserving work balancing. On systems with optimized `OpenMP` runtimes, the rules above can be extended with knowledge about the hardware environment:

- The next available chunk from the thread's higher level static block, if one is available.
- The next available chunk from another thread pinned to a core on the same socket, if one is available.
- The next available chunk from any other thread, if one is available.

In order to optimize cache behavior and prefetching, the chunks chosen from different threads can be taken from the logical end of the other threads' higher level iteration block. A thread from which a chunk is assigned to another thread is therefore working on a logically sequential set of iterations as long as possible.

In regard to the lower-level sub blocks, i.e. the chunks assigned to a thread, different strategies can be applied. The two obvious strategies follow the existing schedules `guided` and `dynamic`. Other approaches are possible and are subject to future research.

3.2. OpenMP Schedule `static-ws`: Definition

In order to formalize the proposed schedule `static-ws`, different parts of the `OpenMP` standard have to be extended. As the schedule is not a new worksharing construct, but part of the loop parallelization construct, all definitions necessary follow the parts defined in regard to the other schedules, especially `static`. The first part is the description of the schedule, which needs to be added to Table 2.5 of the `OpenMP` standard (see [62, 2.7.1 Loop Construct]). Small modifications are necessary in regard to the `omp_set_schedule` and `omp_get_schedule` functions which can be used to set and read the value of the internal control variable (ICV) `run-sched-var` (see [62, 2.3.1 ICV Descriptions - `run-sched-var`]).

As this is a proposed excerpt of the OpenMP standard, it is kept short and precise. The proposed schedule stands by itself, but will be used and evaluated in the context of the proposed commtask in the following chapters.

static-ws Description - OpenMP Standard Table 2.5

When **schedule(static-ws, sub_schedule, chunk_size)** is specified, iterations are divided into two levels. At a higher level, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of these higher level chunks is unspecified.
A compliant implementation of the **static-ws** schedule must ensure that the same higher level assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions bind to the same parallel region, and 3) neither loop is associated with a SIMD construct. The same conditions apply in regard to two loop regions, one being assigned the schedule static and the other the schedule static-ws.

Depending on the specified sub_schedule, the higher level chunks are subdivided into chunks according to the corresponding chunk_size specifications for the possible options. These are dynamic or guided.

Each thread executes a chunk of iterations from its own lower level chunks of iterations. It then requests another chunk, until no chunks remain from the thread's own lower level chunks. When this is the case, it requests a lower level chunk from another thread's higher level iterations. The order of chunk assignment is unspecified and possible optimizations are subject to specific implementations of the standard.

static-ws Description - OpenMP Standard run-sched-var ICV

In order to keep the changes to the existing functions to a minimum, the type definitions need to be extended by two values. While static-ws technically is a single schedule, it has an additional parameter (for the lower level chunk splitting). Adding omp_sched_static-ws_dynamic = 5 and omp_sched_static-ws_guided = 6 to the type definition allows for omp_set_schedule and omp_get_schedule to be unchanged.

Setting the internal control variable (ICV) run-sched-var can also be done using the environment variable OMP_SCHEDULE, whose definition needs to be adjusted as well.

The value of this environment variable takes the form:

type[[,subtype],chunk]

where

- *type* is one of `static`, `dynamic`, `guided`, `static-ws`, or `auto`.
- *subtype* is an optional parameter defining the low level chunk definition when *type* is `static-ws`.
- *chunk* is an optional positive integer that specifies the chunk size.

3.3. Discussion

`OpenMP` is one of the most prominent parallel programming paradigms in regard to shared-memory parallelization. As discussed in Section 2.1.3, the question whether or not the used shared-memory environment provides a uniform memory access environment (UMA) or a non uniform memory enfironment (NUMA) is important when trying to optimize `OpenMP` parallelized programs. The need for NUMA aware strategies has been discussed above. One simple approach, namely adding the new schedule `static-ws` to the `OpenMP` `for` worksharing construct, has been proposed and formalized here. With minimal changes to the existing standard, it provides a basic tool to optimize memory access and introduces efficient work balancing at the same time.
Should the proposed changes be included into the `OpenMP` standard, programmers would be able to add NUMA aware parallelization of `OpenMP` `for` loops with minimal changes to their code. The hierarchical iteration space distribution allows for easy mapping of higher level chunks of iterations, and therefore memory regions accessed, to used threads and the cores they are pinned to. So far, this has only been possible with the schedule `static`. At the same time, the schedule takes into account the need for work balancing, similar to the existing schedules `dynamic` and `guided`.
The proposed schedule will be implemented and applied to parallel applications later in this work.

CHAPTER 4

Hybrid MPI-OpenMP: Real Asynchronous MPI-Functionality through MPI-aware OpenMP Runtimes

Nonblocking communication, as described in Section 2.2.1, has been defined in the first MPI standard MPI-1.0 [54, Chapter 3.7: Nonblocking Communication]. MPI functions have been defined for all necessary steps of nonblocking communication. Instead of one communication function taking care of all communication steps, functions for the different steps have to be used. From the MPI interface user side, this includes communication initialization and communication termination. Depending on the MPI implementation and the available hardware, the standard allows for different ways of data movement. In environments with suitable hardware, the data movement (or message progression) can be done any time between the communication initialization and termination, in parallel to the computation, resulting in the desired overlap. In other environments, where this is not possible, the standard leaves the decision of when and how to take care of the data movement to the MPI implementation.
While the first standard states that "the send start call will return before the message was copied out of the send buffer" [54], this has been updated to "The send start call can return before the message was copied out of the send buffer" [55]. Similarly, "a nonblocking receive start call [...] will return before a message is stored into the receive buffer" [54], changed to "a nonblocking receive start call [...] can return before a message is stored into the receive buffer" [55]. This indicates what has been discussed in literature and will be shown in the following sections: The use of the nonblocking communication functions does not necessarily guarantee that real overlap of communication and computation is being achieved.

Like any other type of communication, asynchronous communication consists of two parts: Synchronization of the communication partners and the actual data transfer. Both can in theory be overlapped with, i.e. hidden behind, useful communication independent work. The synchronization step is necessary to set up different aspects regarding the communication. These are MPI implementation specific and include aspects such as message buffers and the mode of data transfer, which depends on factors like message sizes, available buffer space, etc. One important question is whether or not the communication requires a rendezvous-protocol, i.e. all communication partners to be calling into the MPI library at the same time. The second part, data transfer, consists of the actual sending of the data from the send buffers to the receive buffers. Challenges in regard to overlap exist for both parts. In regard to synchronization overlap, they include the question of how it can be achieved without blocking the respective MPI ranks inside the communication initialization functions. In regard to data transfer, it is not clear whether or not one communication partner can make communication progress while the other one is not calling into the MPI library. These question will be discussed in detail in Section 4.1.

While communication overlap is possible in theory, it is not always guaranteed or available on current HPC systems with modern MPI implementations. While synchronization overlap is available in most MPI implementations, data transfer overlap cannot be observed in any of the HPC systems available for this work when using the standard implementation approach. In order to understand how the different MPI implementations behave in regard to communication overlap, a benchmark will be presented in Section 4.2. In addition to analyzing the behavior of programs using standard MPI nonblocking communication functions, it is designed to evaluate if and how well manual message progression is supported by the tested MPI implementation. Message progression, i.e. taking care of asynchronous communication related work steps outside the calls to communication initialization or termination functions, can be done either using progress threads or by manually calling message progression functions. As the first is not implemented in current MPI versions available on modern HPC systems, it is not targeted by the benchmark.
Possible results of benchmark executions are being evaluated in detail and are discussed in regard to the conclusions which can be drawn in regard to internal MPI library behavior. The benchmark has been tested on a set of modern HPC systems using different MPI implementations. The results show that no overlap is provided through the tested implementations and all communication related steps are moved to the communication termination functions. The use of manual progression functions, `MPI_Test` in case of the benchmark, does provide overlap, but with some restrictions. One of the most important aspects is the timing of the call to the message progression function and the size of the message buffer which is to be transferred. Nevertheless, for cases where the timing is correct and the used amount of calls to the message progression function match the sys-

tem requirements, perfect overlap of communication with the used computation phase of the benchmark can be observed. Details and additional aspects are discussed in Section 4.2.

With the knowledge that communication overlap can be achieved using manual progression, the question arises how this can be achieved automatically. Implementing it manually is very complicated and too much overhead is easily introduced in the corresponding code. Additionally, the aspects which need to be considered can be different between used HPC systems and optimizing them for one platform can result in bad performance on another. Normally no calls to the `MPI` library are done in between the communication initialization and termination functions.
As discussed in previous chapters, the hybrid parallelization of `MPI` and `OpenMP` offers many advantages. It matches the hybrid design of modern HPC system hardware and offers potential for other optimizations such as work balancing inside the used compute nodes and therefore `MPI` ranks. Parallel hybrid `MPI-OpenMP` programs using nonblocking communication are usually implemented to parallelize the communication independent work between the communication initialization and termination functions using `OpenMP` worksharing constructs. Therefore the `OpenMP` runtime is active during the time where message progression functions need to be called in order to achieve communication overlap. Section 4.3 discusses how a hybrid `MPI-OpenMP` environment can be used to achieve automatic message progression by making the `OpenMP` runtime aware of the outstanding `MPI` communication. The presented approach, namely `commtasks`, will be presented and additional advantages discussed. These include include automatic, parallel and prioritized scheduling of work related with communication. Both work directly related to communication (e.g. calls to the `MPI` library and message progression functions) as well as communication dependent work.

4.1. Challenges of Asynchronous MPI-Functionality

While theoretical overlap using nonblocking MPI functions for asynchronous communication and I/O is possible, this is not always available in actual runtime environments. The reason for this is that the MPI runtimes do not provide efficient support for asynchronous progress [11,96]. The MPI standard discusses the support for making progress on pending nonblocking operations. This can be interpreted in two ways [11]: Using a strict interpretation, the outstanding operations make progress independent of subsequent calls to the MPI library. With a more relaxed interpretation, the implementation can require periodic calls to the MPI library in order to advance progress.
Being addressed in different publications [11, 33, 35, 77], the need to be able

to advance pending messages between the start of the communication and the corresponding `MPI_Wait` call really exists. There are different ways to achieve this [35]. The two major ideas are using a progress thread or advancing the messages manually. Not many MPI implementations provide a progress thread and none of those that do could be found on the HPC systems accessable for this work. One MPI implementation which experimented with progress threads is OpenMPI. Nevertheless, they have never been stable and have been removed with OpenMPI 1.6 [28]. Also, using a progress thread is not always efficient [35]. Most current MPI implementations advance pending messages during calls to the MPI library. But, with no need to call any MPI related function during the communication-independent computation, there is no message progression and the actual communication is moved to the `MPI_Wait` call, which will be shown in the results of the benchmark presented in the following Section 4.2.

The reasons why no message progression is done, which are being discussed in literature, are the need for a handshake algorithm before the actual message transfer and regular hardware programming [77].

Going into more detail, different capabilities of the MPI implementation have to be looked at in order to understand how data is transferred from a send buffer in the memory of the sending MPI rank to a receive buffer in the memory of the receiving rank. Independent of whether or not nonblocking communication functions are used, this question arises, if the implementation supports independent progress: Can one rank complete a [blocking] send or receive operation, while the communication partner is not calling into the MPI library. As will be described later, most MPI versions use different approaches depending on the size of the message to be transferred. Small messages can be sent by the sending process without the need for involvement of the receiving side by transferring the entire message directly. On the receiving side, these small messages, on reception, will be stored in temporary buffers and moved to the actual MPI message receive buffer as soon as a matching receive call is encountered. For large messages it is not guaranteed that there is enough temporary buffer available on the receiving side and the actual data can only be transferred whenever the corresponding receive buffer has been made available inside the matching receive function. Nevertheless, depending on the implementation of the actual message transfer, independent progress might still be possible in theory. An implementation supporting independent progress in this case might send the send buffer information in a small message and have the receiving rank get the data through a remote direct memory access (RDMA) read without the need of the sending process to be calling an MPI function at the same time. Nevertheless, this approach needs additional support from the computing environment.

When looking at nonblocking communication, the additional question is, whether or not the MPI implementation supports the desired communication overlap, which has been described in Section 2.2.1. If it does, an MPI rank can do useful computations at the same time as the available hardware and software take care of message related operations. An MPI implementations can support two kinds of overlap [96]:

- Synchronization overlap, and
- data transfer overlap

The communication related work includes message matching and protocol processing [11]. For an MPI rank it is possible to send or receive multiple messages to and from multiple communication partners at the same time. Incoming messages might arrive before, during, or after a matching `MPI_Recv` call and have to be matched to the corresponding blocking or nonblocking receive call, which provides the information on where receive buffers are allocated and at what point in time they are available. Depending on the implementation of the actual data transfer, this information has to be processed, e.g. communicated to the message source, used in possible RDMA-read operations or for copying messages from temporary buffers to their final destinations, etc. Depending on the temporal order in which two matching `MPI_Isend` and `MPI_Irecv` calls are done on two communicating MPI ranks, not all necessary information might be available from the communication partner. This can be a problem for the second kind of overlap, data transfer overlap, for the message transmission implementations described below.
data transfer overlap, in cases where synchronization has successfully been done, consists in the actual transfer of the message data. This includes the computation of all necessary steps in the communication software stack, programming the used hardware, etc.

As mentioned above, most MPI implementations implement the message transmission, for both blocking and nonblocking communications, depending on the message size [35]. The two versions used are the eager- and rendezvous-protocol. With the eager transmission, small messages are sent asynchronously and are buffered on the receiver side until a matching receive operation copies it into the corresponding receive buffer. For large messages, the transmission is being delayed until the matching receive is encountered and the final receive buffer is available, which is confirmed at a rendezvous point. At that point, the data can then be transferred in different ways, which include RDMA read from the receiving rank, RDMA write from the sending rank and pipelined messages [98]. In order to synchronize, the rendezvous protocol requires at least two messages in addition to the actual message transfer [35]. When implementing the transfer using a pipelined message strategy, the sender uses another series of messages for the actual transfer of the data. In order to send these, two approaches are common. When implementing a polling approach, a user level thread needs to query the hardware continuously. The interrupt based approach needs support from the operating system (OS) in order to notify the user thread whenever a message, or part of a message, has to be processed. In order to achieve shorter point to point latencies, current MPI implementations use the polling method, avoiding the operating system. For MPI libraries using InfiniBand, this is implemented for example in OpenMPI and MVAPICH [35]. Nevertheless, using the polling based approach is not the most time efficient, because the polling thread is runnable, even if it yields its CPU time slice when it has nothing to do. It will be rescheduled according to the OS thread scheduling algorithm.

For both the rendezvous and eager protocol, the question remains whether or not the work necessary on either side of the point to point communication can be overlapped, and in cases it can, what kind of overlap is supported.

Research in regard to communication and communication-computation overlap has been done analyzing the different aspects described above. Different benchmarks have been proposed in this area. The authors of [43] introduce a benchmark suite to assess the overall ability of MPI implementations to achieve communication-computation overlap. Another benchmark separates the tests for synchronization overlap and data transfer overlap [96]. The results of the latter benchmark show that all tested implementations support synchronization overlap while no implementation supports data transfer overlap.
The authors of [77] also take a closer look at possible overlap, examining OpenMPI on InfiniBand. In this paper, the distinction is not done regarding the kind of overlap, but regarding sender and receiver. Examining the use of `MPI_Isend` together with a blocking `MPI_Recv`, posted after the immediate send operation returned, it is shown that the necessary acknowledgment message will be received only in the matching `MPI_Wait` call on the sending side. Therefore, due to the used pipelined approach, only the first junk of the sent message can be overlapped. Vice versa, analyzing the behavior of `MPI_Irecv` by pairing it with a blocking `MPI_Send` (again posted after the `MPI_Irecv` has returned) shows that only in a polling based approach can a first message part be overlapped. No overlap is possible for direct RDMA approaches. Finally, paring `MPI_Isend` with `MPI_Irecv`, [77] shows that overlap is only possible for `MPI_Isend` using a direct RDMA approach and not for `MPI_Irecv`.

Extending previous work in [12] and [11], the authors of [13] aim to qualify the source of performance improvements when achieving independent progress and communication-computation overlap and offloading the relevant work steps to intelligent network interface cards. They conclude that independent progress is a significant contributor for performance improvements. More importantly, they show that for the tested benchmarks, the combination of adding independent progress, offload and overlap at the same time results in better performance improvements than the sum of improvements of implementing each part independently.
In the context of planning a large-scale computing system, the definition of network requirements for the target applications is important. For large-scale production scientific codes, the authors of [74] aim to quantify the potential

benefit of overlapping, showing that making use of communication independent work in order to hide communication allows for potentially significant relaxation of these requirements without decreasing application performance.
Being able to overlap communication with computation can also mitigate other shortcomings of compute systems. On some systems one CPU might not be able to saturate the inter-node bandwidth [69, 70]. For these cases it has been shown that overlap can result in best performance for applications which use the hybrid master-only programming model. In this model, only one master thread can take care of communication.
Also addressing hybrid programming approaches, namely `MPI-OpenMP`, the ability of MPI implementations to perform asynchronous point to point communication has been studied in [33]. OpenMP threads or tasks are being used for achieving overlap where it is not available automatically by dedicating one OpenMP thread manually to do the communication related work.

For MPI only approaches, the imbalances between ranks inside one compute node result in idle cycles in individual ranks. The authors of [25] make use of these cycles by implementing a collective polling approach in the thread based MPI implementation MPC [67]. The idea behind this approach is to use idle cycles in any MPI rank on one node to advance pending messages of any other rank on the same node.

Independent of which approach is being used (`MPI-Only` or `hybrid`), one approach to advance pending messages outside calls to the MPI library functions is using progress threads. Overlap can be achieved through multi-threading an MPI implementation using a dedicated communication thread, which can yield many cycles for computation [38]. Nevertheless, as mentioned above, other work has shown that this is not always the best approach. When using a progress thread, many MPI internal data structures have to be protected from concurrent access which is often done by using locking mechanisms. In order to remove this probably time consuming access control, the authors of [42] present a lock free asynchronous rendezvous protocol for progress thread based MPI implementations. At the same time, the authors of [35] present results indicating that it is difficult to achieve good performance improvements in cases where the progression thread has to share a core with computation.
In all cases where either a progress thread is being used or the application is itself multithreaded and wants to make concurrent calls to the MPI library functions, the used MPI implementation has to support the thread safety level `MPI_THREAD_MULTIPLE`, as defined in the MPI standard [55]. Since the performance of these implementations might differ from implementations written for `MPI_THREAD_SINGLE`, `MPI_THREAD_SERIALIZED` or `MPI_THREAD_FUNNELED`, the authors of [87] introduce a test suite to test these implementations. The presented results for different implementations on different systems from 2007 indicate that there was still quite a difference in performance between the different setups.

Different research approaches target specialized hardware. The authors of [65] enhance the MPICH2 MPI implementation to make use of Cray's Core Specialization (CoreSpec) feature along with hardware features of the XE Gemini Interface in order to be able to overlap communication with computation for micro-benchmarks and applications. In [40], the authors implement MPI-NP II, a network processor based message manager for MPI for Myrinet. Other work introduces an Infini-Band (IB) Management Queue in order to provide the means to overlap collective communications managed on the Host Channel Adapter (HCA) with computation on the host CPU [31]. The paper focuses on improvements for implementations of the `MPI_Barrier` collective operation. Other work analyzes the overlap potentials using a bitonic sorting algorithm for two kinds of explicit hardware: the laboratory prototype EM-X multithreaded multiprocessor and a commercially available IBM SP2 with wide nodes [80]. More compiler oriented, the authors of [45] describes compiler transformations for communication time overlap resulting from non-local memory accesses in shared memory environments.

Furthermore, the aim to overlap communication and computation is also a research target for other programming paradigms: The authors of [36] present a possible approach to exploit available overlap found at runtime in UPC programs. Overlapping in the hybrid MPI and SMPSs (SMP superscalar) approach is discussed by the authors of [51]. This paper also reports a reduced code complexity and less sensitivity to network bandwidth and OS system noise. Other results show advantages of one sided communication using Berkeley UPC together with GASNet in comparison to two sided MPI/Fortran implementations of the NAS FT benchmark [5].

4.2. Analyzing Asynchronous Communication Capabilities of MPI Implementations

As described in Section 4.1, many aspects of the used HPC environment play a role in providing real asynchronous communication. These include the availability of specialized hardware (e.g. programmable NICs), utilization of system resources, the implemented communication approach in the used MPI implementation, and more. Results from literature show that the availability of asynchronous communication, and especially the availability of data transfer overlap, is not guaranteed. Nevertheless it can be assumed that achieving overlap is possible. While the presented work reports results on the asynchronous capabilities of MPI implementations, no detailed information on how to best use manual progression in the context of overlap is provided.
In order to use knowledge about the behavior of MPI implementations in OpenMP

runtimes with the goal of improving the performance of hybrid MPI-OpenMP codes, a detailed analysis of this behavior is necessary. This analysis will be presented in this chapter, providing a benchmark which is highly adjustable in all aspects concerning parallel computing systems and asynchronous communication patterns.

4.2.1. Benchmark Goals

The central goal of this benchmark is to test different MPI implementations on different HPC systems in regard to asynchronous communication. In addition to acquiring the knowledge about whether or not synchronization and data transfer overlap is possible, this benchmark also aims to analyze the communication details. It is necessary to get detailed measurements of the different communication related events (e.g. MPI library calls to functions as `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`, etc.). Together with timing information about the overall benchmark execution times, this can be used to 1) quantify possible overlap, 2) show which steps are necessary to achieve overlap and, in cases where no overlap is provided automatically, 3) provide information on how to best use progression functions to do so manually.
For MPI implementations not providing overlap automatically, but providing overlap through manual progression, the additional steps can introduce additional overhead, which must be compared to the overall performance improvements gained through reducing the times spent in the communication functions.

Since the results are to be used in a hybrid MPI-OpenMP environment, it is not the goal to occupy system resources through additional progress threads. Progress threads are not provided by many MPI implementations and none of those that do could be found on the HPC systems used for this thesis. One of the MPI implementations which experimented with progress threads is OpenMPI. Nevertheless, these progress threads have never been stable and have been removed with OpenMPI 1.6 [28]. Furthermore, as mentioned before, the use of progress threads has been reported to not always be efficient, especially in cases where the progress thread has to share a core with the actual application [35]. How to use progress threads or communication offload, in cases where it will become available in the future, together with the approaches presented in Sections 4.3 and 4.4, will be discussed later.

4.2.2. Design Goals

In order to analyze the behavior of MPI implementations in regard to asynchronous communication, the benchmark iteratively tries to overlap a definable set of communications between different MPI ranks with communication independent work. In order to quantify the results, the benchmark measurements need to be compared to a base version whose behavior is well understood and

stable. For this base version, a synchronous version of the benchmark is used, where communication and computation phases are separated and strictly sequential.
In order to obtain useful, reproducible timing results, the benchmark is set up with well defined and adjustable benchmark parameters. A very homogeneous CPU usage is used as computation with which the communication is to be overlapped. This is done so that no random effects can influence the computation times, making sure that timing differences are due to the communication part of the benchmark.
Due to the fact that for most parallel applications (including the reference application presented in Chapter 5) the used MPI ranks communicate in pairs, this analysis also bases the communication patterns upon bidirectional communication partners. I.e. if MPI rank `X` sends a message to rank `Y`, `X` also receives a message from `Y`.
In regard to manual progression, literature discussed in the previous Section 4.1 reports that this is usually done through calls to the MPI library functions. As usually no MPI functions are called between the initializing `MPI_Isend` respectively `MPI_Irecv` and the matching `MPI_Wait` function, it has to be decided which function to use for manual progression. One function in particular has to become involved in the communication advancement eventually: `MPI_Test`. This is a consequence of the fact that any immediate send or receive does not have to be finished using a matching `MPI_Wait` call, but can also be confirmed by a successful `MPI_Test` function. Therefore any `MPI_Wait` can be replaced by a loop calling `MPI_Test` until successful communication completion is confirmed. Algorithm 4.1 and Algorithm 4.2 are equivalent.

Algorithm 4.1 Asynchronous send matched by `MPI_Wait`

```
1: MPI_Isend(A)
2: Compute(X)
3: MPI_Wait(a)
```

Algorithm 4.2 Asynchronous send matched by `MPI_Test`

```
1: Finished = FALSE
2: MPI_Isend(A)
3: Compute(X)
4: while !Finished do
5:     Finished = MPI_Test(A)
6: end while
```

4.2.3. Description of Benchmark

The first parameter to be selected when running the benchmark is the number of MPI ranks used and their placement on the available compute nodes. As the asynchronous communication functions to be tested are part of the two-sided-communication functions, the chosen number of ranks has to be even. In order to test inter-node communication, these ranks must be evenly distributed across an even amount of compute nodes. The simplest setup would be two ranks, placed on one node each. Using four ranks, different placement options are possible: Placing each rank on an individual node, providing each rank with its own network interface, or placing two ranks on two nodes each. As mentioned earlier, the communication for this analysis is always bidirectional. If rank `0` sends to rank `1`, rank `1` also sends to rank `0`. In order to actually measure the behavior of asynchronous communication through the available network, the communication partners are always chosen pairwise between the two sets of ranks, resulting in no intra-node communication.
Since many HPC applications require more than one communication partner per rank, but can require each rank to exchange data with a subset of all remaining ranks, the second parameter available is the number of communication partners each rank has to communicate with. Again making sure that the communication partners are in the two different halves of the used ranks, each rank can have a definable amount of communication partners. For multiple neighbors, each rank chooses the neighbors starting with its corresponding partner in the other half of the ranks in a round robin fashion. The benchmark can be configured so that each rank sends a user-defined number of messages to each communication partner, which must be greater or equal to one. These messages can be defined by providing a desired message size. Using the standard setup, the messages will be created as sequential memory regions containing enough doubles to make up for the message size. As each message buffer can only be used in one MPI communication at the same time, each rank creates individual buffers for all defined messages and initializes them with random double values. These values will not change throughout the benchmark run.
Due to the nature of parallel applications, MPI offers the option to create user defined derived datatypes. These MPI datatypes define data regions which can be passed to communication functions as send or receive buffers and must not be sequential in memory. For details on MPI datatypes refer to Section 2.2. In order to see how well asynchronous communication works in combination with derived datatypes, the benchmark also offers to create the message buffers using strided datatypes created by using `MPI_Type_vector` (see MPI standard [55] for details). The blocklength and stride size for the vectorized datatype can be passed to the benchmark as parameter.
Independent of the message buffer type, all messages in one benchmark run have the same size and memory layout.

In order to overlap communication with computation, a communication independent computation phase is included in the benchmark whose length is user definable. The computation can easily be replaced by any custom code and consists of a stencil update scheme on a matrix. For a definable matrix size, each rank initializes this matrix with random double values and, in the computation phase, updates each element of the matrix as the average of its four neighbors. The computation is chosen to occupy the CPUs hosting the ranks, which could also be done through other operations such as matrix-matrix multiplications or by using integer operations instead of the floating point units.

Algorithm 4.3 Communication pattern for the **synchronous benchmark**

```
 1: numP = "Number of communication partners"
 2: numM = "Number of messages per communication partner"
 3: iterations = "Number of iterations to be executed"
 4: timeC = "Defined compute time"
 5: for $iter \in 1 \ldots iterations$ do
 6:     for all $i \in 1 \ldots numP$ do
 7:         for all $j \in 1 \ldots numM$ do
 8:             MPI_Irecv(i,j,$msg_{(j,i)}$);
 9:             MPI_Isend(i,j,$msg_{(i,j)}$);
10:         end for
11:     end for
12:     MPI_Waitall($msgs_{i,j} | \forall i \in \{1 \ldots numP\}, \forall j \in \{1 \ldots numM\}$)
13:     Compute(timeC)
14: end for
```

Algorithm 4.4 Communication pattern for the **basic asynchronous benchmark**

```
 1: numP = "Number of communication partners"
 2: numM = "Number of messages per communication partner"
 3: iterations = "Number of iterations to be executed"
 4: timeC = "Defined compute time"
 5: for $iter \in 1 \ldots iterations$ do
 6:     for all $i \in 1 \ldots numP$ do
 7:         for all $j \in 1 \ldots numM$ do
 8:             MPI_Irecv(i,j,$msg_{(j,i)}$);
 9:             MPI_Isend(i,j,$msg_{(i,j)}$);
10:         end for
11:     end for
12:     Compute(timeC)
13:     MPI_Waitall($msgs_{i,j} | \forall i \in \{1 \ldots numP\}, \forall j \in \{1 \ldots numM\}$)
14: end for
```

Algorithm 4.5 Communication pattern for the **advanced asynchronous benchmark** with manual progression

```
1: numP = “Number of communication partners”
2: numM = “Number of messages per communication partner”
3: iterations = “Number of iterations to be executed”
4: timeC = “Defined compute time”
5: numT = “Number of calls to MPI_Test”
6: tbt = timeC / (numT + 1)                    ▷ Time between MPI_Test calls
7: for $iter \in 1 \ldots iterations$ do
8:     for all $i \in 1 \ldots numP$ do
9:         for all $j \in 1 \ldots numM$ do
10:            MPI_Irecv(i,j,$msg_{(j,i)}$);
11:            MPI_Isend(i,j,$msg_{(i,j)}$);
12:        end for
13:    end for
14:    for $t \in 1 \ldots numT$ do
15:        Compute(tbt)
16:        MPI_Testall($msgs_{i,j} | \forall i \in \{1 \ldots numP\}, \forall j \in \{1 \ldots numM\}$)
17:    end for
18:    Compute(tbt)
19:    MPI_Waitall($msgs_{i,j} | \forall i \in \{1 \ldots numP\}, \forall j \in \{1 \ldots numM\}$)
20: end for
```

The benchmark takes care to avoid communication setup effects by exchanging all defined messages once before the actual execution of the chosen communication pattern. The same is done for the computation, which is repeated multiple times in order to calibrate the timing mechanism taking care of the computation phase duration. The latter can also be provided through the benchmark parameters.
For the actual benchmark execution, three different communication patterns can be chosen, which will be repeated for a user-defined number of iterations. These patterns are:

- Synchronous version (SYNC)
- Basic asynchronous version (ASYNC)
- Advanced asynchronous version with manual progression (ASYNC(X))

The **synchronous version** is the base version: Communication happens before the defined computation phase. This includes everything from initialization to successful termination of all defined messages. Communication is initialized using the immediate versions of send and receive and finished by a directly following call to `MPI_Waitall`. This is done because each rank needs to send and receive to each neighbor for each message. Using these functions avoids deadlocks and allows for optimized network usage. An outline of the communication

pattern can be seen in Algorithm 4.3. This version will be referred to as "SYNC" in the following sections and represented as "S" in the time measurement graphs.
The **basic asynchronous version** tests whether or not the tested MPI implementation provides overlap automatically. Communication is initialized before and finished after the defined computation phase. Each rank calls `MPI_Irecv` and `MPI_Isend` for each communication partner and each message per communication partner. This is followed by the defined computation phase, which again is followed by a call to `MPI_Waitall`. In this case the used system can take care of the actual message transfer (synchronization and data transfer overlap) during the computation phase. An outline of the communication pattern can be seen in Algorithm 4.4. This version will be referred to as "ASYNC" in the following sections and represented as "A" in the time measurement graphs.
The final communication pattern is the **advanced asynchronous version with manual progression**. As in the basic asynchronous version, the communication is initialized for each message in the beginning of an iteration. During the computation, at even time intervals depending on the amount of calls to be made, the manual message progression function (`MPI_Test`, `MPI_Testall` or `MPI_Testany`) is called. The sum of computation time equals the time spent in computation in the other versions, adding these MPI library calls as overhead. After the computation/advancement phase, `MPI_Waitall` is again used to make sure that all messages have been delivered successfully and all buffers are free to be used. The outline of this communication pattern can be seen in Algorithm 4.5. This version will be referred to as "ASYNC(X)" in the following sections, where "X" is the number of chosen `MPI_Test` calls. In the time measurement graphs, this version is represented by the corresponding "X"-axis labeling.

In order to understand and analyze the behavior of the different benchmark executions, detailed time measurements are carried out for each run. The times presented in this work are always averages of multiple repetitions of the same parameter sets. This ensures that the results show the normally expected behavior and avoid unusual external influences. The timing measurements done are:

- Total benchmark execution time: The difference between two time-stamps taken before the first and after the last iteration, respectively.

- Communication initialization time: The sum of time spent in all `MPI_Isend` and `MPI_Irecv` calls throughout all iterations.

- Communication termination time: The sum of all `MPI_Waitall` calls throughout all iterations.

- Communication advancement time: The sum of all `MPI_Testall` calls throughout all iterations.

- Computation time: The total time spent in the computation function throughout all iterations. This measurement can be used to see if system resources are used during the computation phase which are not part of the application processes but delay the process nevertheless.

With these measurements, it is possible to compute the overall communication time (i.e. time spent in communication related MPI functions) by adding up all communication related times.
In order to understand the final benchmark timing results, the possible scenarios and the expected timing patterns will be discussed in the following paragraphs.

4.2.4. Benchmark Results: Expectations

For the different communication patterns, different behavior concerning the resulting times is possible. Depending on the capabilities of the tested MPI implementation to achieve overlap, different results can be expected for the taken measurements. Using two ranks exchanging one message in each direction for two iterations, these expectations will be discussed here. The presented sketches represent the timeline of events happening in each of the two ranks, where time is increasing along the x-axis. The duration of the individual events shown here is not to scale and should represent proportions only.

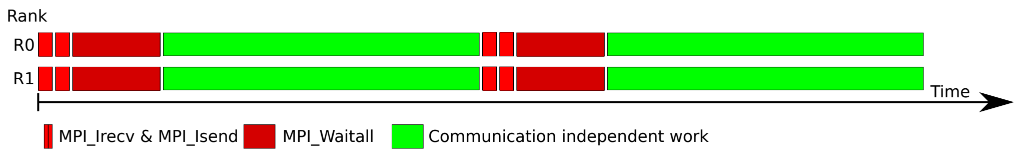


Figure 4.1.: Benchmark Expectations: Behavior of the *synchronous* communication pattern (2 iterations).

The **SYNC** communication pattern is the only one in which the behavior is straight forward and timings can be predicted quite precisely. As the `MPI_Irecv` is always posted before the `MPI_Isend` and these two functions are supposed to return "immediately," the two calls will be relatively short. They should only include the steps necessary to make the communications known in the system and tell the MPI environment that the corresponding buffers are ready to be used. These two calls are followed by the `MPI_Waitall` calls in both ranks, which return only after both messages have been exchanged successfully. The time from initialization until the end of `MPI_Waitall` includes the actual data transfer, as it is guaranteed that at the time `MPI_Waitall` returns both the send and the receive buffers on the calling rank can be reused and contain the expected data (in the receive buffers). Only in cases where the MPI implementation takes care of the communication inside the communication initialization functions could the timing be different. This is unlikely as the compiler and runtime must guarantee that deadlocks are not introduced by blocking here.

Next is the computation phase, which computes the predefined number of stencil operations on the computation phase matrix. This pattern, depicted here for the two iterations, can be seen in Figure 4.1.
As no other requirements are placed on the nodes used for the benchmark (and optimally on the network as well), the total communication time represents the best time possible to transfer the data. The total measured time is expected to equal the sum of all communication times and the computation times.

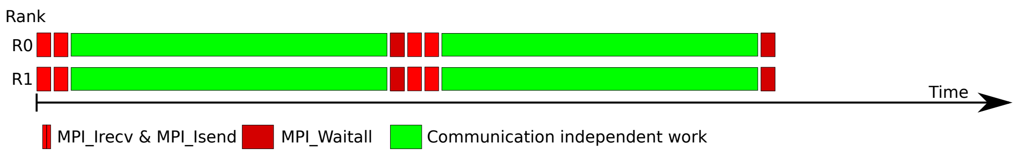


Figure 4.2.: Benchmark Expectations: Behavior of the *asynchronous* communication pattern in cases where overlap **is** possible (2 iterations).

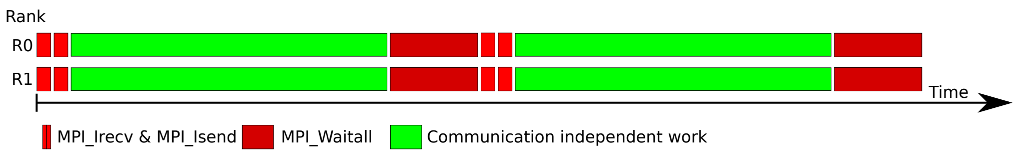


Figure 4.3.: Benchmark Expectations: Behavior of the *asynchronous* communication pattern in cases where overlap is **not** possible (2 iterations). Communication is happening in the `MPI_Waitall` call.

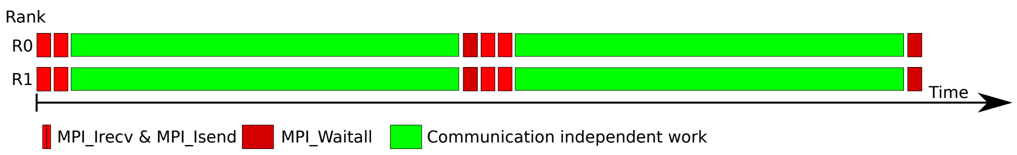


Figure 4.4.: Benchmark Expectations: Behavior of the *asynchronous* communication pattern in cases where overlap is **not** possible, but the time spent in communication functions is short (2 iterations). Communication is happening outside the MPI library calls but occupies the system resources during the computation phase, delaying it.

The possible expectations for the other two communication patterns is more complex, and the presented expectations have been created based on the theoretical possible behavior and the results presented in literature discussed in Section 4.1.
For the **ASYNC** communication pattern, different scenarios are possible. As discussed in the benchmark description above, the `MPI_Waitall` call is moved

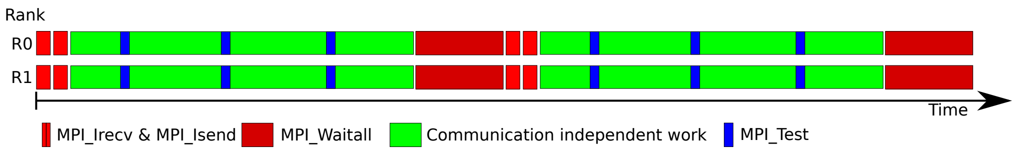


Figure 4.5.: Benchmark Expectations: Behavior of the *asynchronous* communication pattern calling `MPI_Test` regularly in cases where overlap is **not** possible and `MPI_Test` does **not** advance pending messages (2 iterations). The time spent in `MPI_Test` is pure overhead, not contributing to the communication at all. In this example three `MPI_Test` calls are done per iteration.

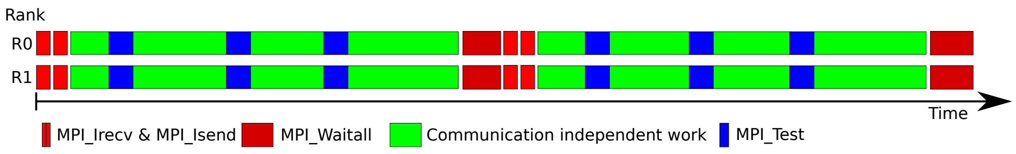


Figure 4.6.: Benchmark Expectations: Behavior of the *asynchronous* communication pattern calling `MPI_Test` regularly in cases where overlap is **not** possible, but `MPI_Test` **does** advance pending messages (2 iterations). The time spent advancing messages in `MPI_Test` is not needed in the `MPI_Waitall` calls, which therefore return faster. In this example three `MPI_Test` calls are done per iteration.

behind the corresponding computation phase, allowing for the system to take care of communication any time between communication initialization and the end of `MPI_Waitall`. This allows for three possible scenarios:
Using MPI nonblocking communication this way results in overlap (Figure 4.2). In this case actual synchronization and data transfer has been overlapped with computation and `MPI_Waitall` can return directly, reporting the communication finished.
The second possibility is that no overlap is achieved and the total communication time is the same as in the SYNC version (Figure 4.3). Here data transfer is also done during the `MPI_Waitall` function, effectively moving the communication from before (in the SYNC version) to after computation.
Finally, it is possible that the communication times reported are short, as in the case where overlap is possible, but showing the same total benchmark execution time as the SYNC version (Figure 4.4). In this case the actual data transfer happens outside the MPI library calls and instead requires CPU resources during the computation phase, which in turn takes longer than expected. While the latter scenario is possible, results from related work suggest that it is the most unlikely of the three possibilities.

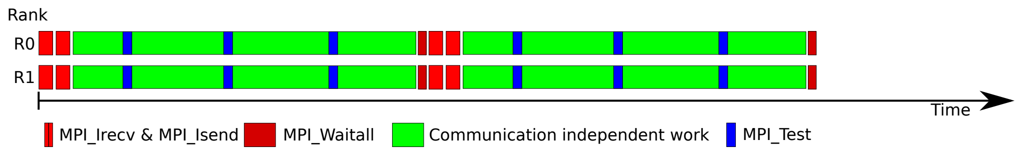


Figure 4.7.: Benchmark Expectations: Behavior of the *asynchronous* communication pattern calling `MPI_Test` regularly in cases where overlap **is** possible through the fact that `MPI_Test` initialized the advancement of pending messages (2 iterations). The time for `MPI_Test` calls is only spent for initializing the communication and the actual data transfer can be done while computation continues. In this example three `MPI_Test` calls are done per iteration.

In the **ASYNC(X)** communication pattern, the computation phase is interleaved regularly with `MPI_Test` calls. For the example scenario, three `MPI_Test` calls are chosen ($\Rightarrow$ ASYNC(3)). In this case three different scenarios are possible:

- `MPI_Test` does nothing. Neither does it advance messages nor does it result in overlap. (Figure 4.5)
- `MPI_Test` advances messages but does so before returning, resulting in no overlap as well. (Figure 4.6)
- `MPI_Test` advances messages and results in overlap. (Figure 4.7)

In case of no overlap and no message progression, the times spent in `MPI_Irecv`, `MPI_Isend` and `MPI_Waitall`, as well as the computation time, are the same as in the ASYNC communication pattern. The `MPI_Test` calls can be considered to be pure overhead, not contributing to communication at all. Besides this overhead, the interruption of the computation might also introduce overhead, which is not considered in Figure 4.5. This scenario seems unlikely. As discussed in previous chapters, an MPI communication initialized with an immediate send or receive function does not require a matching `MPI_Wait` in cases where successful communication is confirmed through `MPI_Test`. Nevertheless is might be the case that a large number of `MPI_Test` calls is necessary to achieve this, resulting in the presented expectation scenario in cases where too few `MPI_Test` calls are being used.
Another possibility is that `MPI_Test` does advance the pending messages partly in each call without providing overlap. In this case the time saved in `MPI_Waitall` is spent inside the `MPI_Test` function, again moving the communication to a different step in the process in comparison to the SYNC and ASYNC versions without reducing the overall execution time of the total number of iterations. Depending on which portion of the messages can be progressed in `MPI_Test`, `MPI_Waitall` is needed to finish the rest of the pending messages or encounters

successful completion of the same (see Figure 4.6).
Finally it is possible that using `MPI_Test` calls does result in overlap and that using `MPI_Test` calls in between the communication initialization and termination does trigger synchronization and data transfer overlap (see Figure 4.7). Two different behaviors are possible internally. One option is that only the synchronization steps are needed, which are taken care of after a certain amount of `MPI_Test` calls. Automatic data transfer overlap is possible afterwards. Another option is that a call to `MPI_Test` can only initialize the data transfer overlap for a part of the message, resulting in the need for multiple `MPI_Test` calls to trigger the transfer of all other parts frequently. In both cases `MPI_Test` calls are short since they do not need to take care of the actual data transfer. The total benchmark execution time is now reduced to the time needed to do computation and the times spend initializing and triggering message progression.

Figure 4.8 depicts a possible timing graph of an imaginary set of benchmark runs for the different expectation scenarios. For a given computation time and a given message size, the benchmark will be executed (multiple times) for all scenarios: SYNC, ASYNC and ASYNC(X) for x=1..15 (in this example). It is assumed that the communication time is smaller than the computation time. The four different possible scenarios depicted are:

1. No Overlap: For no version overlap is possible. The times for SYNC and ASYNC are the same. Adding `MPI_Test` calls is at least as long and might introduce additional overlap.

2. Overlap for all ASYNC and ASYNC(X) versions: Using `MPI_Irecv`and `MPI_Isend` together with `MPI_Waitall` is sufficient for overlap. The time for ASYNC is nearly as short as the computation time. Only the initialization of the messages and the check of successful completion is needed in the MPI library calls. Adding `MPI_Test` calls is unnecessary and might cause overhead.

3. Overlap only when using `MPI_Test` in order to take care of synchronization with automatic overlap afterwards. In the presented example 2 `MPI_Test` calls in each rank would be necessary to achieve synchronization.

4. Overlap only when using `MPI_Test` in order to advance messages block by block, being able to overlap each message part with computation. In cases where not enough `MPI_Test` calls are used, the rest of communication has to be done in `MPI_Waitall`.

4.2.5. Benchmark Results: Measurements and Analysis

With these expectations in mind an extensive set of tests has been executed on different HPC systems. Details about the system descriptions can be found in Appendix A. The following combinations have been used: Intel MPI 4.0, Intel

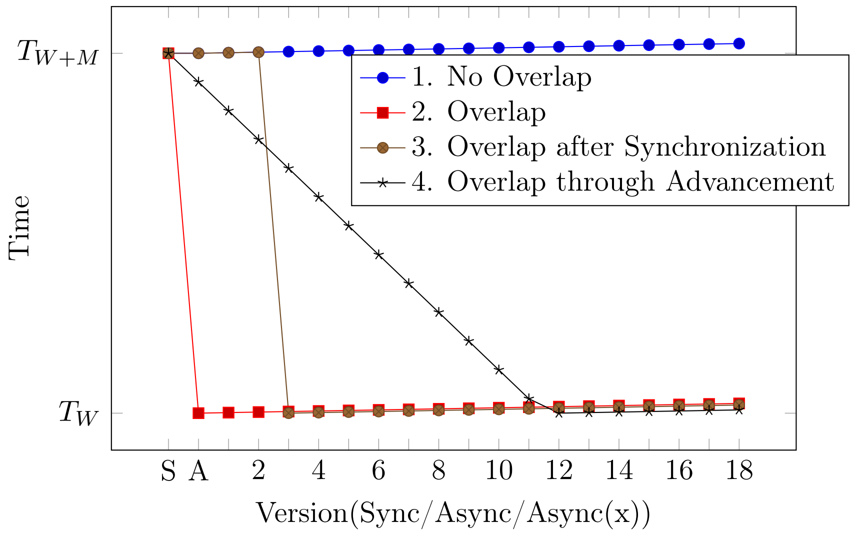


Figure 4.8.: Benchmark Expectations: Example timelines of different possible outcomes for a specified benchmark setup in which T_W is the time needed for computation (work) and T_M is the time needed for communication (message). 1.) No overlap. Neither using asynchronous communication nor adding `MPI_Test` calls results in any overlap. 2.) Overlap as soon as asynchronous communication is used "around" computation. `MPI_Test` calls are not necessary but might result in overhead. 3.) Overlap is possible after a successful synchronization of the communication partners. In this case after using three `MPI_Test` calls (as an example). Any more `MPI_Test` calls are unnecessary and can result in overhead. 4.) Overlap is possible, but data transfer overlap has to be achieved through initializing the asynchronous sending of message junks regularly through `MPI_Test` calls. In cases too few `MPI_Test` calls are done, only part of the messages is overlapped. Too many calls after successful communication completion result in overhead.

MPI 4.1 and MPT MPI on the ICE cluster as well as IBM MPI on the SuperMUC.
As the basic behavior of the different combinations in regard to asynchronous communication should be observable using two ranks on two compute nodes, i.e. doing real inter-node communication, this scenario has been looked at in detail. Also different numbers of messages per iteration and computation partner have been chosen.

For the presented combinations, the used parameters for the benchmark covered the following ranges:

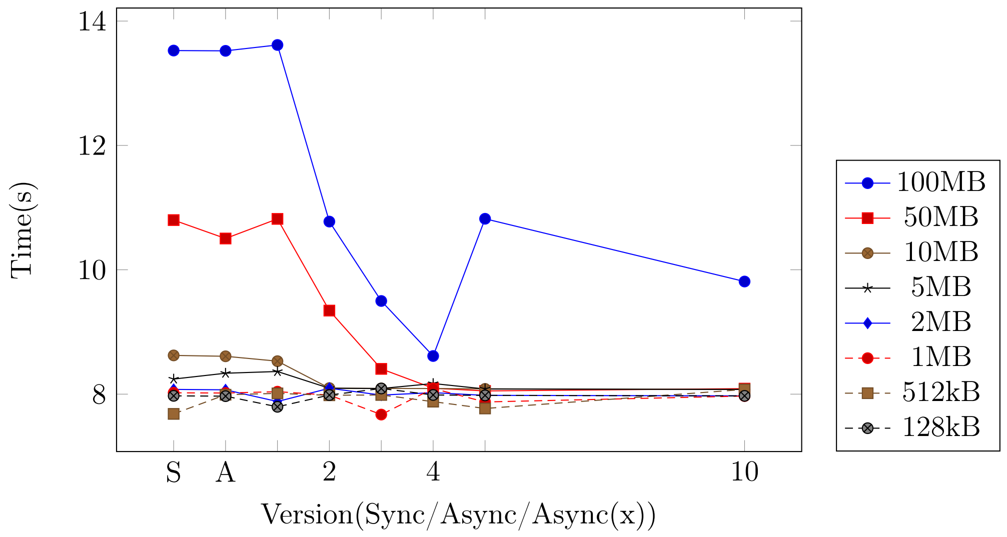


Figure 4.9.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using Intel MPI 4.0. 1 message sent per iteration and rank using a sequential memory buffer.

- Versions: SYNC, ASYNC, ASYNC(X)
- Number of `MPI_Test` calls (= X in ASYNC(X)): 1 to at least 10.
- Iterations: 100
- Message sizes: 128kB up to 200MB
- Communication time per iteration: 80ms

The number of 100 iterations for each test has been chosen after confirming that different iteration counts do not have visible impact on the results. The chosen iteration duration of 80ms covers the communication time for most chosen message sizes on the chosen systems.
Comparing these tests with extra large messages, which sequentially take longer than 80ms, it can be seen that the maximum overlap achievable is, as expected, $max(sequential\ computation\ time, sequential\ communication\ time)$. All tests have been executed multiple times and the resulting timing results have been averaged.
The times presented in the result graphs are for all 100 iterations, i.e. including the time for 100 computation phases, the time for sending the defined number of messages 100 times, etc.

The results showing the overall benchmark execution times for the different setups are similar in all cluster-MPI combinations, even when choosing a different number of messages per communication partner and iteration:

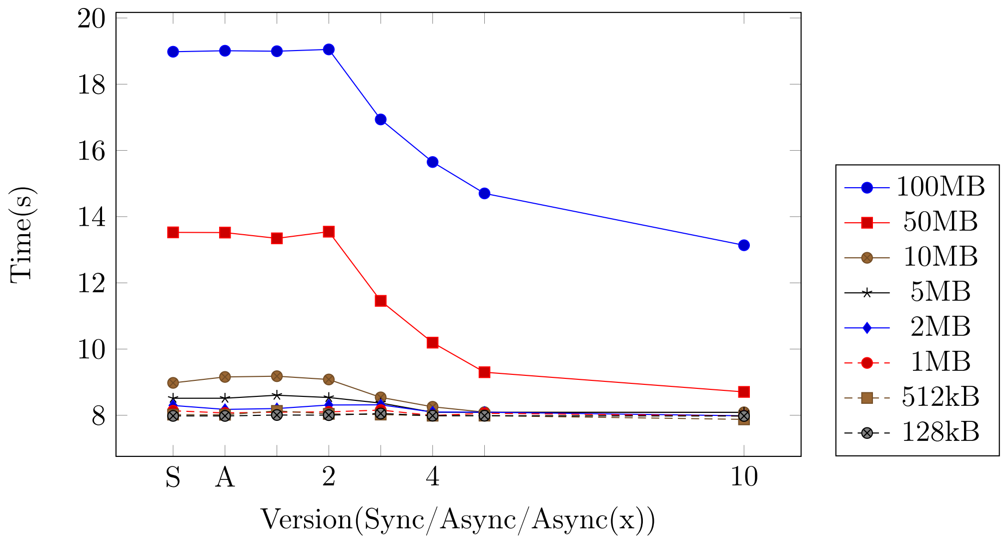


Figure 4.10.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using Intel MPI 4.0. 2 messages sent per iteration and rank using sequential memory buffers.

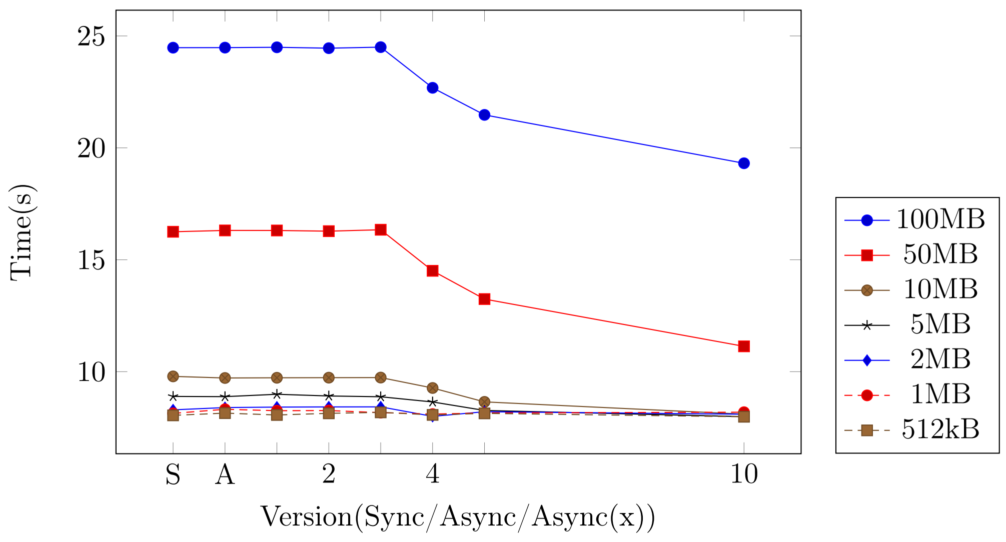


Figure 4.11.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using Intel MPI 4.0. 3 messages sent per iteration and rank using sequential memory buffers.

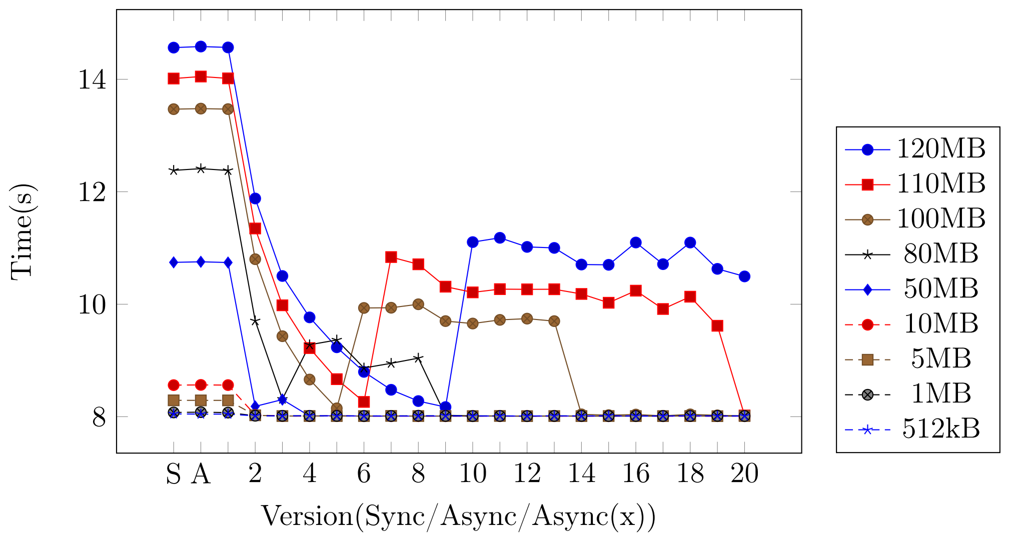


Figure 4.12.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using Intel MPI 4.1. 1 message sent per iteration and rank using a sequential memory buffer. Processes pinned to core number 4 of the respective nodes.

- ICE - Intel MPI 4.0:
  - 1 message: Figure 4.9
  - 2 messages: Figure 4.10
  - 3 messages: Figure 4.11
- ICE - Intel MPI 4.1: Figure 4.12
- ICE - MPT MPI:
  - 1 message: Figure 4.13
  - 2 messages: Figure 4.14
  - 3 messages: Figure 4.15
- SuperMUC - IBM MPI:
  - 1 message: Figure 4.16
  - 2 message: Figure 4.17

Independent of the presented combination of HPC system with MPI implementation as listed above, the SYNC and ASYNC versions show nearly identical overall benchmark execution times. Therefore it can be concluded that in no setup can real overlap be achieved by the exclusive use of immediate send and receive calls combined with a matching wait function surrounding the communication independent work. In all results the use of `MPI_Test` provides the desired

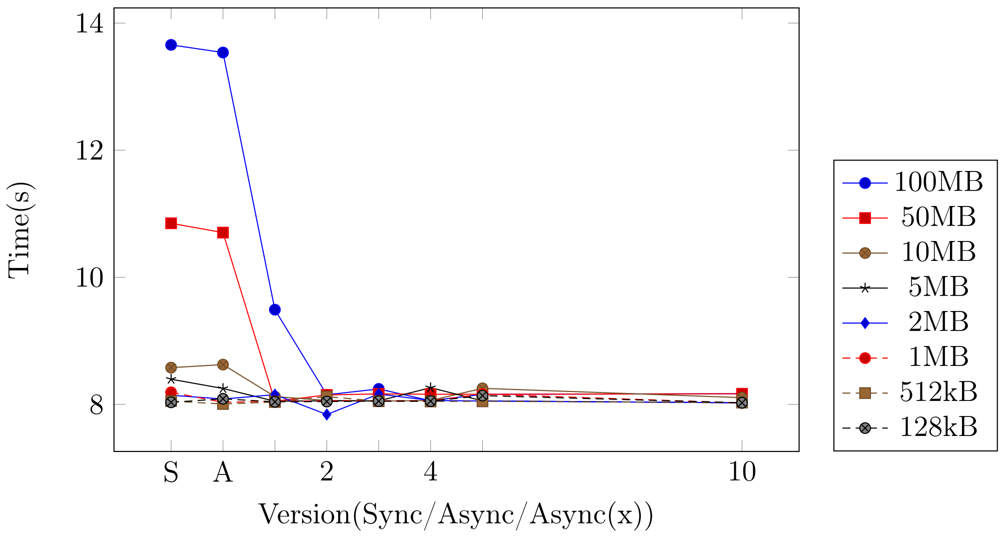


Figure 4.13.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using MPT MPI. 1 message sent per iteration and rank using a sequential memory buffer.

overlap. Nevertheless the necessary amount of `MPI_Test` calls depends on the system and the overall amount of data sent in each iteration. For IntelMPI on the ICE cluster, at least two `MPI_Test` calls are necessary in order so observe any kind of overlap. The other combinations show overlap starting with the first used `MPI_Test` call.
The amount of communication time overlapped depends on the overall amount of data transferred in each iteration. For amounts of data below a certain limit, the difference between no overlap and total overlap is a single additional `MPI_Test` call. For amounts of data above this limit, multiple additional `MPI_Test` calls increasingly reduce the overall execution time. Looking at the chosen message sizes, this limit lies between 10MB and 50MB for IntelMPI on the ICE cluster, between 50MB and 100MB for MPT-MPI on the ICE cluster and between 100MB and 200MB for IBM MPI on the SuperMUC. This limit is not dependent on the system, as can be seen by looking at the results for the different MPI implementations used on the ICE cluster. It is also not dependent on the number of messages used to transfer this amount of data. Looking at the multiple messages per iteration used for MPT-MPI on ICE and IBM-MPI on SuperMUC, it can be seen that the limit is for the sum of message sizes per iteration. E.g. in case of IBM-MPI on SuperMUC, when using 1 message per iteration (Figure 4.16), the limit lies between the message sizes 100MB and 200MB. Using 2 messages in each iteration (Figure 4.17), it lies between 50MB and 100MB per message.
For messages above the limit, multiple `MPI_Test` calls are needed in order to observe total overlap. In cases where the synchronous communication time is

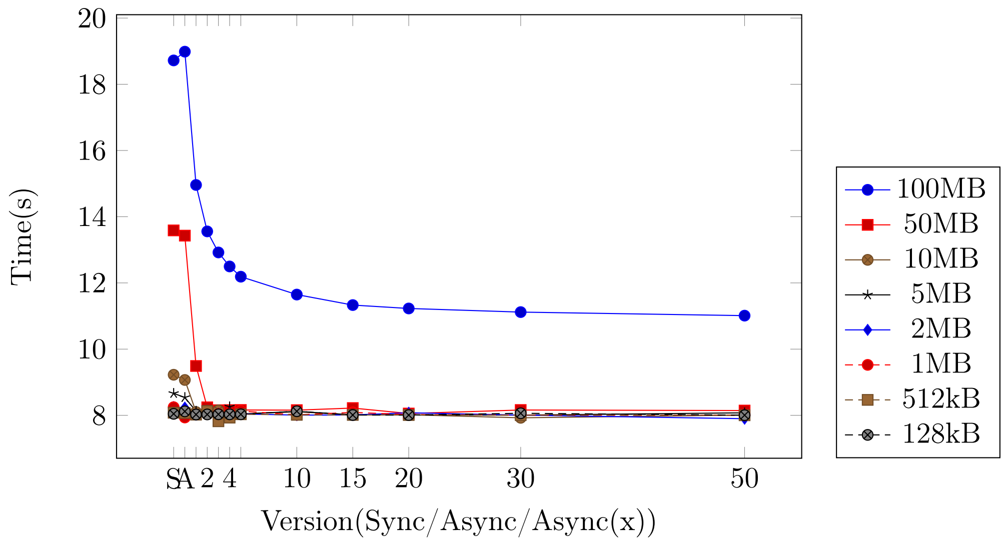


Figure 4.14.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using MPT MPI. 2 messages sent per iteration and rank using sequential memory buffers.

smaller than the computation time, increasing the `MPI_Test` count reduces the overall execution time until it is equal to the computation time. In cases where the synchronous communication time is longer than the computation time, communication time is dominant and the overall execution time can be reduced to match it.
This behavior can be observed on other HPC systems as well. This can be seen in the results of Chapter 5, where the knowledge obtained through this benchmark has been applied in a hybrid MPI-OpenMP approach, showing that MPI awareness in the OpenMP runtime can indeed efficiently create synchronization and data transfer overlap in real applications.

These results indicate that a mixture of the presented scenarios is true: At least some `MPI_Test` calls are necessary for synchronization and then each `MPI_Test` call is capable of initializing the asynchronous sending/receiving of the next part of the message. For IntelMPI on the ICE cluster (Figure 4.9, Figure 4.12), at least two `MPI_Test` calls are necessary, which would suggest that some synchronization is necessary. For the other combinations, this synchronization can be done between the communication initialization and the first `MPI_Test` call, which can directly be used to start the overlap. In case the amount of data to be transferred is below the described limit, data transfer overlap can be achieved by one additional `MPI_Test` call. In cases where additional CPU involvement is necessary for additional data transfer overlap, the increasing overlap through additional `MPI_Test` calls suggests that message progression is achieved.

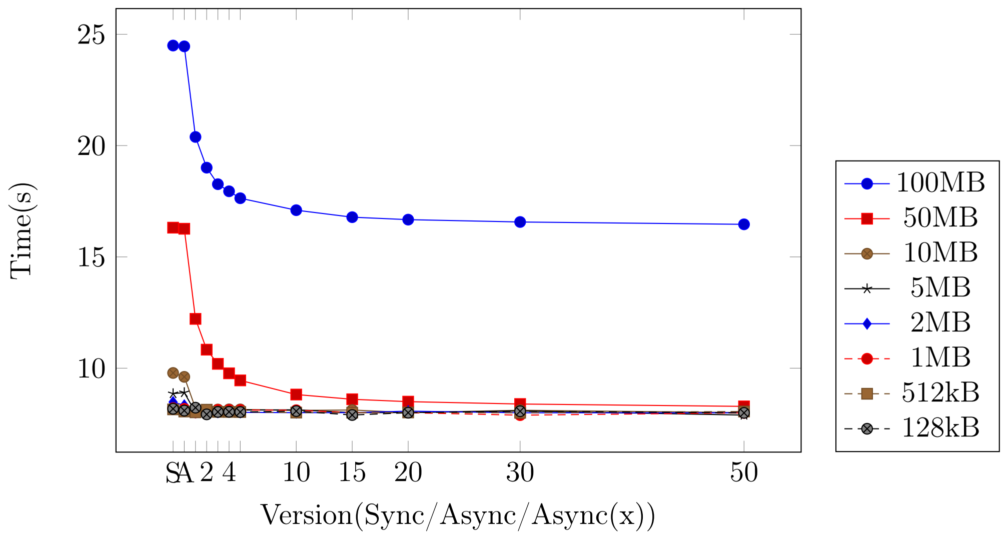


Figure 4.15.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using MPT MPI. 3 messages sent per iteration and rank using sequential memory buffers.

In order to understand the behavior of the MPI implementations in more detail, the timing results for the individual events (e.g. computation, MPI initialization, `MPI_Test` and `MPI_Waitall`) can be used. As the presented results of this analysis are the same for all shown combinations, it will be done using the combination IBM-MPI on SuperMUC as an example. The corresponding graphs for a second combination (IntelMPI 4.0 on ICE using 2 messages per iteration Figure 4.10) can be seen in Appendix B.1.

For the results of the chosen example presented in Figure 4.16, two nodes of the SuperMUC hosted one MPI rank each. The ranks have been pinned to use core 8 of each node in order to minimize NUMA effects. The reason for choosing core 8 instead of the system default pinning strategy, which pins the first to be pinned process on core 0, will be discussed later.
The parameters for the benchmark here were:

- 100 iterations
- 1 communication partner per rank
- 1 message per iteration and communication partner
- 80ms of work per iteration
- Message sizes: 1MB, 10MB, 50MB, 100MB, 200MB

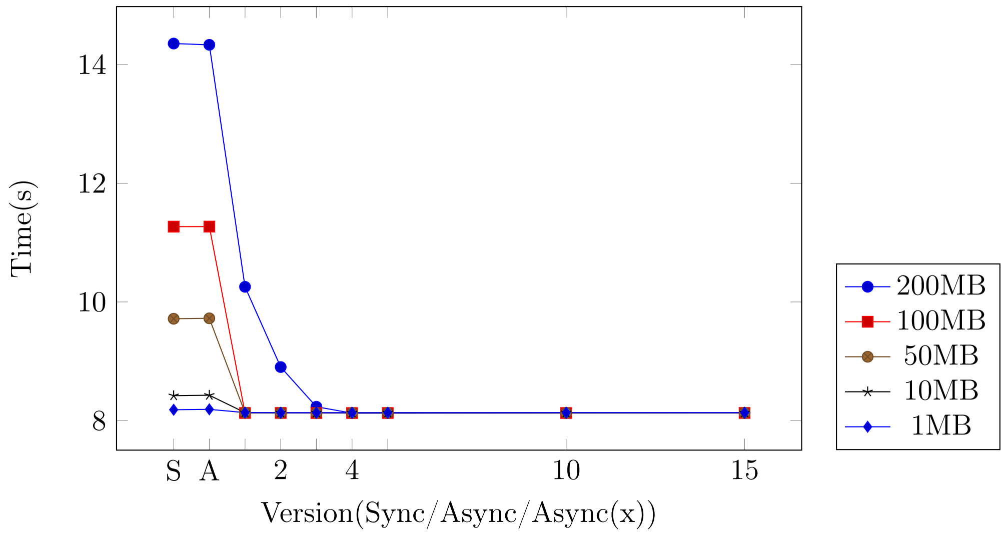


Figure 4.16.: Benchmark Results: 1 `MPI` rank each on two nodes of the SuperMUC using Intel IBM MPI. 1 message sent per iteration and rank using a sequential memory buffer. Processes pinned to core number 8 of the respective nodes.

Each parameter combination has been executed using the SYNC, ASYNC and ASYNC(X) version. The amounts of `MPI_Test` calls (X) chosen are 1, 2, 3, 4, 5, 10 and 15. All combinations have been repeated five times and the maximum relative standard deviation for any combination is below 0.223%. The times shown in Figure 4.16 are the average times of these five runs, including all 100 iterations.
As the expected computation time is 8 seconds for 100 iterations, any additional time needed can be considered overhead. That nearly all the overhead can be considered communication overhead can be seen by comparing Figure 4.16 and Figure 4.18. The latter shows the sum of time spent in MPI related functions for all 100 iterations. The times presented here are nearly exactly the same as the overhead observed in the original overall results. Looking at the synchronous version (SYNC, labeled S in the graphs), the time needed for communication in cases where all resources are dedicated to doing so can be seen.

Looking at the expectations discussed earlier, the fact that the communication time decreases in the same way as the measured overall benchmark execution time shows that real data transfer overlap is being achieved through the use of `MPI_Test` calls. While the communication initialization using `MPI_Irecv` and `MPI_Isend` can be neglected (Figure 4.19), it can be seen that the actual communication which can not be overlapped happens during the `MPI_Waitall` call (Figure 4.21). In case of the standard asynchronous version (ASYNC, labeled A in the graphs), this includes the entire data transfer.

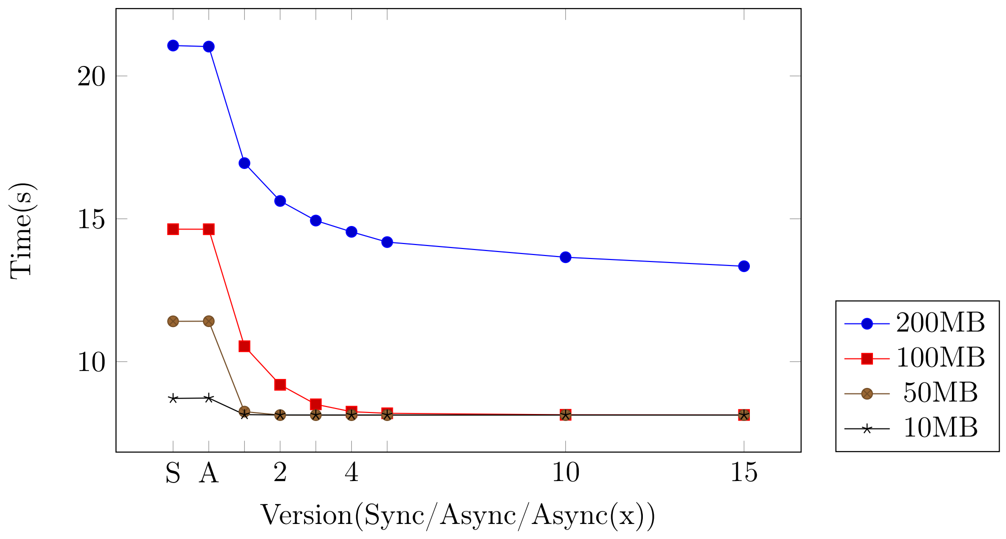


Figure 4.17.: Benchmark Results: 1 `MPI` rank each on two nodes of the SuperMUC using IBM MPI. 2 messages sent per iteration and rank using sequential memory buffer. Processes pinned to core number 8 of the respective nodes.

Looking back at the expectations presented in Figure 4.8, the communication timer measurements suggest that depending on the message size and the MPI implementation - HPC system combination, overlap is being achieved after a successful synchronization for small messages and, through message advancement, for large messages. Also, `MPI_Test` is used to initialize message progression. Actual data transfer is not done during its execution, as can be seen by the neglectable amount of time spend in all `MPI_Test` calls for all 100 iterations (Figure 4.20). Also, the used core is available for useful work during the communication independent work phase.

All in all this benchmark shows that, for the used MPI implementations, not only synchronization overlap can be achieved, but the data transfer can also be overlapped with computation in cases where the programmer calls `MPI_Test` at suitable times. Nevertheless, these calls need to be implemented manually by the programmer.

4.2.6. Using MPI Datatypes

The layout of the data to be communicated between the different ranks is not necessarily sequential in memory. In order to be able to send this data in one single MPI message, MPI offers derived datatypes (see Section 2.2). They can be used to describe non-sequential memory regions in a single identifier which in turn can be passed to an MPI communication function as send or receive buffer.

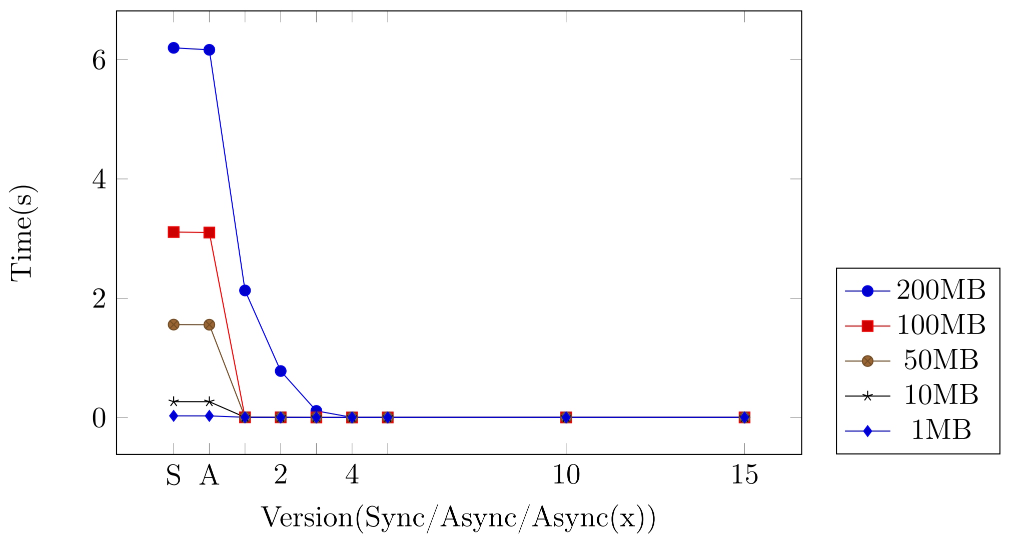


Figure 4.18.: Benchmark Results: Time spent in `MPI` functions only. 1 `MPI` rank each on two nodes of the SuperMUC using IBM MPI. 1 message sent per iteration and rank using a sequential memory buffer. Processes pinned to core number 8 of the respective nodes.

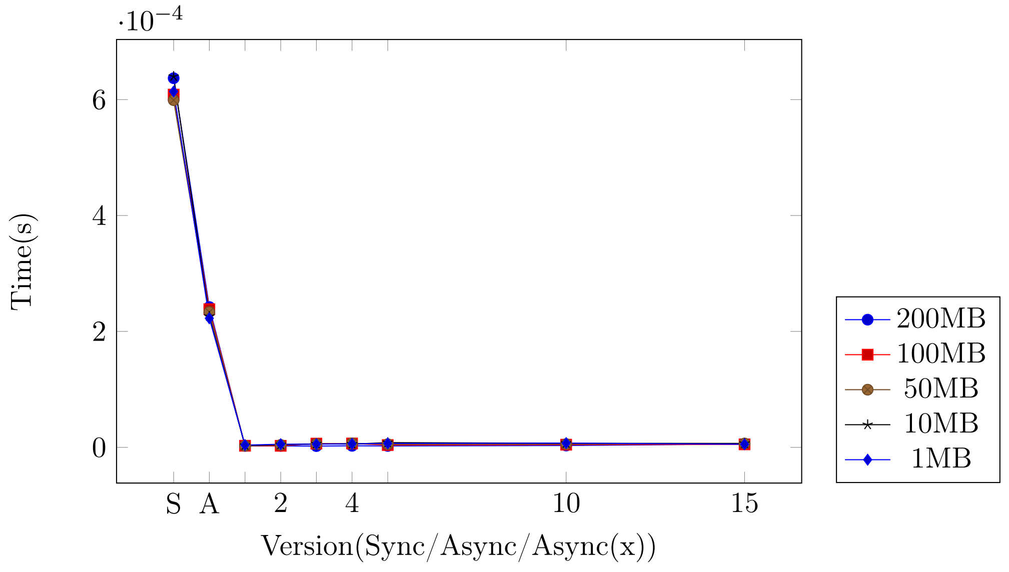


Figure 4.19.: Benchmark Results: Time spent in `MPI` communication initialization functions only (`MPI_Irecv`, `MPI_Isend`). 1 `MPI` rank each on two nodes of the SuperMUC using IBM MPI. 1 message sent per iteration and rank using a sequential memory buffer. Processes pinned to core number 8 of the respective nodes.

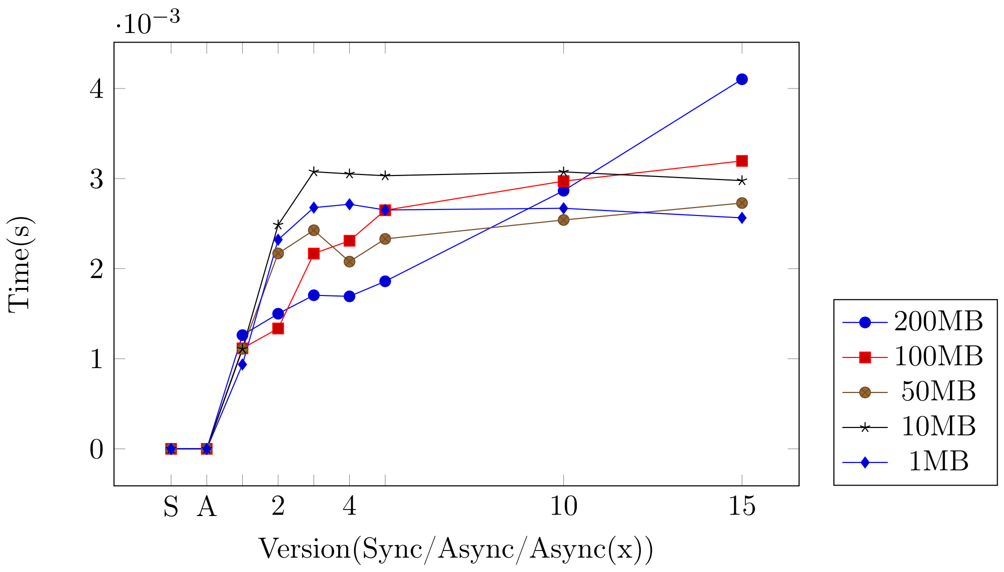


Figure 4.20.: Benchmark Results: Time spent in `MPI` communication advancement functions only (`MPI_Test`). 1 `MPI` rank each on two nodes of the SuperMUC using IBM MPI. 1 message sent per iteration and rank using a sequential memory buffer. Processes pinned to core number 8 of the respective nodes.

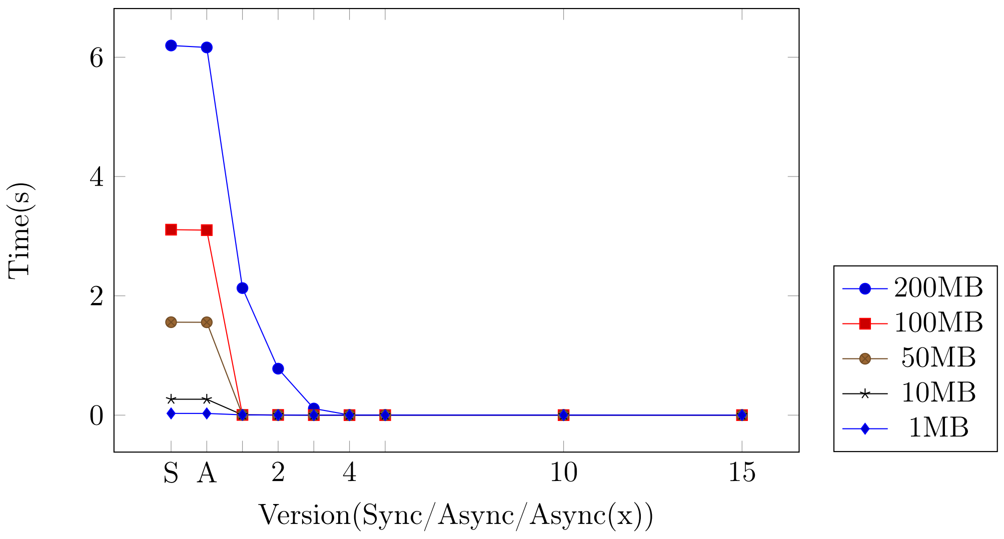


Figure 4.21.: Benchmark Results: Time spent in `MPI` communication termination functions only (`MPI_Wait`). 1 `MPI` rank each on two nodes of the SuperMUC using IBM MPI. 1 message sent per iteration and rank using a sequential memory buffer. Processes pinned to core number 8 of the respective nodes.

Tests executed with the benchmark using an MPI_Datatype,which has been created with MPI_Type_vector show different results in comparison to using sequential buffers. Defining the MPI_Datatype to use the first half of every megabyte in memory, it is possible to achieve overlap using IBM MPI on the SuperMUC. The results in Figure 4.22 show that the amount of overlap is very close to the overlap observed in the results presented above (see Figure 4.16). Nevertheless, for the same amount of work and the same overall message sizes, the benchmark execution times increase dramatically when using MPI_Datatypes. This is a result of necessary data-movement i.e. sequentializing data in memory for the actual data transfer and copying the received data to the corresponding regions in the receive buffer. This overhead can also be seen when using a similar MPI_Datatype on the ICE cluster with the systems default MPT MPI.
Comparing Figure 4.23 to the results shown before in Figure 4.13 shows that while using the same amount of work and the same message sizes per iteration and the same amount of iterations, the overall runtime of the benchmark increases when the used message buffers are defined through MPI_Type_vector. Additionally, even a large number of MPI_Test calls (up to 75 in this case) does not result in any overlap. In this scenario on this system, it is better to manually use a sequential send/receive buffer. This is done by copying the data corresponding to the MPI_Type_vector from the source data into a temporary send buffer before the call to MPI_Isend and from a temporary receive buffer into the desired memory regions corresponding to the MPI_Type_vector after the successful MPI_Waitall. Figure 4.24 shows that the synchronous version of this code (labeled S) is worse than using MPI_Datatypes directly. Nevertheless, the use of MPI_Test results in sufficient overlap to outweigh the overhead of manual data movement with only two used MPI_Test calls. Nevertheless, it can be confirmed that having data corresponding to one message buffer in a sequential memory region is the best approach.

All in all the benchmark results show that overlap, especially data transfer overlap, is not guaranteed through the use of nonblocking asynchronous MPI functions for the tested MPI implementations on the used HPC systems. With some manual placement of MPI_Test calls as communication advancement functions, the overlap can be achieved in most situations. Nevertheless, while a few MPI_Test calls are mostly sufficient to overlap the tested messages, their timing is important. In cases where two MPI_Test calls, executed evenly during the computation phase, are sufficient, the tests using more calls show that when the MPI_Test calls are executed more often and earlier, more than two calls are necessary before MPI_Test returns successful communication.

As MPI codes usually do not call MPI library functions between the asynchronous communication initialization and the corresponding MPI_Wait calls, no overlap is achieved. In order to provide this overlap automatically, an approach using an MPI aware OpenMP runtime in hybrid MPI-OpenMP programming approaches will be discussed throughout the rest of this Chapter.

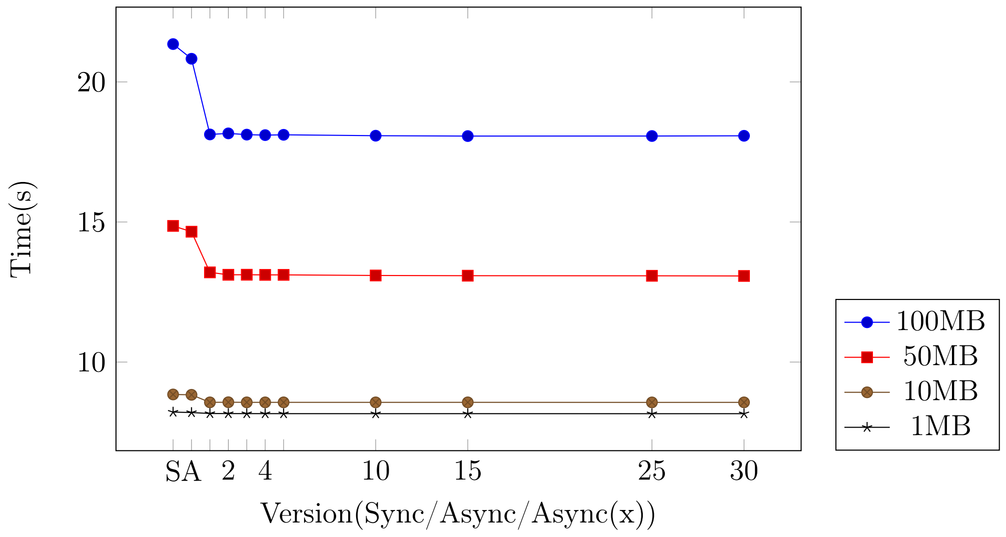


Figure 4.22.: Benchmark Results: 1 `MPI` rank each on two nodes of the SuperMUC using IBM MPI. 2 messages sent per iteration and rank using a nonsequential `MPI` datatype as memory buffer (every first half of every megabyte in memory). Processes pinned to core number 8 of the respective nodes.

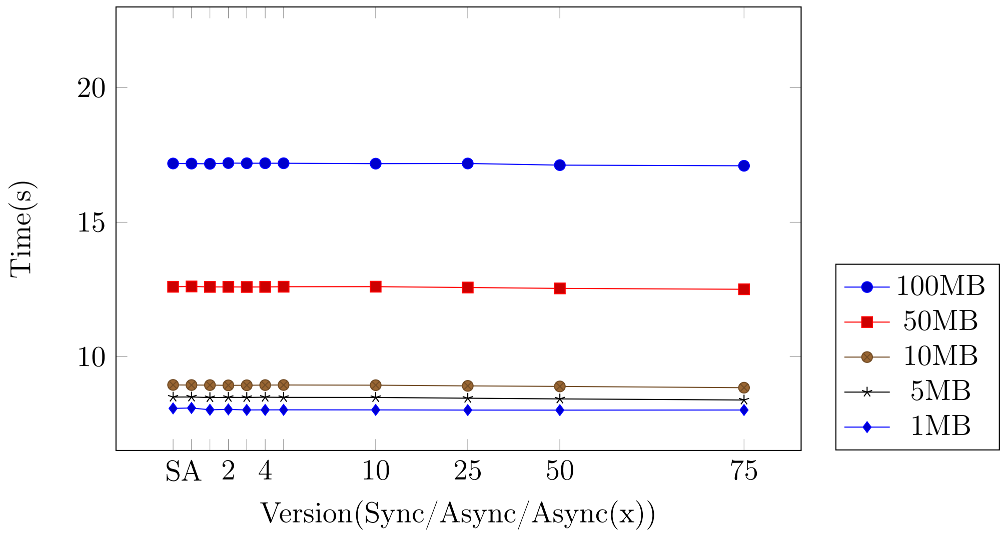


Figure 4.23.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using MPT MPI. 2 messages sent per iteration and rank using a nonsequential `MPI` datatype as memory buffer (every second megabyte in memory).

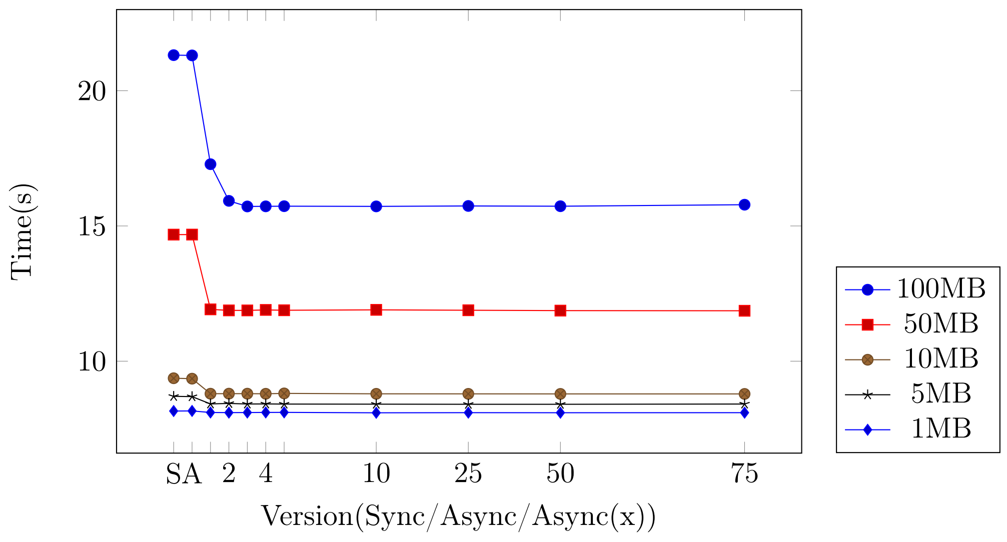


Figure 4.24.: Benchmark Results: 1 `MPI` rank each on two nodes of the ICE Cluster using MPT MPI. 2 messages sent per iteration and rank exchanging nonsequential memory regions (every second megabyte in memory). Temporary message buffers are used and the data is being sequentialized before a send operation and distributed after a receive operation manually.

4.3. Advancing Asynchronous MPI Communication at OpenMP Scheduling Points

As shown in Section 4.2, asynchronous communication is not always available and overlap of communication and computation is not guaranteed when using the asynchronous communication functions of MPI. Nevertheless, with manual calls to `MPI_Test` real asynchronous behavior and overlap is possible. In real life applications, this approach needs manual placement of the advancement functions into the communication independent work code. Additionally, as the perfect number of advancement calls and their timing differ for different systems and different MPI implementations, a lot of knowledge of the code behavior is necessary. Seeing as no MPI calls are usually made in between the communication initialization and termination, another automated way of taking care of the advancement automatically would be desirable.
As described in Section 2.3, the combined use of MPI and OpenMP in a hybrid fashion does offer many advantages. Matching the hybrid setup in modern HPC systems, combining shared memory and distributed memory parallelization, this approach can be used to automate the message progression based on the results presented in Section 4.2. In optimized, hybrid parallel applications, the communication independent work can be assumed to be large in comparison to the

work necessary for communication and the communication related work. With a hierarchical MPI-OpenMP approach, this MPI rank specific communication independent work is being parallelized using OpenMP in a shared memory fashion. The OpenMP runtime is therefore active during the time in which calls to message advancement functions are necessary and can be used to automatically execute them in moments useful for both the message progression and the overall program.

Manually placing the `MPI_Test` calls into the OpenMP worksharing used to parallelize the communication independent work is not feasible. In cases where this is a loop parallelized using an OpenMP for construct, placing additional code inside the loop results in a lot of overhead as it is being executed for every scheduled index. Additionally, this approach neither takes into account the timing of the `MPI_Test` calls nor the question of which OpenMP thread should take care of it at which point in time.
A different approach would be to place the communication dependent work and the communication advancement into OpenMP tasks. This approach has implications on the parallelization of the communication independent work. As of OpenMP 4.0 [61], it is not possible to mix tasks with worksharing through loop parallelization. I.e. there is no direct connection between the work of OpenMP `for` loops and created tasks and therefore no central workpool [85]. As a direct result, the communication independent work must be parallelized using tasks, which would be added to the same task pool as the tasks related to the communication. Nevertheless, restrictions exist which make this approach unusable for an efficient scheduling of communication advancement functions. No priority can be assigned to OpenMP tasks. Tasks cannot be coupled with preconditions for their scheduling or execution. While OpenMP provides a task yield construct, which can be used to interrupt a possible thread testing the state of pending communication, it is not possible to guarantee that it is actually postponed. Even in case the task is replaced and another task started, it is not possible to influence the timing of the yielding task.
Finally, using tasks for the parallelization of the main body of work in a NUMA environment might not be the best approach due to the placement of memory and execution of corresponding tasks working on it to different parts of the NUMA domain.

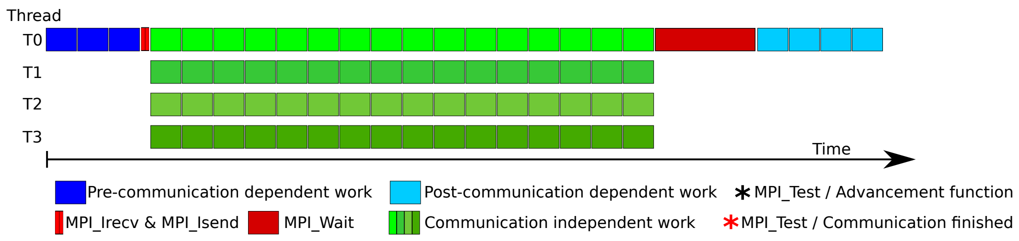
In order to use the OpenMP runtime to efficiently take care of the communication advancement, and additional optimizations for hybrid codes described later, a more general view on asynchronous MPI communication in a hybrid MPI-OpenMP context is necessary. Algorithm 4.6 outlines the central generalized steps of a hybrid MPI-OpenMP program. Overall, as discussed above, two kinds of work exist: Communication dependent and communication independent work. Communication dependent work can again be distinguished in two categories. First, work depending on the send buffer of following communications

has to be executed (Line 2). This work might be parallelized using OpenMP, but is not necessarily best suited for this. The overhead of OpenMP parallelization might be bigger than the benefits of parallelization in cases where too little work is to be done or due to the placement of allocated memory throughout possible NUMA domains. At the same time, not using all available OpenMP threads for this work results in idle times and therefore unused resources.
After this pre-communication dependent work is done, the actual communication can be initialized, e.g. by calling `MPI_Irecv` and `MPI_Isend` (Line 4). The main body of work, communication independent and well suited for OpenMP parallelization, can be executed. This is also the time when the MPI implementation and the execution environment could take care of the actual data transfer (Lines 5 and 6). Only after successful communication termination is guaranteed through a call to `MPI_Wait` (`MPI_Waitall`) (Line 7) can the work be done which is dependent on the received data (Line 9). Again, the same aspects concerning parallelization apply as to the pre-communication dependent work.
For real applications, e.g. the example application used in Chapter 5, multiple messages to different communication partners might be necessary. These can have individual (pre- and post-)communication dependent work.

Algorithm 4.6 Asynchronous communication in a hybrid MPI-OpenMP code

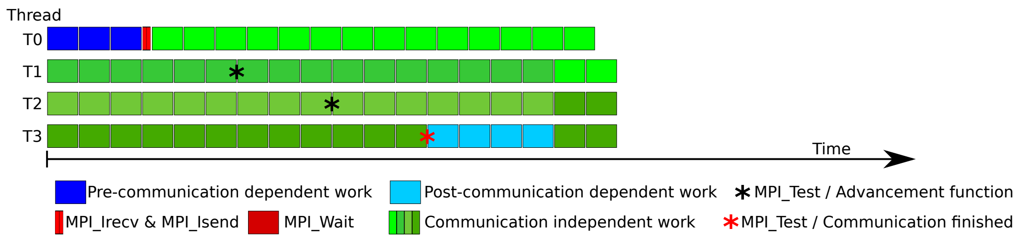
```
 1: OpenMP FOR
 2:   Work necessary before communication can start
 3:                          ▷ (not necessarily "best" suited for parallelization)
 4: MPI_Irecv and MPI_Isend
 5: OpenMP FOR
 6:   Communication independent work
 7: MPI_Waitall
 8: OpenMP FOR
 9:   Work necessary after communication has finished
10:                          ▷ (not necessarily "best" suited for parallelization)
```

Making the OpenMP runtime aware of the MPI communication can solve different problems and provide multiple improvements: At OpenMP scheduling points (e.g. start of new blocks when using `dynamic` or `guided` scheduling or the `static-ws` scheduling presented in Chapter 3), the OpenMP runtime can decide whether or not threads need to call available communication advancement functions. As shown in the previous section, this can result in real overlap. Additionally, introducing knowledge about the different (asynchronous) communications and the corresponding dependent work allows for the compiler and the runtime to schedule the pre-communication dependent work to available threads in an optimized way at the same time as allowing the remaining threads to start executing communication independent work.
Through the use of communication advancement functions and the resulting



Thread
T0
T1
T2
T3
Time
Pre-communication dependent work
Post-communication dependent work
MPI_Test / Advancement function
MPI_Irecv & MPI_Isend
MPI_Wait
Communication independent work
MPI_Test / Communication finished

Figure 4.25.: Possible timeline for an iteration in one MPI rank using 4 OpenMP threads for work parallelization. Pre-communication and post-communication dependent work can be parallelized using OpenMP. MPI communication and OpenMP parallelization as provided by the current standards.



Thread
T0
T1
T2
T3
Time
Pre-communication dependent work
Post-communication dependent work
MPI_Test / Advancement function
MPI_Irecv & MPI_Isend
MPI_Wait
Communication independent work
MPI_Test / Communication finished

Figure 4.26.: Possible timeline for an iteration in one MPI rank using 4 OpenMP threads for work parallelization. Pre-communication and post-communication dependent work are sequential and cannot be parallelized using OpenMP. The proposed approach of advancing communication at OpenMP scheduling points is used together with the scheduled `static-ws`, providing work stealing while trying to take into account the first-touch memory placement strategy (described in Chapter 3).

overlap, communication termination can be observed directly. This allows the OpenMP runtime to schedule post-communication dependent work with a higher priority whenever possible. Together with possible work balancing between the used OpenMP threads (e.g. using the `static-ws` scheduling), this results in optimized resource usage.

Figure 4.25 depicts the behavior of one MPI rank sending and receiving one message respectively for the usual case where MPI calls are done outside OpenMP parallelized regions. In this case the pre-communication and post-communication dependent work is not parallelized through the use of OpenMP, leaving three of the four available threads idle. As no MPI calls are done during the communication independent work and no overlap can be achieved, `MPI_Wait` returns only after taking care of the data transfer. The same code executed within an MPI aware OpenMP runtime can improve the runtime through the described advantages as depicted in Figure 4.26. While the main thread (`T0`)

takes care of the pre-communication dependent work, the other three threads (`T1-T3`) can start working on their part of the communication independent work. After the OpenMP runtime is being notified about the executed `MPI_Irecv` and `MPI_Isend` calls, it can start scheduling communication advancement calls, e.g. as in the benchmark described earlier through calls to `MPI_Test`. These are done at OpenMP scheduling points which happen due to the use of OpenMP schedules `static`, `guided` or, as in this example, the proposed schedule `static-ws` (Chapter 3). Once any thread (e.g. thread `T3` in Figure 4.26) encounters the successful termination of the communication, the OpenMP runtime is able to schedule the post-communication dependent work directly.
Threads working on communication dependent work (`T0` and `T3` in the example) postpone their communication independent work. After finishing their own part of the communication independent work, the other threads therefore encounter "stealable" work, which is now scheduled to the otherwise idle threads, resulting in optimized resource usage and work balancing.

Additional advantages of an MPI aware OpenMP runtime can be seen when looking at a different example depicted in Figure 4.27 and Figure 4.28. Here the presented MPI rank communicates with two communication partners, sending and receiving one message to and from each, respectively. Additionally, the pre-communication dependent work for the send operation to one communication partner and both sets of post-communication dependent work can be parallelized using the available OpenMP threads. As shown in Figure 4.27, MPI calls are done outside the OpenMP parallel regions in the traditional approach. While idle times exist here, all threads can participate in all kinds of work necessary. Nevertheless, an implicit barrier for the threads can exist at the end of each work block (e.g. at the end of used OpenMP `for` parallelized loops). Also, data transfer necessary for the first `MPI_Wait` will have to be finished together with the corresponding post-communication dependent work even if it is not available from the communication partner at this point in time. The second communication can be done only afterwards, even if it would have been available earlier.
Using an MPI aware OpenMP runtime removes the barriers, the idle times and allows for a re-ordering of the parallelized post-communication dependent work, as seen in Figure 4.28. As soon as any thread (`T3` in the example) encounters any successful communication (the second one from the "standard" version in this example), the corresponding post-communication dependent work can be scheduled directly to all available threads at their next scheduling points. Again, the use of the schedule `static-ws` provides these while additionally providing work balancing through work stealing.

After presenting how OpenMP can be made aware of MPI from the programmers' point of view in the following Section 4.4, it will be shown that this approach can be very effective in real HPC application in Chapter 5.

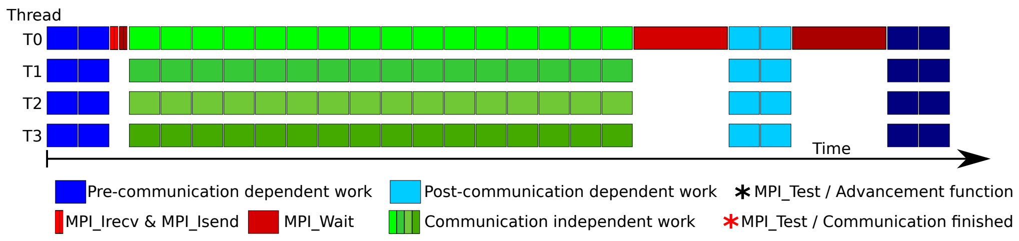

Thread
T0
T1
T2
T3
Time
Pre-communication dependent work
Post-communication dependent work
MPI_Test / Advancement function
MPI_Irecv & MPI_Isend
MPI_Wait
Communication independent work
MPI_Test / Communication finished

Figure 4.27.: Possible timeline for an iteration in one MPI rank using 4 OpenMP threads for work parallelization. Messages are being sent to and received from two communication partners. Pre-communication dependent work for the first send operation and post-communication dependent work for each receive operation separately. Pre-communication and post-communication dependent work can be parallelized using OpenMP. MPI communication and OpenMP parallelization as provided by the current standards.

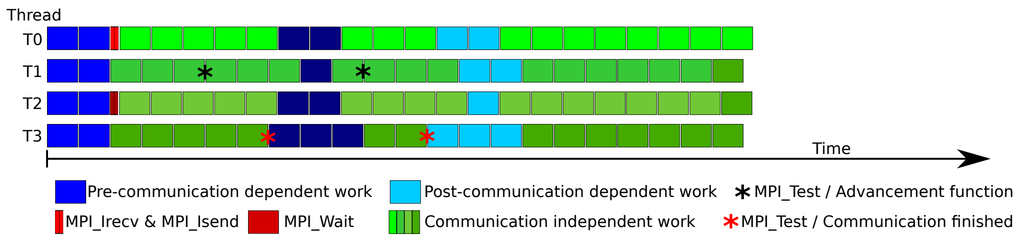

Thread
T0
T1
T2
T3
Time
Pre-communication dependent work
Post-communication dependent work
MPI_Test / Advancement function
MPI_Irecv & MPI_Isend
MPI_Wait
Communication independent work
MPI_Test / Communication finished

Figure 4.28.: Possible timeline for an iteration in one MPI rank using 4 OpenMP threads for work parallelization. Messages are being sent to and received from two communication partners. Pre-communication dependent work for the first send operation and post-communication dependent work for each receive operation separately. Pre-communication and post-communication dependent work can be parallelized using OpenMP. The proposed approach of advancing communication at OpenMP scheduling points is used together with the scheduled `static-ws`, providing work stealing while trying to take into account the first-touch memory placement strategy (described in Chapter 3)

4.4. Introducing A New OpenMP Construct `commtask`

As discussed in the previous section, adding awareness of communication concerning distributed memory parallelization to the shared memory parallelization environment OpenMP can add the necessary information to the execution environment to enable communication-computation overlap (including data transfer overlap). Additionally it provides potential for work balancing and optimization of resource usage inside the OpenMP domains. In order to provide this information to the runtime, the programmer needs to be supplied with additional OpenMP directives.
Looking at the example presented in the previous section in Algorithm 4.6, one MPI communication to and one from a communication partner is present, each with pre-communication or post-communication dependent work, respectively. In order for the OpenMP runtime, used to parallelize the respective work blocks, to know that the communication dependent steps can be mixed into the communication independent work pool, an extension, namely `commtasks`, are proposed in this section. They have previously been presented by the author in [14].
Before a formal definition in the style of the OpenMP standard is proposed in the next Section 4.4, its realization for the aforementioned example can be seen in Algorithm 4.7. As the `MPI_Irecv` call (Line 3) is in itself not depending on any work, it can be included in its own `commtask init` region (Line 1-Line 4). In order to match it to the corresponding `commtask finalize` region, it is assigned the `ID 'R'`. The second `commtask init` region (Line 5-Line 11) with `ID 'S'` combines the pre-communication dependent work necessary to be executed before the `MPI_Isend` call with the same. In cases where this work can be parallelized using OpenMP `for`, this work can be scheduled to all participating threads, otherwise it is scheduled sequentially to one thread, before (in any case) one thread starts the asynchronous communication through the call to `MPI_Isend`.
After the communication independent, OpenMP parallelized work is defined (Lines 13 and 14), the `commtask init` regions must be matched with `commtask finalize` regions. In the presented example, both `MPI_Irecv` and `MPI_Isend` are matched in the single `MPI_Waitall` call. Matching this, both `commtask init` regions are matched with the same `commtask finalize` region (Line 16-Line 23) by the assignation of both `IDs 'R'` and `'S'`. Before the `commtask` block starts with the blocking communication completion function, `#pragma omp commtask-advancecheck` defines the call to the corresponding nonblocking function used to check whether or not the corresponding messages have successfully been transferred. This function should also guarantee (as is the case with `MPI_Test` and `MPI_Testall` for MPI) that its usage results in message progression and real overlap. Finally, the post-communication (on `MPI_Irecv`) dependent work follows.
The OpenMP runtime can now schedule all `commtask init` regions to available threads directly, adding OpenMP parallelized work in these to the thread teams

work pool. In order to get optimal behavior, this is best done with a higher priority. As described in the previous section, the runtime can now use all available work in the work pool to fully utilize all available threads at all times. At scheduling points, either after work on `commtasks` or blocks of communication (in-)dependent work, the provided communication status test function is scheduled, probably with additional timing considerations. Once it returns successful completion of the communication, the post-communication dependent work of the corresponding `commtask finalize` region can again be added to the common work pool and completed with the available threads.

Algorithm 4.7 Asynchronous communication in a hybrid MPI-OpenMP code using the proposed `commtask` to add MPI awareness to the OpenMP runtime

```
#pragma omp commtask init ID(R)
{
  MPI_Irecv
}
#pragma omp commtask init ID(S)
{
  OpenMP FOR
    Work necessary before communication can start
                                ▷ (not necessarily "best" suited for parallelization)
  MPI_Isend
}

OpenMP FOR schedule(static-ws)
  Communication independent work

#pragma omp commtask finalize ID(R,S)
{
  #pragma omp commtask-advancecheck MPI_Testall
  MPI_Waitall
  OpenMP FOR
    Work necessary after communication has finished
                                ▷ (not necessarily "best" suited for parallelization)
}
```

The `ID` can be used to match independent communications with their corresponding work in order to allow the runtime to reorder and execute in parallel the associated steps and work (e.g. as described in Figure 4.28). This allows for post-communication dependent work to be scheduled directly without having to wait for communications to finish which appear beforehand in the code, as discussed previously.

`commtask` Definitions for the OpenMP Standard

In order to formalize the proposed `commtask` approach, different parts have to be defined in a way matching the OpenMP standard. Besides a definition of the worksharing construct `commtask` in itself, an Internal Control Variable (ICV) is needed to define the priority with which the work defined in the `commtask` regions is being scheduled. Consistent with the ICVs available in OpenMP, its manipulation needs to be defined using an environment variable and execution environment routines to change (set) and read (get) its value during runtime.
The definitions take into account the results gained through the benchmark presented in Section 4.2 and their integration into a shared memory programming environment as discussed in Section 4.3. Care has been taken to ensure that code run in OpenMP environments ignoring the proposed `commtasks` can still run as expected with the current OpenMP standard and also, as is the case with all other OpenMP directives, when OpenMP is disabled entirely. The definitions leave room for optimizations in `OpenMP` implementations.
Additionally, the definitions do not limit possible implementations to scenarios where message progression has to be done manually. In cases where the communication library can create and efficiently use a progress thread, the `commtask` construct can be used as proposed here. The OpenMP runtime can be supplied with a nonblocking communication completion function which only checks for completion, without intending to advance messages. Additionally, it can easily be extended to work with the communication libraries progress thread by being notified about successful communication termination by the progress thread. The calls to the checking function could be omitted.
Nevertheless, as no available MPI implementation on the used HPC systems for this work provides progress threads, the `commtask` construct outlined here is targeted especially to work with manual progression functions, such as `MPI_Test`.

4.4.1. OpenMP Construct `commtask`

Summary

The `commtask` construct is a non-iterative worksharing construct which describes asynchronous communication and the corresponding pre-communication and post-communication dependent work. The work and communication advancement described in the structured blocks through a `commtask` is being distributed and executed by the threads in the current team. Each `commtask` consists of two parts. One contains the pre-communication dependent work and the corresponding communication initialization functions (e.g. in combination with MPI calls to `MPI_Isend` [and `MPI_Irecv`]). The second part begins with a communication completion function followed by the post-communication dependent work. Through communication library awareness, the OpenMP runtime can then schedule this work and available communication advancement function calls together with the other work available in the current team's work pool.

Syntax

The syntax of the `commtask` construct is defined as shown in Algorithm 4.8.

Algorithm 4.8 C Syntax of `commtask` construct

```
#pragma omp commtask init [ID(<label>[,<label>[,...]])] new-line
{
  [structured-block]
  <communication-initialization-function> new-line
}

[structured-blocks]

#pragma omp commtask finalize [ID(<label>[,<label[,...]])] new-line
{
   #pragma omp commtask-advancecheck >nonblocking-communication-
  advancement-and-statuscheck-function> new-line
  <blocking-communication-termination-function> new-line
  [structured-block]
}
```

Where label is an integer or char identifier and priority is `0` (same priority as other work in the work pool) or `1` (higher priority as the other work in the work pool).

Binding

The binding thread set for a `commtask` is the current team. A `commtask` region binds to the innermost enclosing parallel region. Only the threads of the team executing the binding parallel region participate in the execution of the structured blocks and the calls to the referenced communication functions.

Description

The `commtask` worksharing construct can be used in hybrid approaches combining OpenMP with shared memory parallelization paradigms (e.g. Message Passing Interface (MPI)). The work defined in the structured blocks enclosed by a `#pragma omp commtask init` region will be merged with the current work-pool of the current team. It will be scheduled to available OpenMP threads

according to the set `commtask`-priority (see ICV `commtask-priority-var`, env-variable and library call).
In cases where a nonblocking alternative to the communication completion function is provided through `#pragma omp commtask-advancecheck`, the runtime schedules calls to this function to available threads at OpenMP scheduling points and uses the results to be notified about successfully finished communications. Corresponding post-communication dependent structured blocks are enclosed in `#pragma omp commtask finalize` regions. They will be scheduled to the available team threads with the `commtask` priority. In cases where no such function is provided, `commtask finalize` regions will be scheduled "in order" to threads after no other work is in the common work pool.
The ID clause can be used to distinguish between different communications and to match `commtask init` with `commtask finalize` regions. In cases where not all `commtask` regions are identified with an ID, all init blocks will have to be finished before finalized blocks are being scheduled and before any nonblocking communication advancement and checking function is being called. With ID provided for all `commtask` regions, this order is imposed only on `commtask` regions containing the same IDs.

Restrictions

Restrictions to the `commtask` construct and the enclosed code are as follows:

- Each communication initialization call in `#pragma omp commtask init` needs a matching communication completion call in `#pragma omp commtask finalize`.
- The code must work correctly in cases where the `#pragma omp commtask` pragmas are being ignored.
- In case no ID is provided,
  - the blocks defined through `#pragma omp commtask init` regions must be executable in parallel, and their order must be interchangeable.
  - the blocks defined through `#pragma omp commtask finalize` regions must be executable in parallel, and their order must be interchangeable.

  otherwise

  - the blocks defined through `#pragma omp commtask` regions must be executable in parallel, and their order must be interchangeable unless an order is imposed through matching IDs.

Cross References

- `commtask-priority-var` ICV

- `OMP_COMMTASK_PRIORITY` environment variable
- `omp_set_commtask_priority` routine
- `omp_get_commtask_priority` routine

4.4.2. Internal Control Variables

1. `commtask-priority-var` - controls whether code from `commtask` regions is being scheduled with higher priority or not. There is one copy of this ICV per data environment.

4.4.3. Environment Variables

`OMP_COMMTASK_PRIORITY` sets the priority for scheduling work defined in `commtask` regions by setting the `commtask-priority-var` ICV. Can be set to `0` (normal priority) and `1` (default - high priority).

Cross References

- `commtask-priority-var` ICV
- Controlling OpenMP `commtask` related work scheduling
- `omp_set_commtask_priority` routine
- `omp_get_commtask_priority` routine

4.4.4. Execution Environment Routines: `omp_set_commtask_priority`

Summary

The `omp_set_commtask_priority` routine affects the scheduling of work defined through the use of `#pragma omp commtask` by setting the `commtask-priority-var` ICV.

Format

Algorithm 4.9 C Syntax of the omp_set_`commtask`_priority Routine

```
1: void omp_set_commtask_priority(int priority);
```

Constraints on Arguments

The value of the argument passed to this routine must evaluate to `0` (normal priority) or `1` (high priority), or else the behavior of this routine is implementation defined.

Binding

The binding task set for an `omp_set_commtask_priority` region is the generating task.

Effect

The effect of this routines is to set the value of the `commtask-priority-var` ICV.

Cross References

- `commtask-priority-var` ICV
- `OMP_COMMTASK_PRIORITY` environment variable
- Controlling OpenMP `commtask` related work scheduling
- `omp_get_commtask-priority` routine

4.4.5. Execution Environment Routines: `omp_get_commtask_priority`

Summary

The `omp_get_commtask_priority` routine returns the value of the `commtask-priority-var` ICV.

Format

Algorithm 4.10 C Syntax of the omp_get_`commtask`_priority Routine

```
1: void int omp_get_commtask_priority(void);
```

Binding

The binding task set for an `omp_set_commtask_priority` region is the generating task.

Effect

The effect of the `omp_get_commtask_priority` routine is to return the priority assigned to work defined in `commtask` regions.

Cross References

- `commtask-priority-var` ICV
- `OMP_COMMTASK_PRIORITY` environment variable
- Controlling OpenMP `commtask` related work scheduling
- `omp_set_commtask-priority` routine

4.5. Discussion

Asynchronous MPI functions have been part of the `MPI` standard for a while. They split the necessary communication steps into different parts with the goal to improve performance. These parts are 1) communication initialization, 2.) synchronization and data transfer, and 3) communication termination. Communication initialization sets up the communication parameters, such as sender and receiver information, describes the message buffers and makes the start of the communication known to the `MPI` runtime. Functions available for communication initialization include `MPI_Irecv` and `MPI_Isend`. Communication termination functions, such as `MPI_Wait`, are used to make sure that the communication has successfully been finished and that the used message buffers can safely be used again, which is not the case after the initialization. While these two parts of the communication steps are clearly assigned to the respective `MPI` functions, can the second step be implemented differently, depending on the `MPI` implementation and other factors such as hardware support. As discussed above, synchronization overlap is available in most `MPI` implementations, but data transfer overlap has been reported to not work well or exist at all.
In this chapter, a benchmark has been presented, discussed and executed. The results show that for no `MPI` implementation available on the used HPC systems does using asynchronous `MPI` functions result in data transfer ovelap with useful computation. The actual data movement is happening inside the used `MPI_Wait` calls, which results in a communication phase equal to using blocking communication functions. One way to overcome this problem is using manual progession function. One function which can be used, as it eventually must finish communication, is `MPI_Test`. The presented benchmark uses this function in order to achieve data transfer overlap. The results show that data transfer can be achieved, but timing and number of calls to the progression function are the important factors in regard to performance. Calling the progression function

too often or frequently results in overhead or even no overlap at all.
For applications parallelized in a hybrid fashion, i.e. MPI-OpenMP, the message progression calls can be automatized efficiently as presented above. An OpenMP extension, namely the commtasks, have been proposed and formalized. Adding MPI awareness to the OpenMP runtime allows the OpenMP runtime to take care of choosing when, how often and by which thread a call to message progression functions is necessary. This allows for optimizations to be included into the OpenMP and MPI installations and implemented by system professionals, not application programmers. Additionally, this allows for optimizations tuned for the used system, removing the need to adjust the applications when moving to a different HPC environment.
The proposed commtasks will be used in a stencil code representing a wide range of parallel applications in the following chapter.

CHAPTER 5

Real Asynchronous MPI-Communication: Proof of Concept

In high performance computing, many different applications with a large variety of requirements exist. In order to understand and possibly predict the behavior of these applications on existing or future HPC platforms, the applications are often grouped together. These groups or, as they are often called, classes, are based on system requirements, memory access patterns and data layout, etc. The authors of [3] call their class specifications `dwarfs`, which are defined as "a pattern of communication and computation common across a set of applications." These `dwarfs` are used in other work: The authors of [95] record the communication patterns of MPI parallelized HPC applications and try to match the given application to the berkeley dwarfs presented in [3] through pattern analysis on the resulting communication graphs.
Classification of HPC applications is also done in regard to more recent discussions in the area: The authors of [21] use the observed communication patterns together with power consumption information in order to classify applications running in HPC or Cloud environments. Their goals include optimization of resource usage through platform providers, which is becoming a more important concern with faster and larger supercomputers.

An area where classifications of HPC algorithms is important is benchmarking. In order to see how well hardware and software combinations perform in regard to the expected work load, it is not possible to run or simulate every possible production code. Therefore, supercomputing centers and HPC system vendors, as well as researchers, use benchmark suites which test representative codes executing computations and communications corresponding to the class of applications of interest, e.g. the expected work load of a planned HPC systems. One well known benchmark suite is the NAS Parallel Benchmark (NPB) suite [4] together with the NAS Mutli-Zone (NPB-MZ) extensions described in [89]. The

NPB describes representative problems for different classes of algorithms, including "MT," a multigrid kernel, and "EP," the embarrassingly parallel kernel. The NPB-MZ extensions focus on kernels with multi-level parallelism with the goal of providing portable test scenarios for hybrid and multi-level parallelization approaches and corresponding tools.

While these benchmarks are important and a very good way to analyze the behavior of the application class on different HPC systems, the classes can also be used in regard to novel approaches. Approaches like the `commtasks` presented in this work can be applied to a representative application in the class and the results allow for conclusions regarding all applications which are part of it.

The fifth `Berkeley Dwarf: Structured Grids` is based on stencil codes and has applications in different areas. In embedded computing, applications can be found in the automotive area, for example the (FIR) and (IIR) filters used in the EEMBC benchmark engine knock detection, vehicle stability control, and occupant safety systems. Other embedded computing applications are the encoding and decoding of MPEG-2 and MPEG-4 [3]. More important for this work, applications can also be found in general purpose computing. The authors of [3] list quantum chromodynamics, magneto hydrodynamics, fluid dynamics, finite element methods, and weather modeling as example application areas. Finally, stencil codes can be found in the area of graphics algorithms [3, 20].

In HPC applications, "the main application of stencil-based computations include numerical PDE solvers that use a finite difference or multigrid method." [20, 24]. Research using stencil codes includes the work on an auto-tuning framework which tests a wide range of possible optimizations including NUMA affinity, blocking, prefetching, and others [24]. The authors state that, with this framework, they reach the fastest multicore stencil performance up to publication date. The authors of [41] present cache optimization techniques for multigrid methods in the context of PDE solvers. Stencil based kernels are further used to study how trends in memory system organization influence the efficacy of traditional cacheblocking optimizations [23]. The authors of [97] use them as part of their simulation workload in order to explore novel HPC system approaches using alternative architectures. Some of the presented work also mentions using nonblocking MPI functions in order to overlap communication and computation (e.g. [20]), but it is never discussed if this actually works. The results presented in the previous chapter suggest that communication is moved to the `MPI_Wait` call, providing only latency hiding. In regard to [20], this assumption is further backed by the fact that they cite [81] concerning MPI and communication overlap. While the authors of [81] do mention MPI as "emerging as a widely accepted standard" (at the time of publication), they do not make use of MPI but present basic research on hiding communication latency using a redefined UNIX send operation on local area networks (LAN).

Offloading and parallelizing computation to GPUs is also an area where the work on optimizing stencil codes underlines their relevance.The basic approach to run stencil codes on GPUs is described by the authors of [17]. The authors of [91] show their results for heterogeneous multi-CPU and multi-GPU implementations of the Jacobi's method using two-dimensional computational domains. For the bandwidth limited problem, they especially show how performance improvements can be achieved by using what they call a “wildly asynchronous” approach: removing or delaying synchronizations between the iterations. This can be done by using more iterations in order to get the same results. As shown in this work, the authors of [91] highlight the fact that optimizations which reduce or even remove the impact of high synchronization and communication costs should be very relevant to the design of future implementations and systems. This is relevant to the presented `commtask` approach, as these are properties of the same.
In [2], the authors focus on comparing two different GPGPU (General Purpose GPU) programming approaches, namely CUDA and OpenGL, using a weighted Jacobi iteration. The difference to the problem used here is the two dimensional character in their problem and the special case that, for their problem, only nine diagonals of the matrix describing the computational domain are non-zeros and can be saved in nine corresponding vectors. Besides comparing the execution times of the different approaches for different problem sizes on different hardware, the authors go into detail about the advantages and drawbacks of each approach including ease of use and portability questions.

5.1. Introduction to a Representative Example

One stencil code as described above is the Jacobi Relaxation method, which can be found at the heart of numerous linear solvers. It can be used to solve the Laplace's equation. Physical systems like temperature in a two- or three-dimensional object are modeled in a corresponding two- or three-dimensional array and initialized with starting values. These will then iteratively be recomputed as the (possibly weighted) average of a set of neighboring points in the used array. In this work, the array will be referred to as `computational domain`. With every iteration, the solution averages out more and will eventually reach the desired accuracy. [17, 44]
Inside each iteration the algorithm has a large potential for parallelization. As every stencil uses the values from the previous iterations for computing the new value, all stencils can be computed independently and therefore in parallel. While this would allow every stencil to be assigned to one computational core, this is neither practical nor efficient. Many things have to be taken into account when parallelizing the algorithm, including, but not limited to, hardware characteristics of the used parallel system, like the size of main memory, memory per core and number of available cores as well as memory access patterns and

communication patterns resulting from the parallelization. Basic approaches for parallelizing the Jacobi method can be found in [44] and more advanced approaches in the publications discussed above. For this work, distributed memory parallelization using MPI and hybrid distributed and shared memory parallelization using MPI together with OpenMP or Pthreads are the main target.
In order to see how well the `commtask` approach described in Chapter 4 works in real applications, it is applied to a three-dimensional Jacobi code. The basic parallelization approach and relevant topics will be discussed here, followed by two chapters on different approaches to split the used computational domain. Results for applying the `commtask` in combination with a one dimensional computational domain splitting have partially been published by the author in [14]. All computing systems used throughout this work are described in Appendix A. For all approaches, the base version used for comparison will be an optimized `MPI-Only` parallelized code with independent communication and computation phases. Building up to a hybrid approach combining the distributed memory parallelization with shared memory parallelization in a hierarchical manner incorporating the presented `commtask`, different other approaches will be presented and evaluated.
In this sense, the used Jacobi algorithm not only represents the stencil codes described above, but any code which can be parallelized like this:

1. The computational domain can be split into parts which are to be distributed to the individual MPI ranks used on the different nodes/sockets.
2. The blocks of the computational domain contain work which can again be split up into independent sub-parts. These can then be worked upon in parallel by the different (OpenMP-)threads.

5.1.1. Main Algorithm Steps

The general steps of the different approaches presented below are the same in all of them.

1. Initialization of the MPI environment (details below).
2. The program parameters are parsed on MPI rank zero and broadcasted to all other ranks using `MPI_Bcast`. This is necessary as MPI guarantees only that the command line parameters are passed to rank 0 and are not necessarily available on all other ranks.
3. Memory is allocated for all necessary datastructures, including the two used copies of the computational domain part assigned to each rank. These copies are alternately used in the iterations to store the computational domain state of the previously computed and the to be computed iteration. More details on the memory layout, initialization of the computational domain as well as the differences when using different kind of stencils will be discussed below.

4. When the necessary environment is set up, all MPI ranks (and their threads) synchronize before starting the time measurement. The Jacobi steps are repeated according to the desired iteration count and finished with another synchronization point before getting the end time measurements. This synchronization is used to include possible imbalances between the ranks.

5. After reporting the timing results, all memory regions allocated before are freed, and the program is terminated after calling `MPI_Finalize`.

Initialization of the MPI Environment

As all implementations presented here use MPI for shared memory parallelization, the initialization of the MPI environment is the first step. As described in Section 2.2, MPI defines different thread safety levels. For the MPI only versions, using `MPI_Init` is sufficient. For the hybrid approaches combining MPI with OpenMP or Pthreads, the availability of the needed thread safety level is being checked through `MPI_Init_thread`. For the `commtask` approach, the highest thread support, `MPI_THREAD_MULTIPLE` level is needed as any thread can call MPI library functions any time. For the other hybrid approaches, `MPI_THREAD_FUNNELED` and `MPI_THREAD_SERIALIZED` are appropriately used.

Concerning the MPI rank placement, it is important to place those ranks physically close together which communicate most frequently [53]. In cases where the computational domain is split along a single dimension, each rank r has to communicate only with ranks $r-1$ and $r+1$. By placing the ranks to the cores of the nodes by filling each node before placing a rank on a new node, the minimal inter-node communication is achieved. For the case where the computational domain is split along multiple axes (e.g. all axes as used in Section 5.3), the ranks have to communicate with up to 6 neighbors. MPI offers functions to optimize the rank placement for cases like these. Using the function `MPI_Cart_create`, a new MPI `communicator` is being created to which topology information is being attached [55]. First of all, this tells the MPI environment that the communication pattern of the program will be following three dimensional grid pattern. It also allows the MPI environment to reorder (if configured to do so) the ranks using the hardware information together with the topology information in order to minimize inter-node communication. Finally, additional functions (e.g. `MPI_Cart_rank`, `MPI_Cart_shift`) allow the programmer to get their communication partners as “neighbors in the grid” as opposed to manual computation of the necessary rank numbers.

5.1.2. Different Kinds of Stencils Used

In each Jacobi step $t+1$, i.e. each iteration of the program, every element in the computational domain is updated using the values of elements from step t.

The elements which are to be used in the update are defined by the so called stencil. Stencils are defined through the location of the source elements in regard to the target element. Two important characteristics of a stencil are the stencil order and the number of source elements. The order is defined through the farthest distance of a source element from the target element along one dimension [63]. When higher precision is desired in stencil codes, higher order stencils are used [26]. For example, the stencil used in [23] is a first order 7 point stencil. Here, the average of all direct neighbors in all three dimensions and the value of the target element in step t is used to compute the new value for step $t + 1$. The same kind of stencil is used in [97] with the difference that the presented algorithm needs values not only from the previous iteration t, but also from iteration $t - 1$. In addition to a good explanation of the stencil order, the authors of [63] use both a first order 19 point stencil and a sixth order 25 point stencil in their work.
Not all stencil based algorithms compute the average of the used stencil points but add a weight to each source element. These weights can be fixed in space and time or vary depending on the application. The authors of [20] state that they use "a more versatile stencil stemming from a real-world application." While the stencil used is also a first order 7 point stencil, the weights associated with the stencil points are fixed only in time, but not in space. Also the value of step t of the target element is used multiple times in the computation of its new value.

In this work, two kinds of stencils are being used:

- **First order 6 point stencil**: Referred to as the `1n-Stencil`, this stencil uses the direct neighbors of the element which is to be computed.

$$e_{ijk}^{t+1} = \frac{1}{6} * (e_{(i-1)jk}^{t} + e_{(i+1)jk}^{t} + e_{i(j-1)k}^{t} + e_{i(j+1)k}^{t} + e_{ij(k-1)}^{t} + e_{ij(k+1)}^{t}) \quad (5.1)$$

- **Fourth order 24 point stencil**: Refered to as the `4n-Stencil`, this stencil used 4 neighbors in both directions of each dimension in the computational domain.

$$\begin{aligned} e_{ijk}^{t+1} = \frac{1}{24} * (& e_{(i-1)jk}^{t} + e_{(i-2)jk}^{t} + e_{(i-3)jk}^{t} + e_{(i-4)jk}^{t} + \\ & e_{(i+1)jk}^{t} + e_{(i+2)jk}^{t} + e_{(i+3)jk}^{t} + e_{(i+4)jk}^{t} + \\ & e_{i(j-1)k}^{t} + e_{i(j-2)k}^{t} + e_{i(j-3)k}^{t} + e_{i(j-4)k}^{t} + \\ & e_{i(j+1)k}^{t} + e_{i(j+2)k}^{t} + e_{i(j+3)k}^{t} + e_{i(j+4)k}^{t} + \\ & e_{ij(k-1)}^{t} + e_{ij(k-2)}^{t} + e_{ij(k-3)}^{t} + e_{ij(k-4)}^{t} + \\ & e_{ij(k+1)}^{t} + e_{ij(k+2)}^{t} + e_{ij(k+3)}^{t} + e_{ij(k+4)}^{t}) \end{aligned} \quad (5.2)$$

In regard to the results presented below, the important characteristics of these stencils are the order of the stencil and the amount of necessary floating point operations. The order directly influences the memory regions which need to be accessed when computing the stencil and therefore also the necessary ghost cells (discussed below) and MPI message sizes. The amount of floating point operations for the used stencils is relatively small, making the problem memory bound.

5.1.3. Computational Domain

Application Wide Size and Nomenclature

For the presented approaches and the corresponding results, the computational domain can be defined application wide through the definition of the elements in each of the three dimensions. Throughout this work, the dimensions will be referred to as `x`-dimension, `y`-dimension, and `z`-dimension. The sizes provided to the programs define the number of stencils which need to be computed throughout every iteration. Additional memory regions needed to store boundary values surrounding the computational domain will be added. A two dimensional example computational domain can be seen in Figure 5.1 (bottom-left). In this example, a first order 4 point stencil is depicted for different elements showing the dependencies in space. Due to the fact that a first order stencil uses elements which are at most one element away in each dimension, a single row of boundary elements is necessary, as depicted in Figure 5.1 (top-left).

Memory for Each MPI Eank - Ghost Cells and Data Layout

For the domain decomposition, different options are available. Independent of the question along which of the dimensions the domain will be split up, each available MPI rank will end up with its own subblock of the computational domain. The rank is responsible for storing and computing the values of the elements in its subblock. For those sides of the subblock, which are on the edge of the computational domain, the rank needs to store and initialize the boundary values. For the sides which border on the subblock of another MPI rank, so called ghost cells need to be added. As the update of the elements on these sides need values from neighboring elements which are part of other MPI ranks subblocks, these values need to be received by the corresponding ranks and stored in order to be used. The number of element layers needed for the ghost cells is the same as for the boundary values and defined through the order of the used stencil. For the presented two dimensional example and a set of nine MPI ranks, splitting the example computational domain in both dimensions equally, the assignment of computational domain subblocks to `MPI` ranks is shown in Figure 5.1 (top-right). For nine ranks, the depicted splitting is done equally along both dimensions. The correlation between ghost cells and the corresponding elements in the neighboring MPI rank can be seen in Figure 5.1 (bottom-right).

Each rank allocates two matrices, the size of both including the elements assigned to its subblock and the necessary boundary values and ghost cells. The two copies alternately contain the computed values of step t and are used to store the new values for step $t + 1$. As the memory allocated for each matrix is sequential, it is important to keep in mind the mapping of the three dimensions to the used memory region. In this work, `y`-dimension is the innermost dimension, such that stepping along the `y`-axis equals a sequential step in memory. The `x`-dimension is the middle dimension, such that stepping along the `x`-axis equals a step in memory of the size of the ranks elements in `y`-dimension, including ghost cells and boundary values. Finally, the `z`-dimension is the outer dimension, such that stepping along the `z`-axis corresponds to a step in memory of the size of an `y`-`x`-plane.

Pinning and Memory Initialization

As some of the used HPC systems consist of nodes providing a NUMA environment, as discussed in Section 2.1.3, it is important to take into account where memory resides and on which cores the used MPI ranks and their threads run. For all presented test runs, the MPI ranks and the used threads have been pinned to the available cores on the respective nodes in order to make sure that the work on a given set of elements is performed on the same core as much as possible. Exceptions in case of possible work stealing in cases where the proposed `commtask` approach is being used will be discussed below.
When allocating memory, especially large amounts of memory as needed for the two copies of the computational domain, a widely used practice of operating systems in combination with the hardware is to reserve the virtual memory only and to defer the allocation of physical memory to the time it is accessed for the first time. This `first-touch policy` is very important in regard to NUMA domains and the use of threads [8, 32, 68, 71–73, 94]. For `MPI-Only` applications, pinning the ranks to cores is enough to make sure that the memory accessed by the process is physically located close to the used core because of the way the `first-touch policy` works. When using threads, the best practice is to make sure that the first time a memory location is used is by the (pinned) thread, which will use it mostly. Otherwise, when initializing the allocated memory using the main thread, the `first-touch policy` will try to allocate all memory close to the core this thread is running on.
For all presented implementations, each thread initializes the memory region it is going to work on directly after the memory allocation. Afterwards, the boundary values can be initialized by any thread (in this case the master thread) without changing the placement in the physical memory. The boundary values are initialized as values in the range from 0.0 to 1.0 equally distributed across the outer edges of the computational domain.

Optimizations

In addition to the presented optimizations (i.e. process and thread pinning, memory allocation close to the core where it is mostly going to be used, the minimization of inter-node communication) additional optimizations are still possible and applicable to all versions presented below. They are even necessary, as compilers are not good at optimizing stencil codes [24]. For this work, the main optimization aspect has been accessing the matrix elements sequentially in memory and applying cache blocking inside the ranks/threads computational domain parts. For results comparing the different versions for a given set of computational domains on the same hardware environment and node/core count, the same cache block sizes have been used.
One of the optimizations applied, as described above, is taking into account the `first-touch policy` when using threads (both OpenMP and manual use of Pthreads). The impact of this was observed during the implementation and testing phase of the hybrid MPI-`commtask` approach for the three-dimensional computational domain distribution presented in Section 5.3. In a previous version, the `first-touch` initialization of the computational domain has been done by evenly splitting the used matrices (including the boundary values) to the used threads. During computation phase, the computational domain blocks were computed in regard to the actual elements which had to be computed (omitting the boundary values). At the same time, the ranks computational domain has been distributed along the `x`-dimension to the used threads, as cache blocking has been done along this axis anyway. Due to unsatisfactory performance, the code has been double checked and both issues have been fixed: The computational domain blocks are now being computed exactly the same way in both the initialization and computation phase. Also the threads split the ranks computational domain along the `z`-axis. While they still do cache blocking along the `x`-axis, this avoids the need to access memory close to another thread for nearly all elements. With these changes a performance improvement of up to 30% has been observed for the used test scenarios. Therefore, these changes are part of the implementations presented later in this chapter.

Other optimizations would have included vectorization, which has not been applied in the presented results, as it would have been out of the scope of this work. Nevertheless, vectorization and any other possible optimization which could have been applied would have sped up the computational phase. Therefore the communication phase would be larger in comparison to the computation phase and the impact of being able to overlap communication with computation would increase. None of the ideas behind the `commtask` would prevent optimizations like vectorization or memory padding nor would those optimizations prohibit the use of `commtasks`.

5.1.4. Implementations

Having discussed all aspects which are the same throughout the different versions, the differences and important characteristics of these are going to be presented here. For each of these versions, a close look at the actual steps taken inside the time measured phase will be discussed. This includes the desired repetition of computing all stencils and everything concerning the communication of data for the ghost cells. Starting with the simple `MPI-Only` version with distinct communication and computation phases, the advanced implementations will build up to a version including everything presented in regard to the `commtasks`.

MPI-Only

Every iteration of the program has two separate steps for communication and computation. First the necessary "edge"-values for the ghost cells are communicated, exchanging data from the previous iteration or initialization in the first iteration. In the simpler case, distributing the computational domain along the `z`-dimension only (see Section 5.2), the MPI function `MPI_Sendrecv` is used once for exchanging data along one direction of the `z`-axis and once along the other direction. For the three-dimensional computational domain splitting version, nonblocking MPI functions are used in order to avoid deadlocks and communication serialization. Each rank posts an `MPI_Irecv` for each neighbor before initializing all needed send operations using `MPI_Isend`. These calls are directly followed by a call to `MPI_Waitall` to finish the communication phase before computation starts. Using these functions optimizes the communication phase without overlapping it with the computation phase and therefore avoids the necessary adjustments for it.
For each rank, the computation phase starts once all communication has finished. Since all necessary information for all elements is locally available to the rank, it can compute all of them without the need to distinguish between elements which need information from other ranks and those who do not. During this computation, all optimizations discussed above are applied.

Starting with the `MPI-Only` version, two directions can be taken in direction of the `commtask` approach. Either it can be adjusted to use nonblocking communication functions in order to overlap communication and computation (⇒ `MPI-Only-Nonblocking`), or it can be turned into a hybrid distributed-/shared-memory program by parallelizing each MPI process using OpenMP or Pthreads (⇒ `hybrid`). In a later step these two version can be combined to a hybrid version with overlap (⇒ `MPI-OpenMP-Nonblocking` and `MPI-PthreadClassic--Nonblocking`).

MPI-Only-Nonblocking

Based on the `MPI-Only` implementations, this is the first step in trying to hide the communication behind necessary (useful) computation using functions provided by MPI. Both computational domain distribution versions start the iteration by posting `MPI_Irecv`, and `MPI_Isend` calls for all neighbors in order to initialize the data transfer of values for the ghost cells. Instead of waiting for the communication to finish, the computation in each rank can start on those elements which are independent of data transfers. For these computations, the discussed optimizations are applied the same way as before, making sure that the communication dependent elements are omitted.
After all communication independent work is finished, successful reception of the needed data must be checked using one of the `MPI_Wait` calls (e.g. `MPI_Waitall` for the receive operations). Only then can each rank start computing the stencils for the communication dependent edge elements. Once this is finished, if not done so with the `MPI_Waitall` call before, the successful termination of the send operations must be checked. For the presented implementations, a combined `MPI_Waitall` has been used.

MPI-OpenMP

Using `MPI-Only` on hybrid hardware, such as the different HPC systems used in this work, results in two different kinds of communication: intra-node communication and inter-node communication. As each MPI rank is pinned to one dedicated core, the communication partners can be located either on a different core of the same compute node or on a different compute node in the system, resulting in intra-node or inter-node communication, respectively.
The MPI implementations can optimize the communication by bypassing the networking interface when intra-node communication is detected [29]. This optimization has been the subject of research in the field of high performance computing [50, 56]. Nevertheless, it might not be included in all MPI implementations [79]. As the affected ranks are individual processes, they do not share the same memory space and, even for well optimized MPI implementations, at least one copy operation is necessary to move the data from the sending ranks memory to the receiver. By assigning multiple cores to each MPI rank, assigning larger computational domain parts to each rank and parallelizing the work inside each rank using OpenMP (or any other shared memory approach), the intra-node communication can be omitted. Each thread and the core it is pinned to can work on the same data the corresponding `MPI-Only` rank worked on and has direct access to the data provided by other threads on the same node.

Concerning the computation, the simple approach of using the `OpenMP for` construct turned out to be inefficient. While OpenMP creates the necessary threads only once at the first time they are needed, keeping them idle for the next `OpenMP parallel` region, the work distribution and overall performance slowed down application performance drastically in all tested configurations. Therefore, instead of using the `OpenMP for` construct inside the `OpenMP parallel` region, the distribution was created manually based on the thread identifier (`tid`). For all implemented `MPI-OpenMP` versions, this turned out to be the best approach. The main thread is responsible for all MPI related operations which would be the same in case the `OpenMP for` construct would be used. As discussed above, thread pinning and initialization of the memory according to the `first-touch policy` has been applied and `MPI_Init_thread` was used in order to confirm `MPI_THREAD_FUNNELED` support.

Finally two options for assigning cores to MPI ranks have been considered. The option used for the results presented in this work, in case not stated otherwise, is one rank per node with one thread per available core. Instead of assigning an entire compute note to each rank, the second option is to use NUMA domain sockets. For each NUMA domain socket, one MPI rank is created. Internally the rank places one thread on each of the cores inside the socket. While this introduces intra-node communication between the sockets, the threads run in a UMA domain like environment. This option has been tested for the one dimensional domain decomposition approach presented in Section 5.2, but was not applied in detail, as the results were nearly identical to using one node per rank. For the results presented in Section 5.3, the second option has been applied and results will be presented.

The main differences to an `MPI-Only` version are the direct memory access inside the processes memory region instead of the intra-node communication and the fact that, for the corresponding elements, no ghost cell memory has to be allocated.
In the one dimensional domain decomposition versions, the mapping of elements to be computed to the available cores is the same for both `MPI-Only` and `MPI-OpenMP`. Observed differences in timing results can therefore be directly credited to the communication part of the application. For the three dimensional domain decomposition approach, the element to rank and thread, respectively, has been optimized individually. This allows for the different approaches to incorporate all of its optimization potential.

MPI-OpenMP-Nonblocking

In order to implement the `MPI-OpenMP-Nonblocking` version of the algorithm, all aspects of the `MPI-OpenMP` and `MPI-Only-Nonblocking` versions have been considered. Work items have been distributed to the available threads using their `tid`. The main thread takes care of the communication initialization before all threads work on the communication independent work. After the main thread confirms successful communication completion using `MPI_Waitall`, all threads share the communication dependent work.

MPI-PthreadClassic-Nonblocking

Implementing the same work distribution as the MPI-OpenMP-Nonblocking version, this version is implemented using classic Pthreads instead of OpenMP for the shared memory parallelization. The first difference is that the thread management is not done inside a library such as OpenMP. All threads are active during all Jacobi steps, i.e. they are created before and destroyed after the measurement points. As the used HPC systems are dedicated to one user job at a time, having active idle threads on the available cores is efficient, and it is not necessary to suspend them as no other processes need the resources at the same time.
In addition, the threads are used to parallelize the necessary MPI function calls. Depending on the number of neighbors an MPI rank has to communicate with, multiple threads can for example be used to call MPI_Isend and MPI_Irecv for the different messages. As this is being done concurrently, the thread level which has to be tested is not MPI_THREAD_FUNNELED, as in the MPI-OpenMP and MPI--OpenMP-Nonblocking versions, but MPI_THREAD_MULTIPLE. In the same way as the MPI-OpenMP-Nonblocking version, this version does not overlap communication code phases with computation code phases, i.e. all threads synchronize between the calls to the communication initialization functions and the computation as well as after the computation phase, etc. This does not mean that MPI cannot overlap the actual communication with computation, as is desired when implementing the algorithm this way.
The other aspects such as pinning and memory initialization regarding the first-touch policy, have been applied.

As this version has been implemented only for the three-dimensional computational domain decomposition, the distribution of matrix elements to threads has been optimized and is different from the element to core mapping in the corresponding MPI-Only versions. Testing different distributions, it turned out that splitting the ranks computational domain block along the z-dimension and applying cacheblocking along the y-axis and x-axis is most efficient. This reduces the spatial and temporal access to other threads' memory along the splitting borders while giving each thread a memory region to work on which is entirely sequential in memory. Other tested distributions were splitting along the x-axis as cache blocking along this axis is done anyway and giving each thread the entire z-dimension to work along. Another approach was splitting the ranks computational domain along multiple axes in the same way as done on the higher level for the MPI ranks. Both of the latter versions were inferior in performance to the used approach.

Commtask

One possible way of implementing the aspects concerning the proposed `commtask` (see Chapter 4) would be to extend an existing OpenMP implementation. Nevertheless, the open source OpenMP versions do not use a common work pool for their worksharing constructs. One of the possible implementations to choose is the `GOMP`[1] OpenMP implementation for the `GNU Compiler Collection`[2]. The task scheduling code is part of the thread barrier synchronization code, as this is a point where all tasks have been created and are available to the OpenMP runtime. At the same time, the work scheduling for the `OpenMP for` construct is done separately, such that the execution of index-sets assigned in a parallelized for loop will never be mixed with the execution of scheduled tasks.
As the goal of the presented implementations is to compare the performance of the approaches and not the ease with which it can be implemented in a given OpenMP compiler and runtime, a manual implementation has been chosen. Therefore the presented implementation uses Pthreads in such a way that they represent the behavior of a possible OpenMP implementation using Pthreads internally, as done in `GOMP`[1].
Implementing a library providing work queues, which can be accessed in parallel by multiple threads, each thread creates one work queue for the communication independent work. This queue is being filled with the index blocks defined through the cache blocking (`x`-dimension and `y`-dimension) and a new block size determining the amount of work done before a thread should check communication dependent steps (`z`-dimension). In accordance with the presented `commtask` approach, this represents a high level `OpenMP for` schedule `static`, splitting the communication independent work to the used threads. On a lower level the individual threads split their work internally into blocks corresponding to a `dynamic` schedule. This corresponds to the presented `OpenMP` schedule `static-ws`, as presented in Chapter 3. Additionally the main thread creates work queues for the communication dependent work, as defined through the `commtask` as presented in Section 4.4.
The steps taken by the individual threads, which are repeated until all work has been done, are shown in Algorithm 5.1.

The Pthread implementation is representative for an OpenMP implementation supporting the `commtask` approach being transformed into code using Pthreads. The work to core mapping is the same as in the `MPI-OpenMP` versions as well as the `MPI-Only` versions with the expected differences: The communication dependent work is assigned to those threads which have the resources for them first and as soon as the communication has been finished. Other threads keep on working on the communication independent work. In the final steps, threads finishing their own work early, due to less overhead through communication de-

[1] `https://gcc.gnu.org/projects/gomp/`
[2] `https://gcc.gnu.org/`

Algorithm 5.1 Work selection for threads using the `commtask` approach

```
 1: if !(Communication finished) then
 2:     if !(Executed by other thread) then
 3:         Check communication with MPI_Test
 4:     end if
 5: end if
 6: if Communication finished then
 7:     while Communication dependent work exists do
 8:         Take index block from respective work queue
 9:         Compute stencils for index block
10:     end while
11: end if
12: if Thread local work exists then
13:     Take index block from own work queue
14:     Compute stencils for index block
15: else
16:     if Work exists for different thread then
17:         Take index block from respective work queue
18:         Compute stencils for index block
19:     end if
20: end if
```

pendent work, use their resources to reduce the workload of threads which have participated in communication dependent steps by using the added workstealing.

For two different domain decomposition approaches, namely 1D-Decomposition and 3D-Decomposition, the implementations have been executed for different combinations of nodes, cores per node, computational domain sizes, etc. The results will be presented in the following Section 5.2 and Section 5.3, respectively.

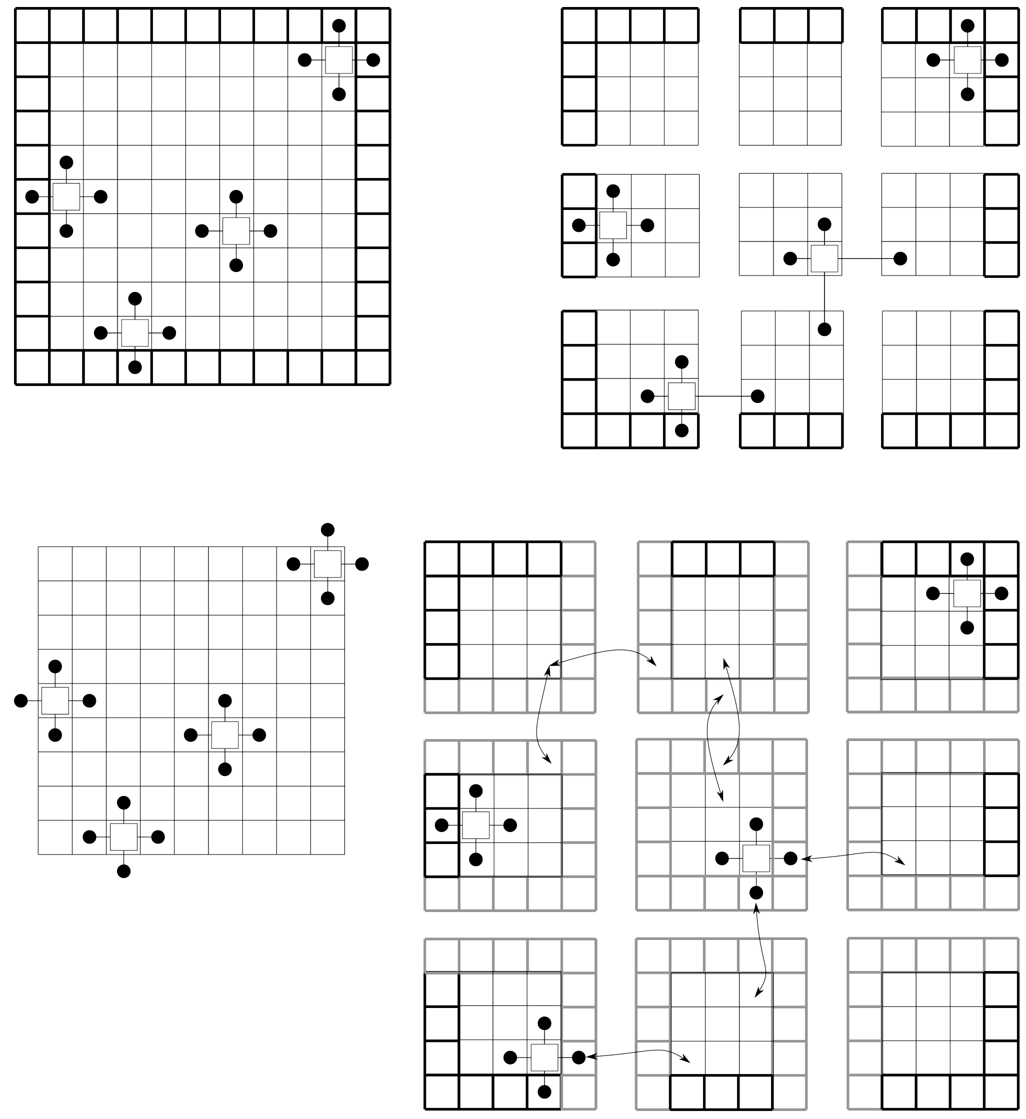


Figure 5.1.: Example computational domain. bottom-left: 9x9 element computational domain with first-order 4 point stencil examples. top-left: Computational domain with memory region for boundary values. One additional element in each direction due to the first order stencil used. top-right: Decomposition to 9 ranks splitting evenly along both dimensions. bottom-right: Decompositied computational domain with ghost cells and representative examples of corresponding memory regions.

5.2. One-dimensional Decomposition

5.2.1. Computational Domain Decomposition

The algorithm and the different implementations presented in the previous section are targeted to use many computational cores spread out across multiple hardware nodes in a parallel computing environment, especially high performance computers. The combined hardware, including (but not limited to) cores, memory and network, is used to work on a single problem instance defined through the three dimensional computational domain. The only aspect concerning the presented implementations not discussed above is the question of how this computational domain is being distributed to the used MPI processes. In this section, the three dimensional computational domain is being split along one axis only. This is the easiest approach concerning the complexity of the implementation for programmers. The choice of axis is relatively straightforward. The results in Section 4.2.6 showed that using non-contiguous buffers introduces high overhead and seems to negatively influence the overlap capabilities of the proposed `commtask`. The layout of the computational domain in combination with the programming language `C` results in the fact that multiple planes made of the `x`- and `y`-dimension are located sequentially in memory. Splitting along the `z`-axis, assigning computational domain blocks to each MPI rank with communication partners only in the `z`-dimension creates dependencies in such a way that the necessary communication buffers are sequential in memory. All but the two extreme ranks (0 and `MPI_Comm_size` − 1) communicate with the two adjacent ranks only. The two exceptions have a single communication partner. Splitting in the other two dimensions would either result in using "stripes" from the computational domain matrix (i.e. splitting along the `x`-dimension) or using single elements (i.e. splitting along the `y`-dimension). For both, distributed and shared memory communication, this would results in overhead. For MPI messages, the distributed data would have to be either joined into a contiguous memory buffer or sent in multiple messages (or a combination of both). For shared memory access to the corresponding regions, the non sequential planes would result in more accesses to different cache lines. Nevertheless, as will be discussed and shown in Section 5.3, a domain decomposition across multiple axes can be advantageous as the overall relation of communication dependent to communication independent elements is reduced. For the one dimensional splitting, no advantage can be found or observed by splitting along the `x`-dimension or `y`-dimension.

The implementations used for the presented results in this section use the `1n-Stencil` as presented in Section 5.1.2. As this is a first order stencil, each rank needs one `x`-`y`-plane from each neighbor to compute the border elements in this neighbor's direction. At the same time, each rank has to send one plane to each neighbor in each iteration. The message size for these runs is therefore directly defined through the size of the x- and y-dimensions. The implementations use `double` as the data type to store each element. The used systems have a size of 8-bytes per double. The size S for each necessary message M for a computational domain of dimensions x,y,z is therefore:

$$S_M = x * y * 8\ Byte \tag{5.3}$$

In order to optimize the communication patterns on the used system, the ranks are placed by filling up a node before placing the next rank on a new node. Therefore the inter-node communication is limited to the messages between those ranks with a direct neighbor is located on a adjacent node. The communication pattern for inter-node communication does not change between the different versions. For all hybrid approaches, the rank placed on each node is responsible for the computational domain part which is made of all parts assigned to the same node in the `MPI-Only` version. As the splitting is along the z-axis only, the messages even contain the same part of the computational domain with the same size. The difference is in the cores and the respective processes and threads which are communicating. While, for the `MPI-Only` version, the communication always happens between fixed ranks pinned to a fixed core, the `MPI-OpenMP` versions always have the master thread and its dedicated core doing all MPI related work. The `commtask` approach dynamically assigns this work to any available threads.
The intra-node communication of the `MPI-Only` implementations is replaced by shared memory accesses in all hybrid versions as the hybrid versions presented here all use one MPI rank per available node with one thread for each available core.

Computational Domain Sizes

For the presented results, two aspects have been looked at in regard to the chosen computational domain sizes. As discussed above, the message sizes are directly dependent on the chosen x- and y-dimensions. This is also directly related to the amount of time which can be saved when achieving real communication overlap, i.e. synchronization and data transfer overlap. The presented result graphs show stencil updates per second (higher is better) for different message sizes (larger messages on the right). The message sizes are the result of different chosen dimensions for the x- and y-axes. The dimensions range from small x- and y-dimensions(e.g. $x = 500$ and $y = 300$ on the Woodcrest Cluster) to large ones (e.g. $x = 2000$ and $y = 2000$ on the SuperMUC Fat Nodes).
The second aspect is the amount of work needed to be performed on each available core and therefore the amount of communication independent work which can be used to hide all communication related steps. This is directly related to the z-dimension of the computational domain. In order to have comparable results throughout the different used clusters, the z-dimension has been defined

through the amount of cores available. Defining the `planes per core` as *ppc*, every `x`-`y`-combination has been combined with the different values $ppc = 5$, $ppc = 10$, $ppc = 20$, ..., $ppc = 50$. This results in different sized `z`-dimensions (e.g. $z = 1200$, $z = 2400$, $z = 4800$, $z = 7200$, $z = 9600$ and $z = 12000$ for each `x`-`y`-combination used on the LiMa Cluster).

5.2.2. Discussion of Results

	Woodcrest (64*4 cores)	**ICE** (16*4 cores)	**SuperMUC Fatnodes** (5*40 cores)	**LiMa** (20*12 cores)
`MPI-Only`	5.2	5.17	5.22	5.28
`MPI-Only-Nonblocking`	5.3	5.18	5.23	
`MPI-Only-Nonblocking` with `MPI_Test`	5.4			
`MPI-OpenMP`	5.5		5.24	5.29
`MPI-OpenMP-Nonblocking`	5.6	5.19	5.25	5.30
`MPI-OpenMP` with `MPI_Test`	5.7	5.20	5.26	
`commtask`	5.8	5.21	5.27	5.31

Table 5.1.: Overview of result graphs

An overview of the different graphs for the different implementations on the used HPC systems can be found in Table 5.1. For all presented results, the communication phase in the `MPI-Only` version is shorter than the computation phase in each iteration. Therefore it is theoretically possible to hide the entire communication behind the computation.
All in all, the presented results show that the impact of communication is visible and all but the `commtask` approaches are not able to overlap communication with useful computation. The results show two different general tendencies: For the larger and newer multi-socket systems in the SuperMUC Fat Nodes and the LiMa Cluster, the use of `hybrid` parallelization results in a clear advantage over the `MPI-Only` implementations. Nevertheless, neither adding nonblocking communication to the `MPI-Only` nor the `MPI-OpenMP` version results in the desired overlap. Using the proposed `commtask` approach, the stencil update per second rate is both at least as high as in any other result for the same system as well as independent from the message size, as the entire communication can be hidden in all cases.
For the smaller and older systems, ICE and Woodcrest, the combined use of `OpenMP` with `MPI` results in a decrease of the observed stencil updates per second rate. Adding nonblocking communication to either the `MPI-Only` or the

MPI-OpenMP again does not result in the desired overlap. Nevertheless, manually adding MPI_Test calls to the MPI-OpenMP-Nonblocking version on the Woodcrest Cluster results in partial overlap as can be seen when comparing Figure 5.6 and Figure 5.7. Nevertheless this works only for small messages. Using the proposed commtask approach, the same high and message size independent stencil update rates can be observed as on the other two system.

Woodcrest Cluster

Using 64 nodes of the Woodcrest Cluster providing 4 cores each, a total of 256 cores were used for the presented results. Information about the Woodcrest Cluster can be found in Appendix A.6.
The base of the comparison is the MPI-Only implementation using blocking communication functions as described above. Comparing these measurements in Figure 5.2 with the results of the MPI-Only-Nonblocking implementation in Figure 5.3, no real difference can be seen. As expected, the the use of nonblocking communication function does not result in any speedup.

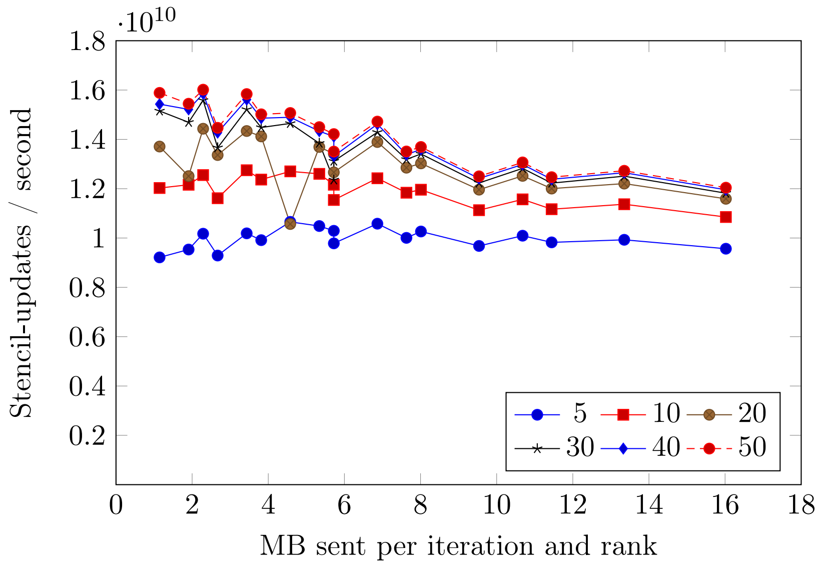


Figure 5.2.: 3D-Jacobi with a one dimensional domain decomposition. MPI-Only on 64 nodes of the Woodcrest Cluster.

One of the initial tests done was also placing MPI_Test calls manually inside the communication independent work of the MPI-Only-Nonblocking version. As the benchmark presented in the previous Chapter 4 showed that for all tested MPI implementations a relatively small number of calls to MPI_Test can result in overlap in an easy setup, the question is whether or not this works in real life applications as well. Looking at the test results in Figure 5.3, it can clearly be

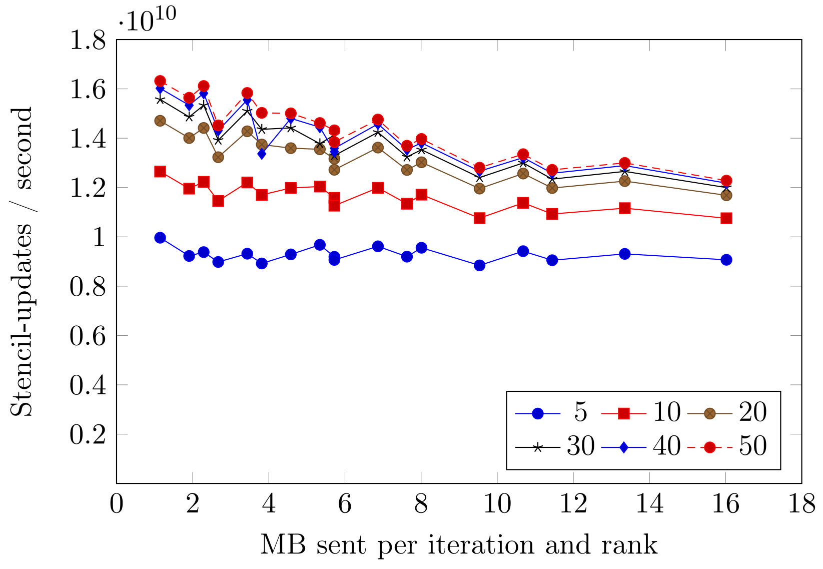


Figure 5.3.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-Only-Nonblocking` on 64 nodes of the Woodcrest Cluster.

seen that this is not the case.

Turning to the hybrid `MPI-OpenMP` approach of combining `MPI` with `OpenMP` using blocking communication functions, it can be seen (Figure 5.5) that for the environment in the Woodcrest Cluster, no advantage compared to the `MPI-Only` approach can be observed. On the contrary, a decrease in performance is visible. While it is small for the computational domain setups with larger message sizes, the decrease is dominant for setups with little work and small messages.

Adding nonblocking communication to the `MPI-OpenMP` approach does also not result in the desired overlap, as shown in Figure 5.6.

The first increase in performance for this cluster can be observed when manually placing `MPI_Test` calls into the `MPI-OpenMP-Nonblocking` implementations. This version achieves a better stencil updates per second performance by achieving communication overlap as can be seen in Figure 5.7. The performance improvement is larger for computational domain setups with little work and small messages. The impact decreases with the increase of the message sizes and is smallest for those computational domains with large messages and larger values for *ppc*.
Reasons for the missing performance gains can be a combination of 1) the overhead of adding the `MPI_Test` calls, 2) a not optimal amount and timing of the added `MPI_Test` calls, and 3) resulting imbalances between the used threads.

While the `MPI-Only` and `MPI-OpenMP` versions using blocking communication behave as expected, the use of nonblocking communication and even the manual addition of `MPI_Test` calls does not result in the desired overlap of communica-

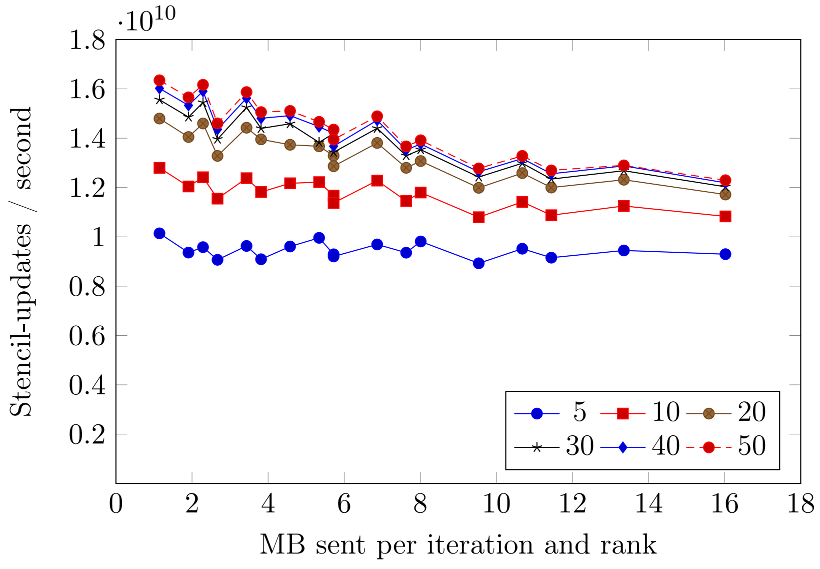


Figure 5.4.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-Only-Nonblocking` calling `MPI_Test` regularely on 64 nodes of the Woodcrest Cluster.

tion with the communication independent work. As all the chosen computational domain setups result in times for the communication phases which are shorter than the respective computation phases, a total overlap would be the optimal outcome. Applying the proposed `commtask` approach to the same computational domain setups on the same hardware environment shows stencil updates per second rates which are very close to this, as can be seen in Figure 5.8. While the results for very small message sizes are worse than the `MPI-OpenMP-Nonblocking` version with `MPI_Test` calls, the stencil updates per second rates for larger messages are high, independent of the message size. While the overhead of assigning the communication related work dynamically to the used threads and adding work stealing to the implementation can be seen when little has to be done in regard to communication, the overhead is negligible in comparison to the improvements gained for cases with long communication phases.

Before looking at the results obtained on the other HPC systems, which have been summarized above, in more detail, a closer look at the results from the Woodcrest Cluster shows that the observed performance improvement is actually a result from the goals of using `commtasks`. Using the Intel Trace Analyzer and Collector[3] (ITAC), a test run has been executed on three nodes of the Woodcrest Cluster. The used computational domain has the dimensions of $(x, y, z) = (500, 500, 600)$. Five Jacobi iterations have been executed and recorded together with the necessary initialization and finalization phases of

[3]`https://software.intel.com/en-us/intel-trace-analyzer`

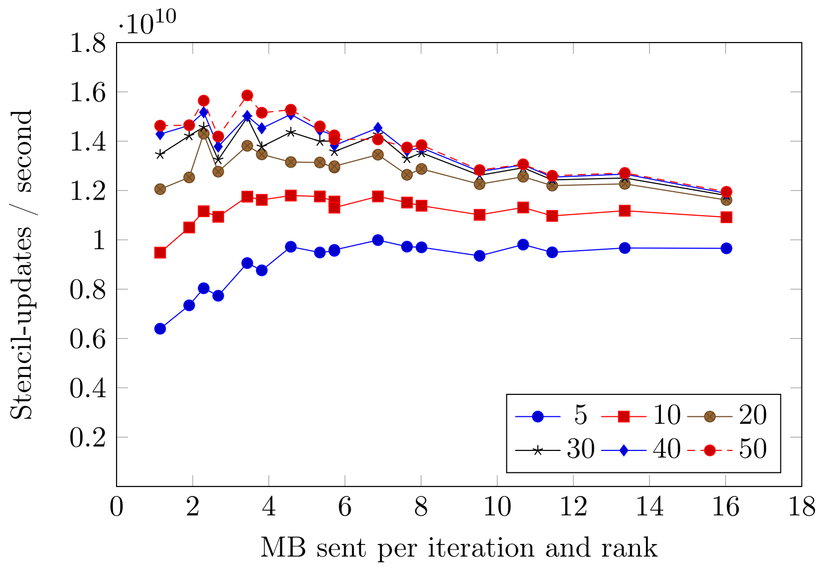


Figure 5.5.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-OpenMP` on 64 nodes of the Woodcrest Cluster.

the implementations. As the Woodcrest Cluster provides four cores per node, the `MPI-Only` version starts 12 ranks, placing ranks $P0 - P3$, $P4 - P7$ and $P8 - P11$ on the three nodes, respectively. For the `MPI-OpenMP-Nonblocking` and `commtask` versions, three ranks $(P0, P1, P2)$ are placed on one node each, starting one thread per available core $(PXT0, ..., PXT3)$. The `MB sent per iteration and rank` are 3.8MB and the value for the used planes per core is $ppn = 50$. As ITAC on Woodcrest does not record the OpenMP threads used by the `MPI-OpenMP-Nonblocking` version, only the traces for the three ranks can be seen in the respective figures. The existence, and correct use of the threads has been confirmed.

The traces for the entire program execution for `MPI-Only` (Figure 5.9), `MPI--OpenMP-Nonblocking` (Figure 5.10) and `commtask` (Figure 5.11) show the five iterations with the necessary communication phases. While seeing the clearly defined communication phases for the blocking communication in the `MPI-Only` implementation is as expected, the visible communication phases for the `MPI--OpenMP-Nonblocking` version show that communication is indeed not being overlapped. Taking a closer look at the traces of one iteration (Figure 5.12) confirms the observation of the benchmarks in the previous chapter: The main part of the communication is done during the call to `MPI_Wait`.
In comparison to this, the traces for the `commtask` implementation show a shorter runtime for the entire execution. The communication phases, while still visible through the lines representing the individual messages, are not visible as time

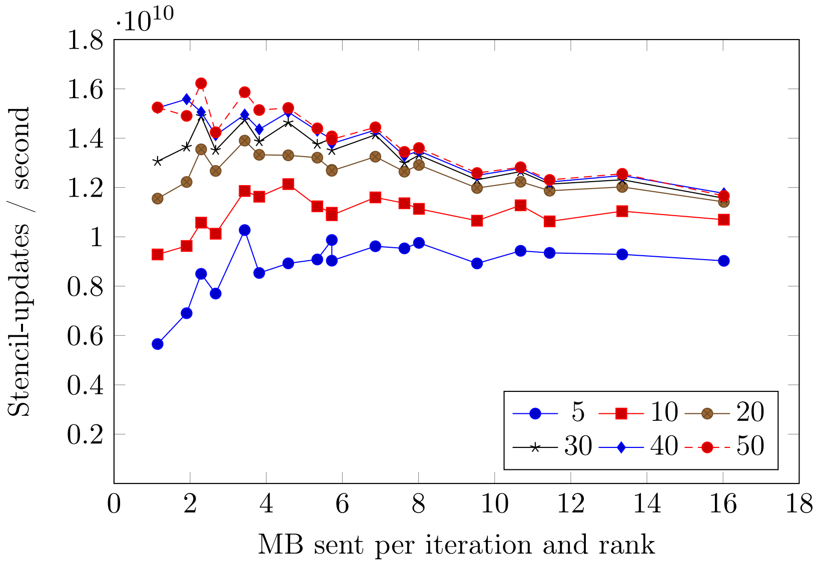


Figure 5.6.: 3D-Jacobi with a one dimensional domain decomposition. MPI--OpenMP-Nonblocking on 64 nodes of the Woodcrest Cluster.

spent in MPI functions. A closer look at one iteration (Figure 5.13) shows that while multiple calls to MPI_Test are done by each thread, these are negligibly short. Also, the communication partners are no longer fixed, but the messages are sent dynamically between two threads of the respective neighboring ranks. These traces show that the communication related steps are no longer executed during calls to MPI library functions and suggest that actual overlap of communication with useful computation has been achieved. Nevertheless, it is not clear what the individual cores spend their time on during the application part of the traces, i.e. outside the MPI functions. In order to show that the observed performance improvement is actually a result of the presented commtask approach and the resulting overlap and work balancing inside the nodes, additional measurements of the same test scenario have been done using the software LIKWID [88]. Using the same node/core count and computational domain, ten iterations have been executed for these measurements. Using likwid-perfCtr, the tool has been used to measure how many double precision floating point operations have been executed on each core of the "middle" node, hosting ranks $P4 - P7$ in the MPI-Only and rank $P1$ in the MPI-OpenMP-Nonblocking and commtask cases. As these operations are the useful work concerning the Jacobi steps, this shows how efficiently the cores are used in regard to the actual goal. Again, the communication phases are clearly visible for the MPI-Only version in Figure 5.14. The MFlop/s (mega floating point operations per second) drop to zero for all cores during the communication phases. While not as pronounced, the same holds true for the MPI-OpenMP-Nonblocking measurements in Figure 5.15. As

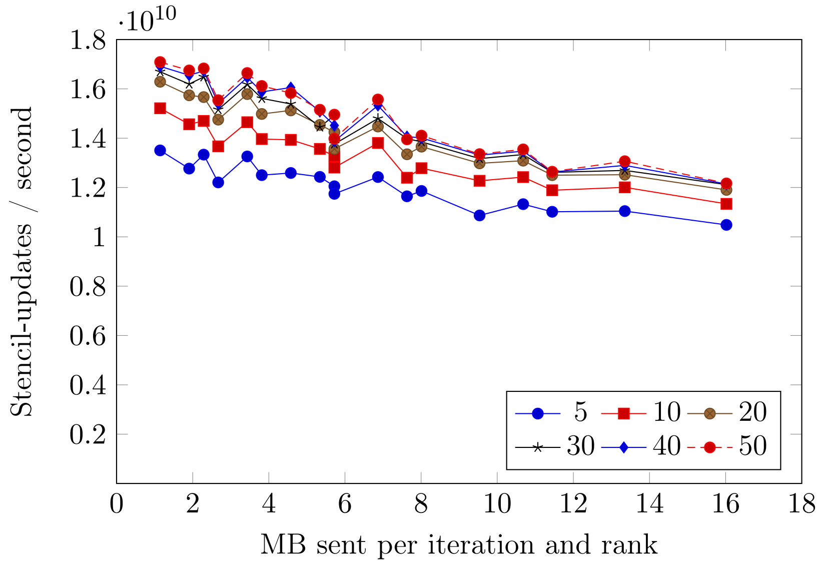


Figure 5.7.: 3D-Jacobi with a one dimensional domain decomposition. MPI--OpenMP-Nonblocking calling MPI_Test regularely on 64 nodes of the Woodcrest Cluster.

shown above, the calls to MPI_Wait are indicated here. Looking at Figure 5.16, the advantage of the commtask approach can clearly be seen. While the iterations are still recognizable by slight drops in the MFlop/s rate for individual cores, all cores are executing a high and constant number of double precision floating point operations throughout the entire program execution. While the slight drops indicate the calls to MPI_Test, it is shown that the resources of the used system are dedicated to useful work and the necessary, but not directly useful, communication can be hidden efficiently.

ICE Cluster

Using 16 nodes of the second single socket system, the ICE Cluster, a total of 128 cores are available for the results presented here. Showing a behavior similar to the results discussed for the Woodcrest Cluster, a decreasing performance for setups with larger messages can be seen for the MPI-Only implementation in Figure 5.17. No difference can be observed when separating communication dependent from independent work and using nonblocking communication functions in the MPI-Only-Nonblocking implementation (Figure 5.18). As mentioned above, using a classic hybrid approach does result in performance penalties for this system. As the measurements for MPI-OpenMP-Nonblocking in Figure 5.19 show, the penalties for this are quite high. Manually adding MPI_Test calls decreases the performance further (Figure 5.20). Nevertheless, using a combined shared- and distributed memory parallelization as proposed in this work can overcome

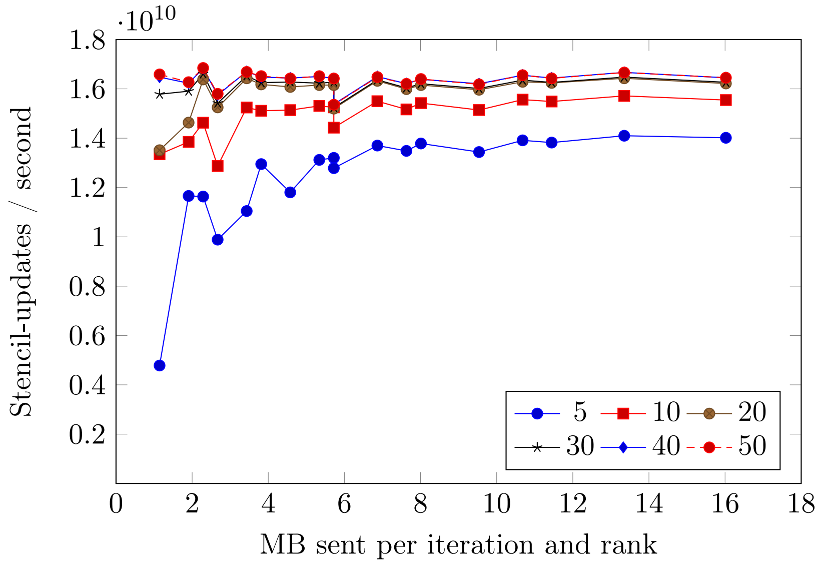


Figure 5.8.: 3D-Jacobi with a one dimensional domain decomposition. `commtask` on 64 nodes of the Woodcrest Cluster.

these drawbacks and make optimal use of the provided hardware environment as can be seen when looking at the `commtask` results in Figure 5.21. As with the results for the Woodcrest Cluster, the stencil update per second rate is high independent from the necessary communication load. The same aspects as discussed above apply.

SuperMUC Fat Nodes

In contrast to the two systems discussed above, the SuperMUC Fat Nodes are providing a multi-socket non uniform memory access (NUMA) environment. Each node has four sockets with ten cores each. For the `MPI-Only` versions, this results in a large number of rank pairs having to do intra-node communication as well as a lot of concurrent access to the same physical memory. The presented results were configured to use 200 cores on a total of five nodes of the system. For this setup a lot more intra-node communication partners exist as rank pairs which do actual inter-node communication. The `MPI-Only` implementation does not perform very well, as can be seen in Figure 5.22. The results get even slightly worse for the `MPI-Only-Nonblocking` implementation (s. Figure 5.23).
The main difference from the results presented for the two single socket systems (i.e. ICE Cluster and Woodcrest Cluster) is the impact of using hybrid parallelization even in the classic way. Just going from `MPI-Only` to `MPI-OpenMP`, both using blocking communication, increases the performance drastically. Especially for computational domain configurations with smaller message sizes, the

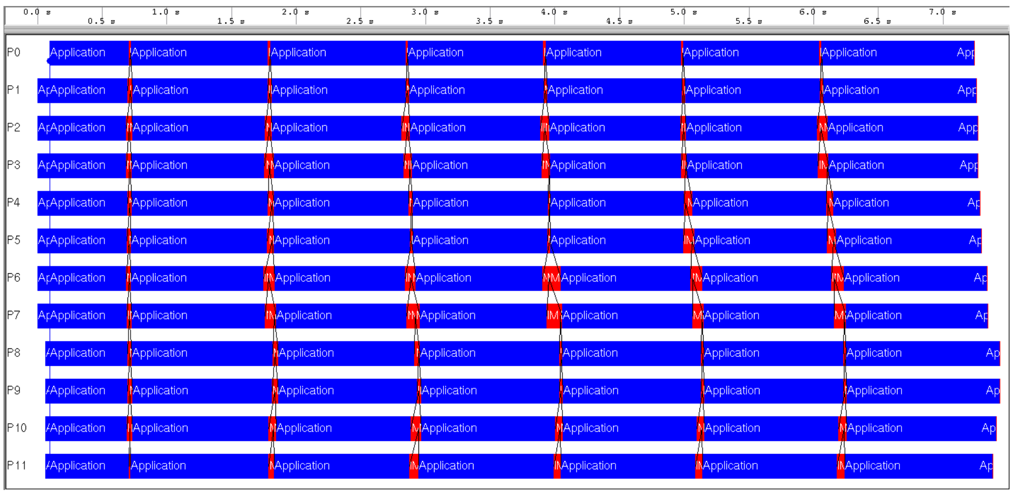


Figure 5.9.: ITAC traces: `MPI-Only` version of a test setup using three nodes on Woodcrest Cluster. 5 iterations for a computational domain with dimensions (x,y,z) = (500,500,600).

hardware resources can be used a lot more efficiently. Nevertheless, the impact of larger messages for each rank in each iteration can be seen in the corresponding results in Figure 5.24. Again, making use of nonblocking communication functions when splitting communication independent and dependent work does not make a difference (Figure 5.25). While it cannot be expected that adding `MPI_Test` calls manually to the `MPI-OpenMP-Nonblocking` version results in performance improvements on this system due to the results presented in Chapter 4, the dramatic performance loss seen in Figure 5.26 was not expected. No strategy for placing the `MPI_Test` calls (i.e. different loops in the computation of the communication independent work) improved this.
Reducing the overhead of message progression by dynamically assigning the work to the available threads in the `commtask` implementation again turned out to work very well. Work balancing together with the achieved communication overlap resulted in the stencil updates per second rate shown in Figure 5.27. Independent of the message size, the performance is as high as in the best combination for the `MPI-OpenMP` implementations. Especially for large messages, a lot of time was needed for the communication which now can be hidden behind useful work perfectly.

LiMa Cluster

The LiMa Cluster provides dual socket NUMA environments with 12 cores per node. Using 20 nodes of the system, the presented results base on the use of 240 cores. Less cores per node than on the SuperMUC Fat Nodes seem to be the reason why the `MPI-Only` version does show pretty good performance as can be

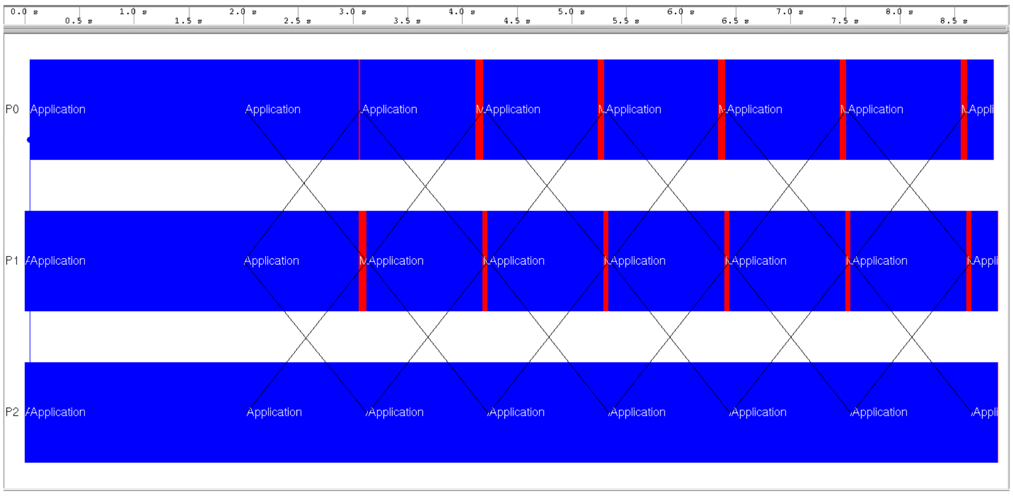


Figure 5.10.: ITAC traces: `MPI-OpenMP-Nonblocking` version of a test setup using three nodes on Woodcrest Cluster. 5 iterations for a computational domain with dimensions (x,y,z) = (500,500,600).

seen in Figure 5.28. Nevertheless a similar performance increase can be observed when switching from `MPI-Only` to hybrid `MPI-OpenMP`(Figure 5.29). Especially for computational domains with less communication independent work, the hybrid approach increases performance. Adding nonblocking communication does not make a difference (Figure 5.28), again analogous to the results presented for the SuperMUC Fat Nodes. Applying the proposed `commtask` results in the desired efficient use of the system, as shown in Figure 5.31.

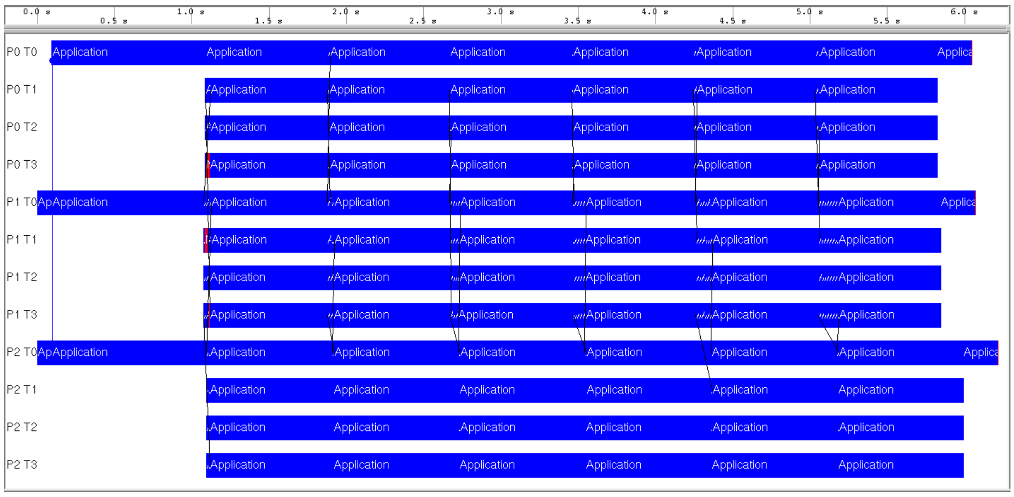


Figure 5.11.: ITAC traces: `commtask` version of a test setup using three nodes on Woodcrest Cluster. 5 iterations for a computational domain with dimensions (x,y,z) = (500,500,600).

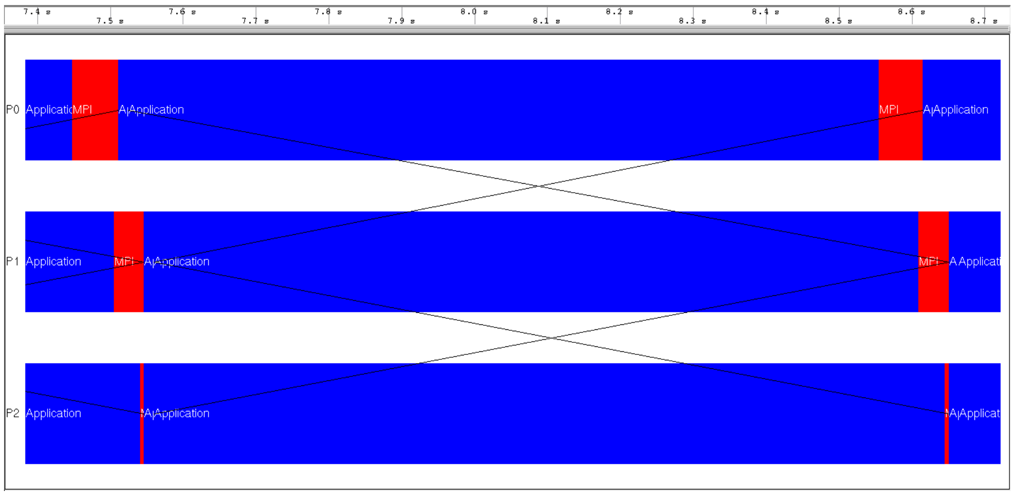


Figure 5.12.: ITAC traces: `MPI-OpenMP-Nonblocking` version of a test setup using three nodes on Woodcrest Cluster. Detailed view of one computation phase surrounded by two communication phases for a computational domain with dimensions (x,y,z) = (500,500,600).

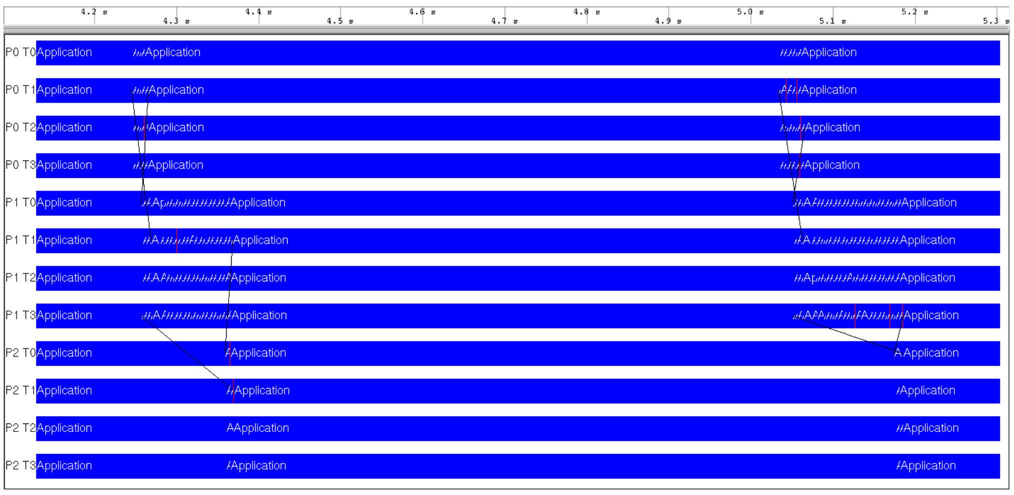


Figure 5.13.: ITAC traces: `commtask` version of a test setup using three nodes on Woodcrest Cluster. 5 iterations for a computational domain with dimensions (x,y,z) = (500,500,600).

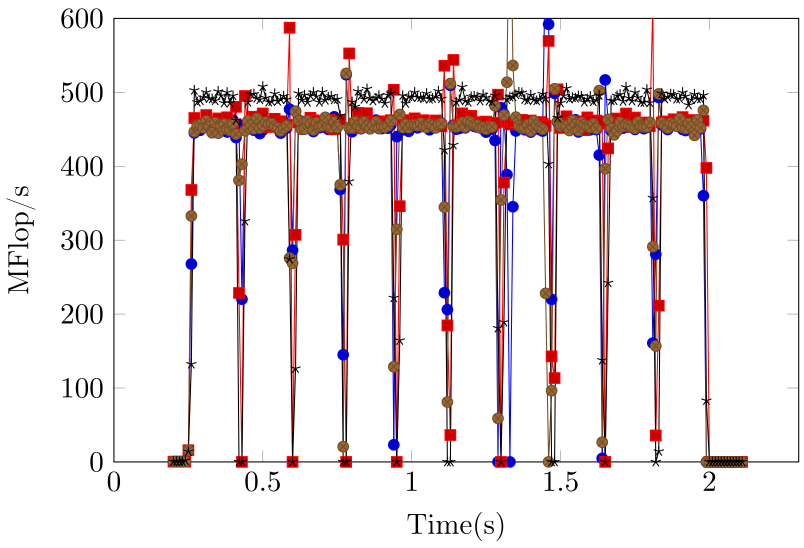


Figure 5.14.: LIKWID double precision floating point per second (FLOAT_DP) measurements using three nodes of the Woodcrest Cluster: `MPI-Only`. 10 iterations for a computational domain of the dimensions (x,y,z)=(500,500,600). The results show the measurements of the middle node hosting ranks 4 to 7.

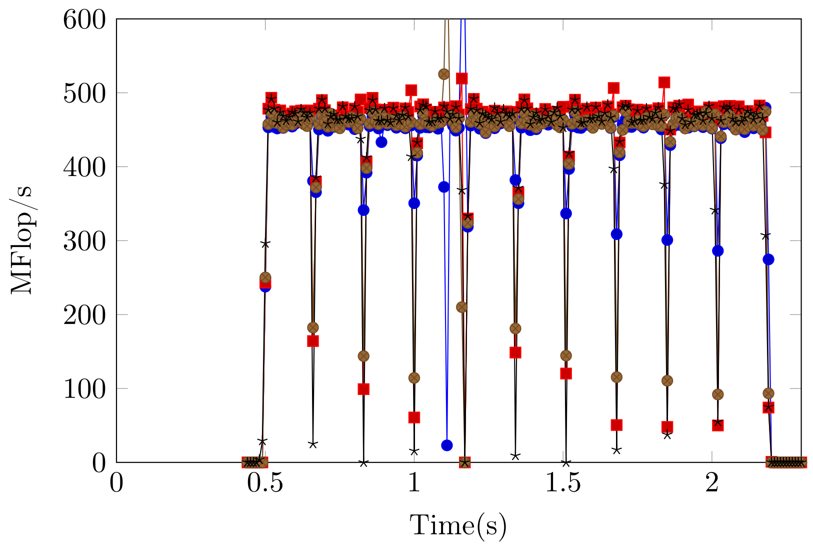


Figure 5.15.: LIKWID double precision floating point per second (FLOAT_DP) measurements using three nodes of the Woodcrest Cluster: `MPI--OpenMP-Nonblocking`. 10 iterations for a computational domain of the dimensions (x,y,z)=(500,500,600). The results show the measurements of the middle node hosting rank 1 using 4 `OpenMP` threads.

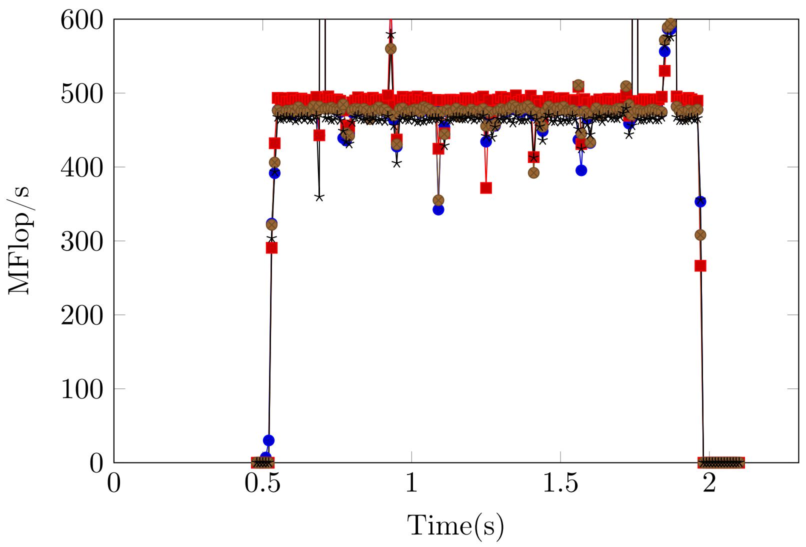


Figure 5.16.: LIKWID double precision floating point per second (FLOAT_DP) measurements using three nodes of the Woodcrest Cluster: `commtask`. 10 iterations for a computational domain of the dimensions (x,y,z)=(500,500,600). The results show the measurements of the middle node hosting rank 1 using 4 threads.

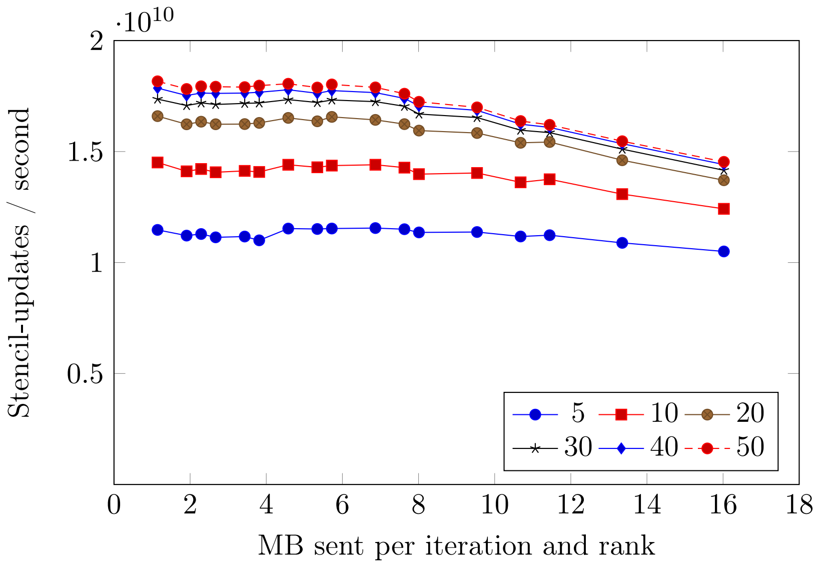


Figure 5.17.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-Only` on 16 nodes of the ICE Cluster.

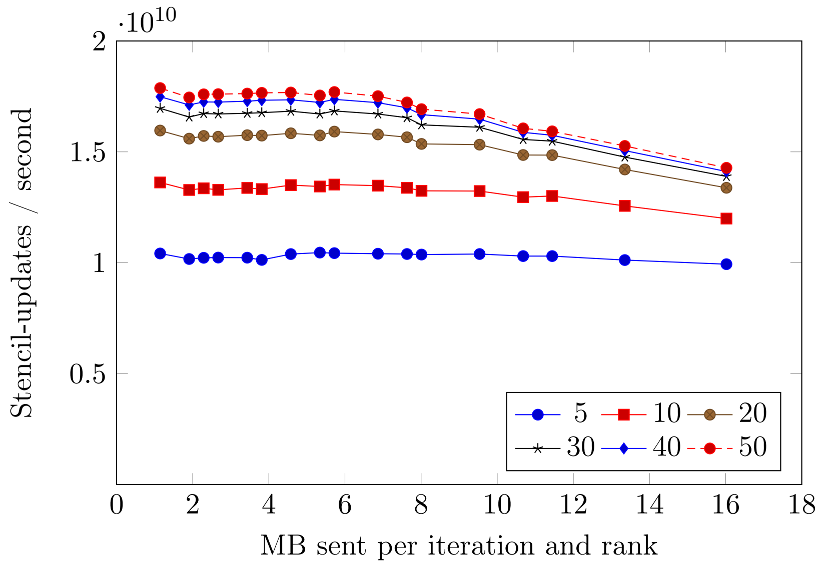


Figure 5.18.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-Only-Nonblocking` on 16 nodes of the ICE Cluster.

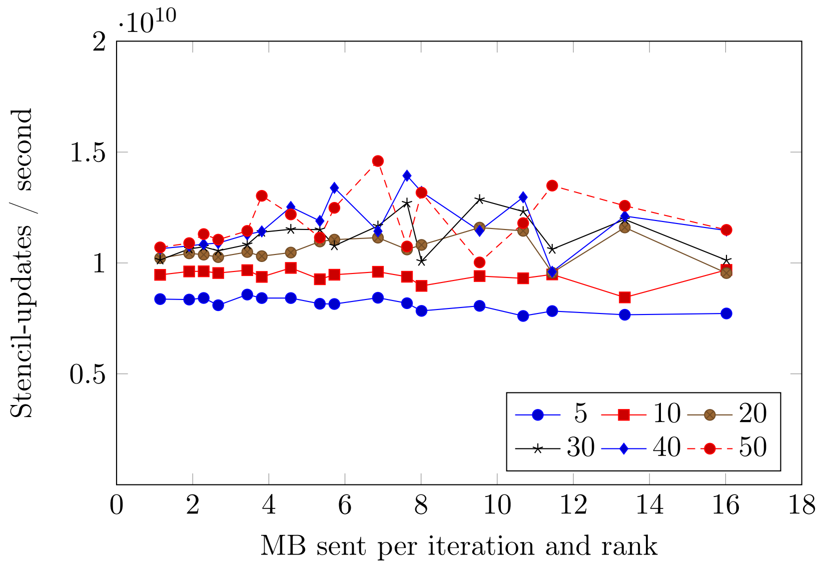


Figure 5.19.: 3D-Jacobi with a one dimensional domain decomposition. `MPI--OpenMP-Nonblocking` on 16 nodes of the ICE Cluster.

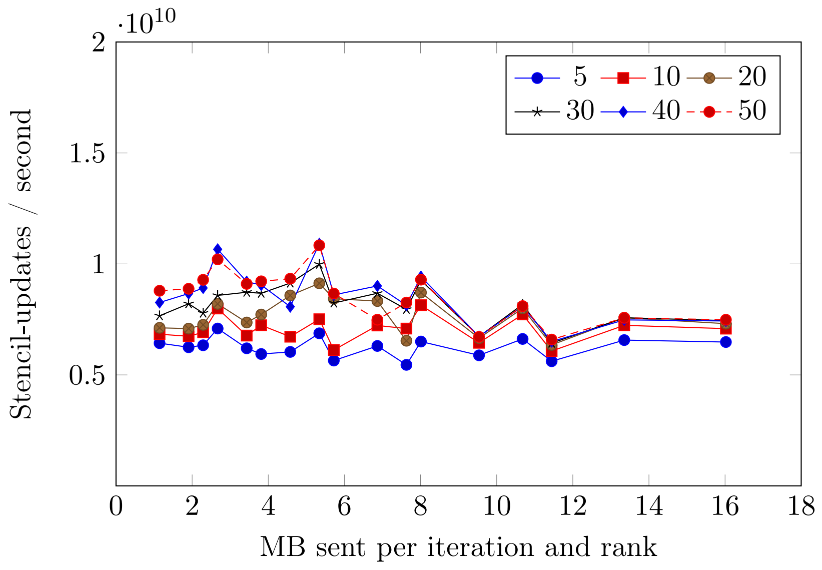


Figure 5.20.: 3D-Jacobi with a one dimensional domain decomposition. `MPI--OpenMP-Nonblocking` calling `MPI_Test` regularely on 16 nodes of the ICE Cluster.

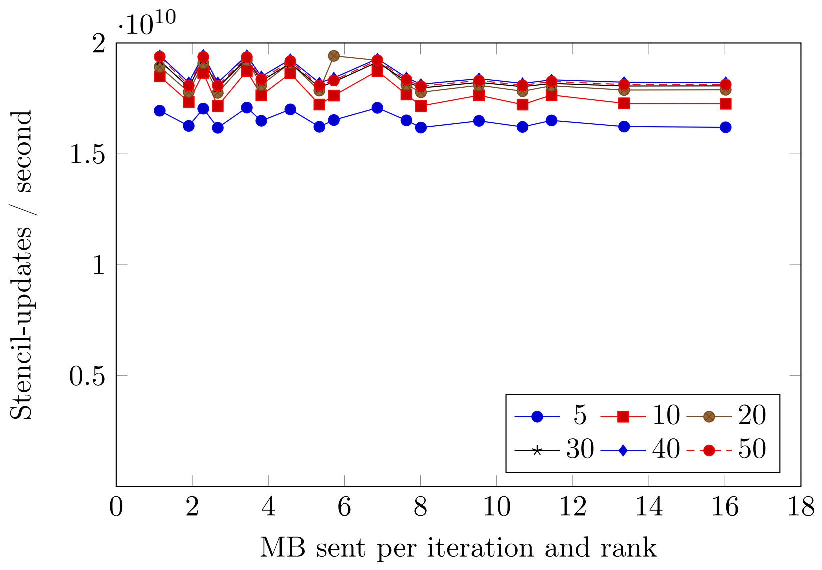


Figure 5.21.: 3D-Jacobi with a one dimensional domain decomposition. `commtask` on 16 nodes of the ICE Cluster.

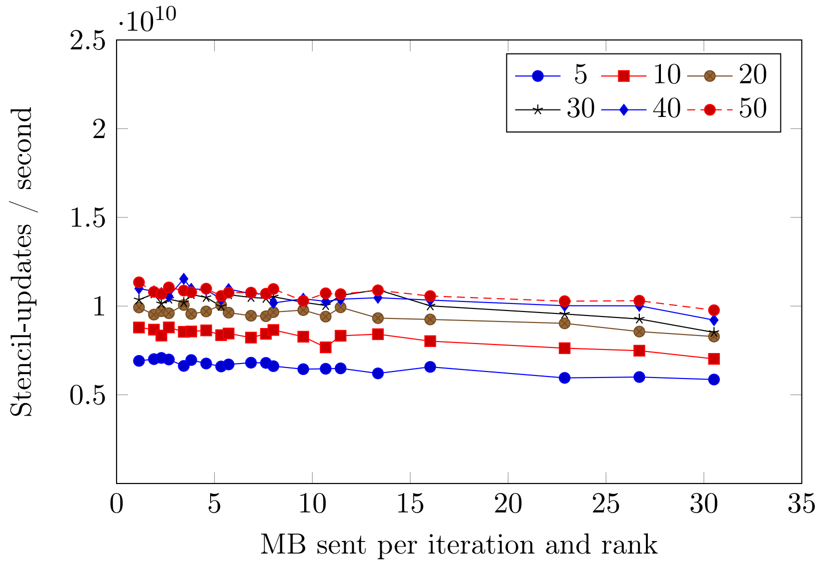


Figure 5.22.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-Only` on 5 nodes of the SuperMUC Fat Nodes.

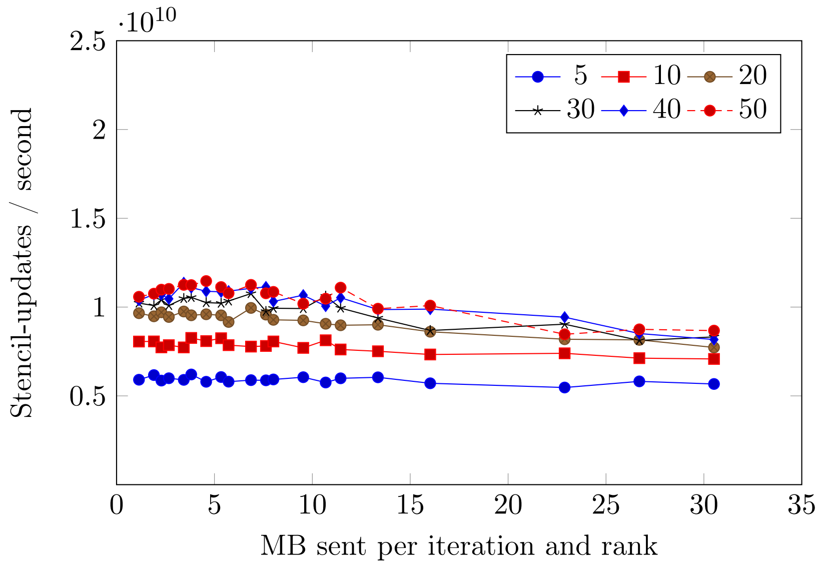


Figure 5.23.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-Only-Nonblocking` on 5 nodes of the SuperMUC Fat Nodes.

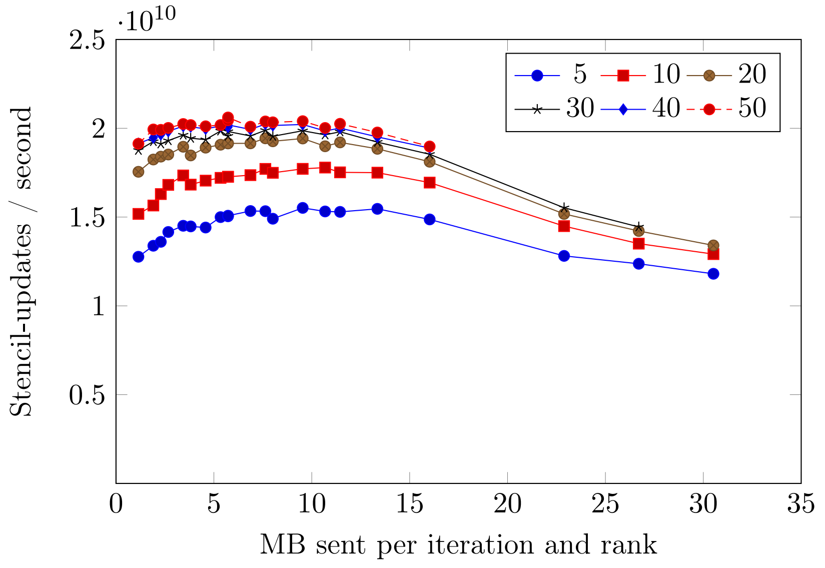


Figure 5.24.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-OpenMP` on 5 nodes of the SuperMUC Fat Nodes.

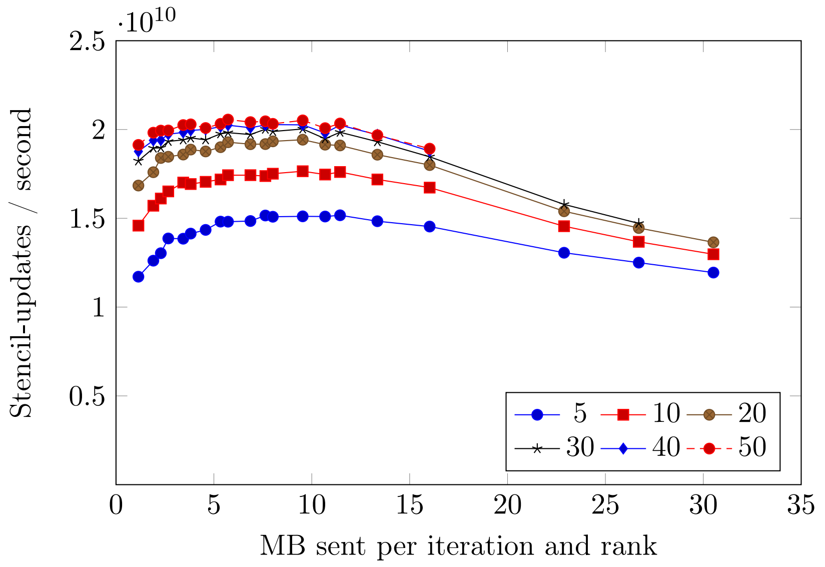


Figure 5.25.: 3D-Jacobi with a one dimensional domain decomposition. `MPI--OpenMP-Nonblocking` on 5 nodes of the SuperMUC Fat Nodes.

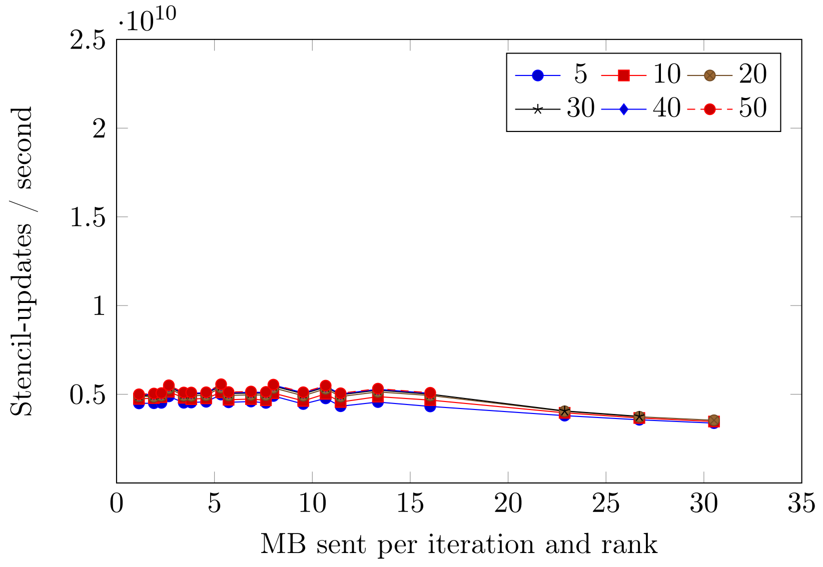


Figure 5.26.: 3D-Jacobi with a one dimensional domain decomposition. `MPI--OpenMP-Nonblocking` calling `MPI_Test` regularely on 5 nodes of the SuperMUC Fat Nodes.

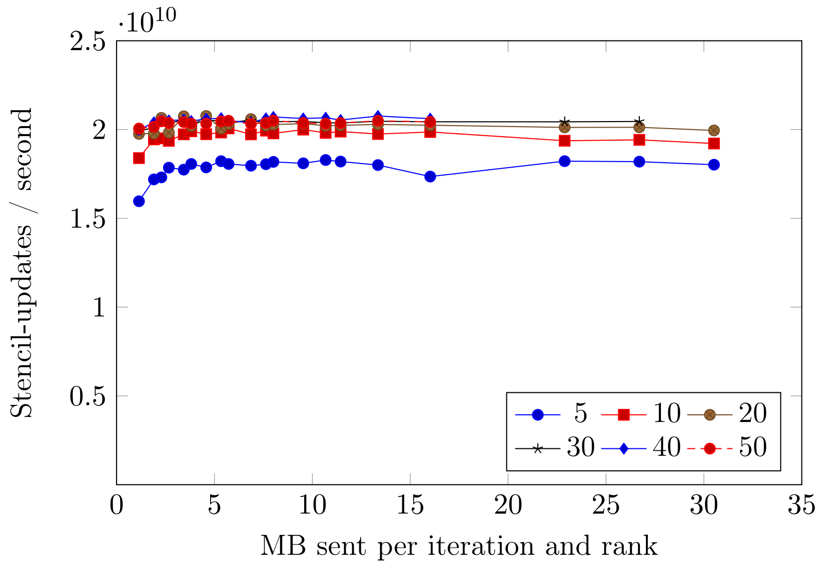


Figure 5.27.: 3D-Jacobi with a one dimensional domain decomposition. `commtask` on 5 nodes of the SuperMUC Fat Nodes.

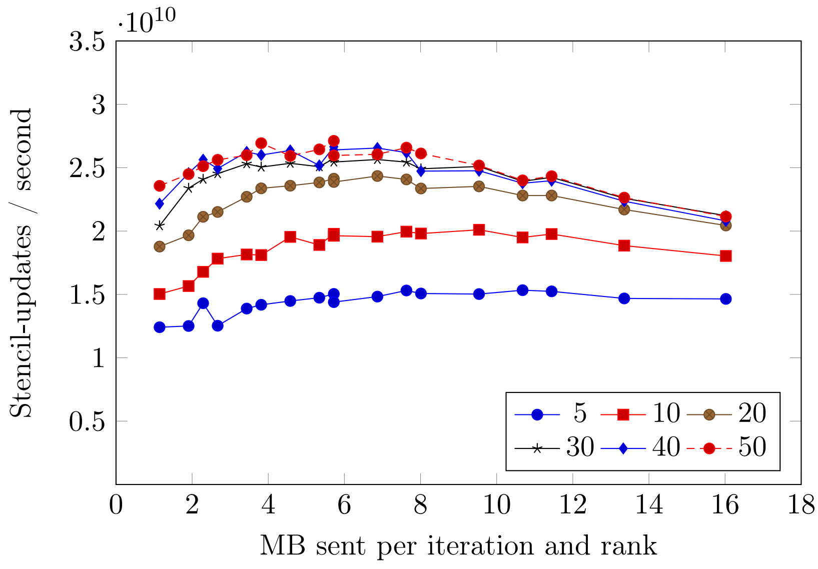


Figure 5.28.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-Only` on 20 nodes of the LiMa Cluster.

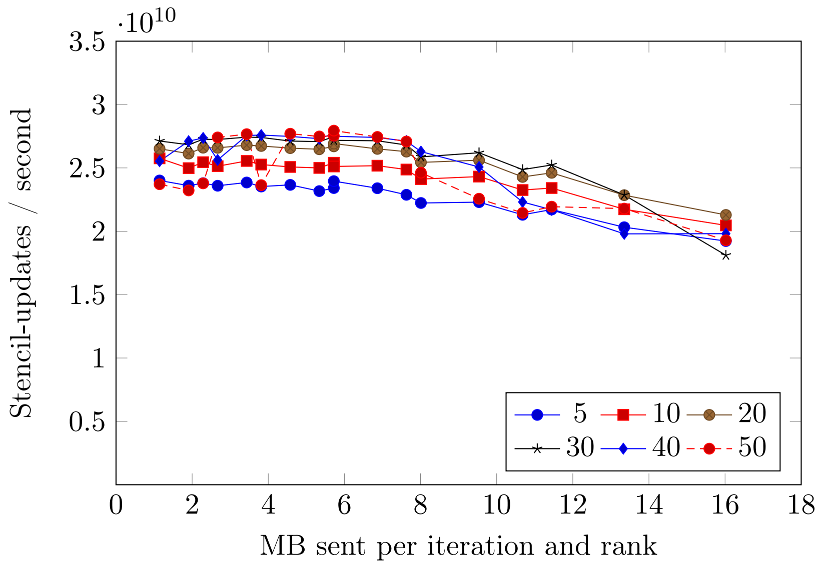


Figure 5.29.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-OpenMP` on 20 nodes of the LiMa Cluster.

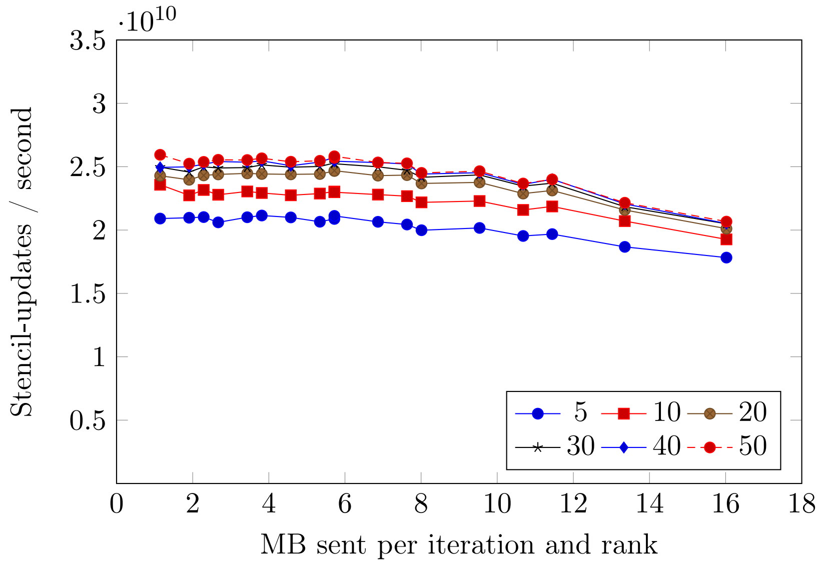


Figure 5.30.: 3D-Jacobi with a one dimensional domain decomposition. `MPI-OpenMP-Nonblocking` on 20 nodes of the LiMa Cluster.

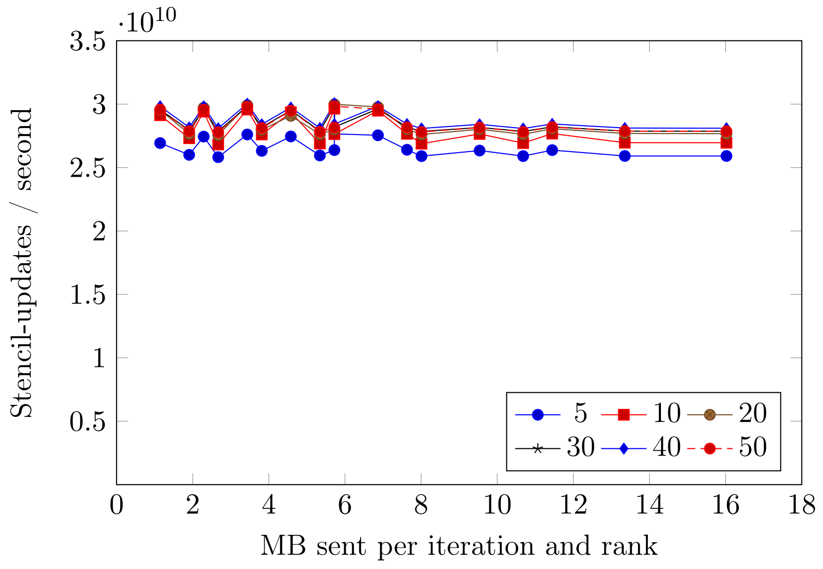


Figure 5.31.: 3D-Jacobi with a one dimensional domain decomposition. `commtask` on 20 nodes of the LiMa Cluster.

5.3. Three-dimensional Decomposition

5.3.1. Computational Domain Decomposition

While the one dimensional domain decomposition used in the previous Section 5.2 is the easiest from a programming standpoint, it is not the most efficient. As the communication in the parallel Jacobi algorithm can be seen as (necessary) overhead, it should be minimized if it can not be removed, or hidden, entirely. The common approach to do this is by minimizing the relation of useful work (i.e. stencil computations) to the surface of the computational blocks assigned to the used `MPI` ranks (i.e. message sizes). This is done by splitting the computational domain into multiple dimensions and assigning the resulting blocks to `MPI`-ranks, which are therefore also placed in a three dimensional virtual grid.
Depending on the placement of the rank in regard to the computational domain block it is being assigned, a rank can have a different number of neighbors. In cases where the computational domain is being split along two axes, two to four neighbors are possible. In cases where the distribution is done along all three axes, ranks can have anything from three to six neighbors. Due to the fact that the number of neighbors differs between the used ranks, imbalances in regard to the amount of data necessary to communicate and the communication dependent work exist. Additionally, the placement of ranks on the provided nodes in the system is not as straight forward as with one-dimensional splitting. In order to minimize the communication distance `MPI` provides library functions to optimize and facilitate the assignment of computational domain blocks to ranks. For this work, this has been done using the `MPI`-function `MPI_Cart_create`, which creates a new `MPI` communicator to which topology information has been attached. The rank order may be renumbered by the `MPI` implementation. Through additional functions, such as `MPI_Cart_shift` and `MPI_Cart_coords`, the neighbors in the created cartesian topology can, for example, be determined easily.
Another major difference to the one-dimensional splitting is the data layout for the different messages needed to be exchanged with the different neighboring ranks. The same as discussed above applies: Messages sent along the `z`-dimension are based on `x`-`y`-planes and therefore a communication buffer which is sequential in memory. Communication along the `x`-dimension is based on `z`-`y`-planes. As elements sequential in the `y`-axis are also sequential in memory, these messages are based on multiple stripes. The length of each stripe is dependent on the ranks assigned part of the `y`-dimension. The number of stripes depends on the assigned `x`-dimension size. Communication along the `y`-dimension is based on `z`-`x`-planes which are made of individual elements which are distributed in the computational domain's memory region evenly, depending on the `x`- and `z`-sizes assigned to the corresponding rank.
As shown in the previous Chapter 4, a lot of overhead seems to be necessary when using noncontiguous memory regions for communication. For `MPI-Only` implementations using blocking communication functions, this overhead is smaller

than the performance improvement gained through the overall minimized communication, but the impact on overlapping and the hybrid approaches has to be seen with the measurements presented below.

The implementations used for the presented results in this section use the `4n-Stencil`, as presented in Section 5.1.2. As this is a fourth order stencil, each rank needs to send and receive four respective planes to and from each communication partner. The message sizes are defined through the respective sub-dimensions assigned to the rank. As the number of ranks is different for the different versions (`MPI-Only` vs. hybrid approaches), the message sizes differ between them. With fewer ranks used in the hybrid implementations, the computational domain parts assigned to the ranks are larger and therefore as well the necessary messages. Nevertheless, the number of messages per iteration for all ranks per iteration are reduced. The intra-node communication is again replaced by shared memory accesses inside the ranks.

Computational Domain Sizes

For the presented results for the three dimensional computational domain decomposition, the chosen dimensions are cubic in all cases. For a chosen system setup (i.e. HPC system, number of nodes on the system, computed iterations), different cubic computational domains have been used, starting with small domain sizes up to large sizes using nearly all of the available main memory. The presented result graphs show time in seconds (`Time(s)`) or the speedup compared to the respective `MPI-Only` version on the `y`-axis compared to the computational domain size on the `x`-axis. The `x`-axis is scaled linearly in regard to overall computational domain size, i.e. the number of elements defined for the entire computational domain.

5.3.2. Discussion of Results

	Dell notebooks (2*4 cores)	**CoolMUC-2** (2*4*7 cores)	**CoolMUC-2** (27*28 cores)
Time(s)	5.32	5.34	5.36
Speedup	5.33	5.35	5.37

Table 5.2.: Overview of result graphs

An overview of the different graphs presenting the results from the different used systems can be found in Table 5.2. Three different setups have been used to represent different scenarios of possible resource environments. For programmers who do not have very large dedicated HPC systems available, parallelization is still possible through the use of multiple regular computers or notebooks. Two Dell notebooks running the Linux operating system have been directly connected

via an Ethernet cable. Using a switch and more notebooks, the same setup can easily be extended to use more compute nodes (i.e. notebooks or desktop PCs) in order to provide more computational cores. Each of the two notebooks provide a uniform memory access (UMA) environment with four cores, as hyper threading has been turned of for the test executions.
For programmers who have access to a small set of connected servers, tests have been executed on two nodes of the CoolMUC-2. Each node on the CoolMUC-2 provides a NUMA environment with four sockets providing seven cores each. For the hybrid approaches, other than for the other system setups used, one `MPI` rank has been used per socket, not per node. Therefore eight `MPI` ranks have been used, each being pinned to a socket and each of its seven threads pinned to one of the respective cores. While the compute nodes do provide a NUMA environment, the shared memory parallelization inside the ranks happens in a UMA subenvironment.
Finally, to investigate the behavior of the implementations on a larger HPC system, 27 nodes of CoolMUC-2 have been used. As a three dimensional domain decomposition is done, this allows for a splitting of each dimension into three parts, resulting in cubic subdomains for each of the 27 ranks.

As the `OpenMP` versions did not perform well for the approach taken in Section 5.2, the versions used here are as follows: `MPI-Only`, for which the communication phase has been implemented using `MPI_Isend` and `MPI_Irecv` for all communication partners followed directly by `MPI_Waitall`. This avoids deadlocks while not having to separate communication independent from the dependent work. Instead of the `MPI-OpenMP` versions, a hybrid version combining `MPI` with `Pthreads` has been implemented, as described above (`MPI-Pthread-Classic-Nonblocking`). The two versions represented in the result graphs differ only in regard to the use of `MPI_Test`. While one version implements the classic approach of not calling into the `MPI` library during the computational phase (communication independent computation), the second version does so at the beginning of the computation of cache each block. Both versions distinguish between communication dependent and independent work. Finally, the `commtask` version implements the combination of all proposed aspects, including work stealing, dynamic message progression and prioritization of communication dependent work by scheduling it as soon as the respective communication is finished.

Only for the scenario using 27 nodes of the CoolMUC-2 did the `MPI-Pthread-Classic-Nonblocking` version show better performance than the `MPI-Only` approach. For the other setups, they were outperformed by `MPI-Only`. Nevertheless, using the `commtask` approach, the performance was better than the `MPI-Only` approach in all cases. The improvement was little for the two setups using two nodes only but was able to result in a large performance improvement when using the 756 cores of the 27 nodes on CoolMUC-2, especially for large

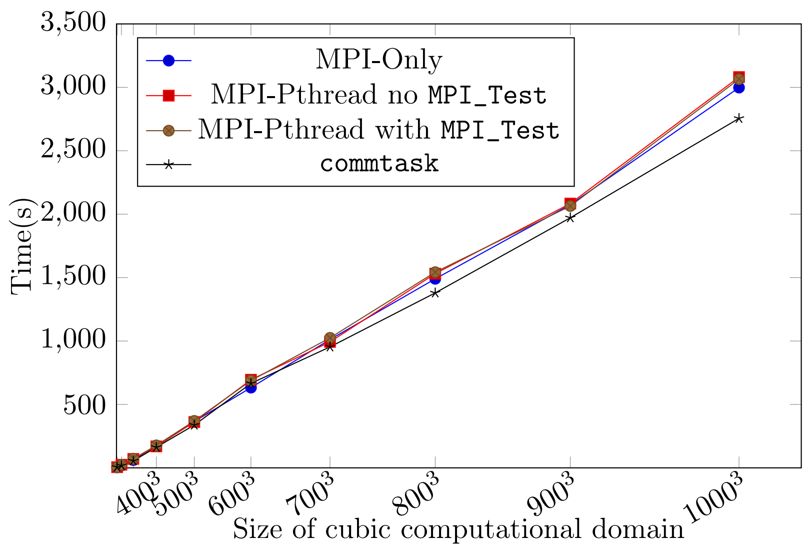


Figure 5.32.: 3D-Jacobi with a three dimensional domain decomposition. Times for 1000 iterations executed on 2 Dell notebooks connected via direct Ethernet. For each version - computational domain combination, the time for the best splitting of the dimensions is shown.

computational domains. In all cases the `commtask` approach improved performance of the hybrid implementations.

Dell notebooks

On the two Dell notebooks, the eight cores were used as follows. As in all presented results, the `MPI-Only` version started one rank per core. The three dimensional splitting of the computational domain was done by splitting each axis into two parts ($2^3 = 8$). For the hybrid approaches, one rank was placed on each node, using four threads each.
The performance of the `MPI-Only` and `MPI-PthreadClassic-Nonblocking` versions is quite close in regard to absolute timing, as can be seen in Figure 5.32. The `MPI-PthreadClassic-Nonblocking` versions are a little worse for nearly all computational domains chosen, and it can be seen that adding `MPI_Test` calls manually to the code nearly always results in overhead. Especially for larger computational domains can the `commtask` approach improve performance. Due to the little communication done and the previous detailed analysis on the behavior of the approach, this suggests that the communication is actually hidden behind useful computation. The performance differences can be seen more clearly when looking at the speedup of the different versions in respect to the `MPI-Only` implementation in Figure 5.33.

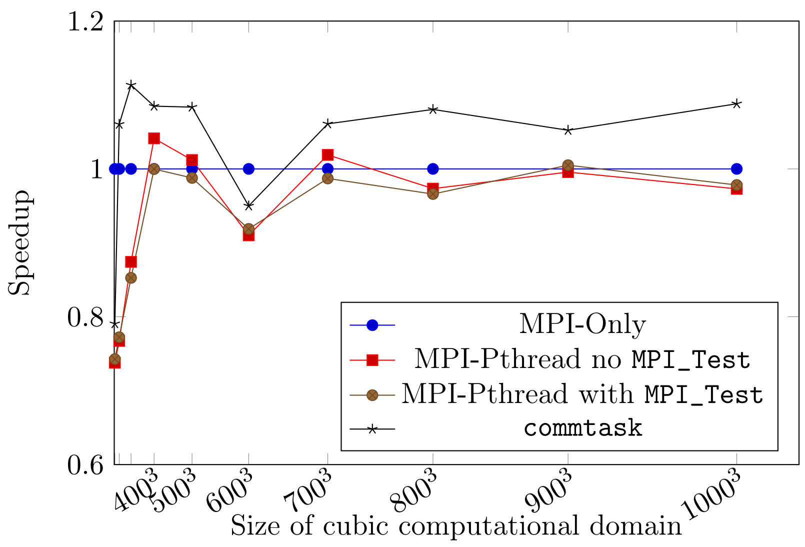


Figure 5.33.: 3D-Jacobi with a three dimensional domain decomposition. Speedup against the best observed `MPI-Only` version for 1000 iterations executed on 2 Dell notebooks connected via direct Ethernet.

CoolMUC-2 - two nodes

The timing results for the execution of the different versions on two nodes of the CoolMUC-2 can be seen in Figure 5.34 and the corresponding speedup in Figure 5.35. For these results, one `MPI` rank was placed on each of the eight sockets of the two nodes, each using seven threads for the respective cores for the hybrid approaches. As the overall number of ranks in the `MPI-Only` version was not cubic, different splittings of the different dimensions have been tested. These resulted in different times, showing that splitting along multiple dimensions is better than a one dimensional domain decomposition as used in Section 5.2 (which, on the other hand, is easier to implement). The used splittings along the three dimensions are $\{x, y, z\} = \{4, 2, 7\}, \{4, 1, 14\}, \{2, 1, 28\}, \{1, 1, 56\}$, in order of decreasing performance. As a result, different splittings have also been executed for the hybrid approaches, showing different performance as well, but with the difference that the performance can not be ordered as for the `MPI-Only` approach. The tendency of the used splittings in decreasing performance is $\{x, y, z\} = \{2, 2, 2\}, \{2, 1, 4\}, \{1, 1, 8\}$. Nevertheless, for the `MPI--PthreadClassic-Nonblocking` version with `MPI_Test` calls the splitting $\{x, y, z\} = \{2, 1, 4\}$ outperformed the even splitting for the largest chosen computational domain size, if only slightly.
For each implementation, the presented results use the best observed time from the different splittings.

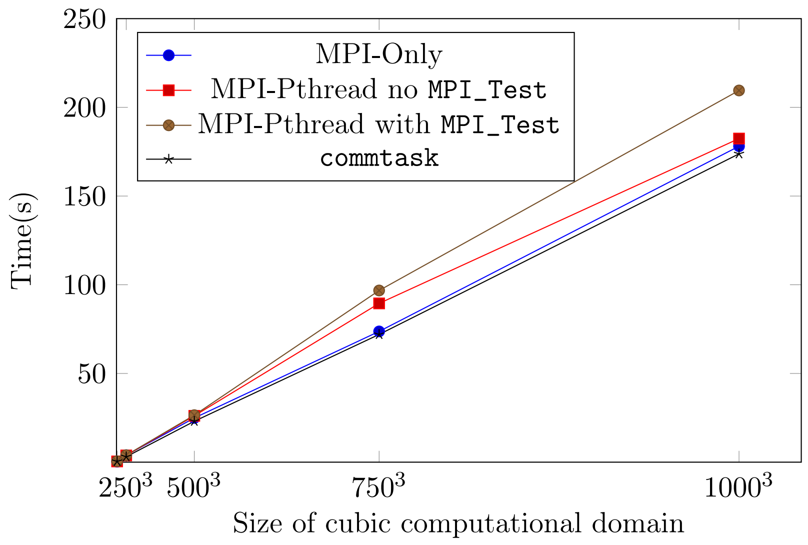


Figure 5.34.: 3D-Jacobi with a three dimensional domain decomposition. Times for 1000 iterations executed on 2 nodes of the CoolMUC-2. For each version - computational domain combination, the time for the best splitting of the dimensions is shown.

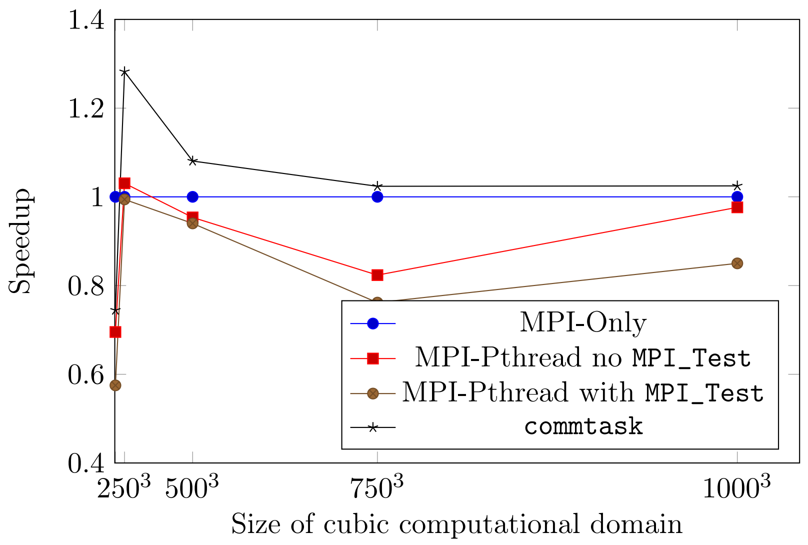


Figure 5.35.: 3D-Jacobi with a three dimensional domain decomposition. Speedup against the best observed `MPI-Only` version for 1000 iterations executed on 2 nodes of the CoolMUC-2.

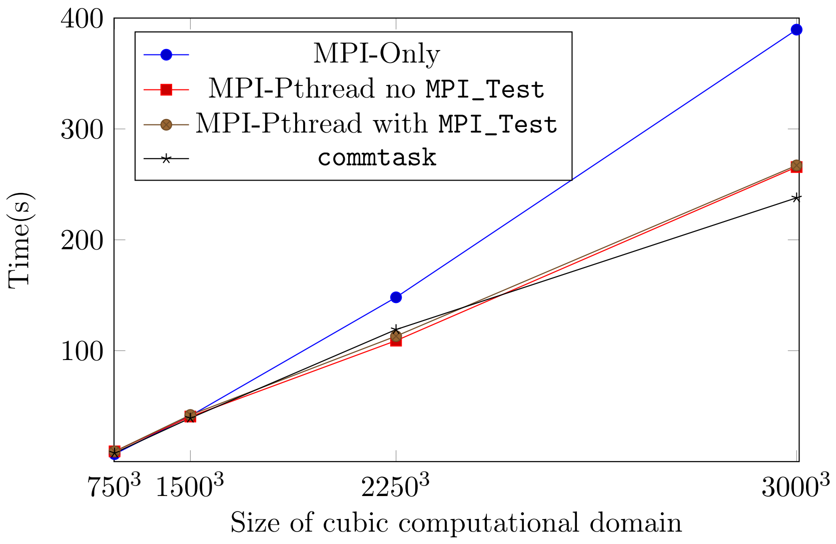


Figure 5.36.: 3D-Jacobi with a three dimensional domain decomposition. Times for 1000 iterations executed on 27 nodes of the CoolMUC-2. For each version - computational domain combination, the time for the best splitting of the dimensions is shown.

Looking at the results it can be seen that the hybrid approach can only outperform the `MPI-Only` version when all proposed optimizations are applied in the form of the `commtask` approach. The other hybrid approaches perform worse and are unable to make use of the theoretical advantages.
With little communication due to the use of only two nodes, and therefore little inter-node communication, the advantages of using the proposed `commtask` approach are limited. Nevertheless, the hybrid codes are improved by applying the concepts proposed throughout this work.

CoolMUC-2 - 27 nodes

For these results, presented in Figures 5.36 and 5.37, 27 nodes of the CoolMUC-2 cluster have been used. This results in 756 ranks on 756 cores for the `MPI-Only` version and 27 ranks on one node each for the hybrid approaches. Each rank uses 28 threads which are being pinned to one core each.
As for the setup described above, different splittings were used and showed different performance. For the `MPI-Only` approach, the used splittings are $\{x, y, z\} = \{9, 4, 21\}, \{9, 2, 42\}, \{8, 1, 42\}, \{9, 1, 84\}$ in decreasing performance. A difference between $\{x, y, z\} = \{9, 2, 42\}, \{8, 1, 42\}$ is very small, but the gap between the best and worst splitting increases with increasing computational domain size. For the `MPI-PthreadClassic-Nonblocking` versions the behavior is similar independent of the question whether or not `MPI_Test` is being called. $\{x, y, z\} = \{3, 3, 3\}$

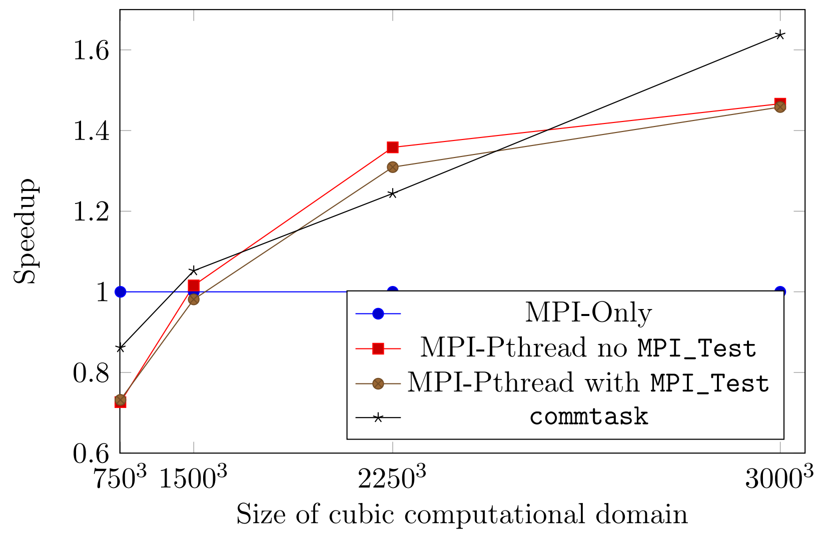


Figure 5.37.: 3D-Jacobi with a three dimensional domain decomposition. Speedup against the best observed `MPI-Only` version for 1000 iterations executed on 27 nodes of the CoolMUC-2.

is worse than a splitting along two dimensions only, which has been done using $\{x, y, z\} = \{1, 3, 9\}$ and $\{3, 1, 9\}$ with a slight advantage for the former for large computational domains. This is surprising as the first results in more communication along the `y`-dimension, which consists of single elements distributed throughout the used memory regions instead of "stripes" of elements as would be the case for communication along the `x`-dimension.

For the `commtask` approach, the same splittings were used and the performance was closer to the expected behavior: Splitting along the `y`-dimension in favor of the `x`-dimension results in worse performance. While for small computational domains the even splitting along all dimensions is best, for larger computational domains the splittings in order of decreasing performance are $\{x, y, z\} = \{1, 3, 9\}, \{3, 3, 3\}, \{3, 1, 9\}$. Splitting in multiple dimensions reduces the ratio of communication to computation and not splitting along the `y`-dimension avoids having to deal with an `MPI` data type describing many individual `double` values distributed throughout physical memory.

With the large number of nodes, the inter-node communication necessary is larger than in a setup with two nodes only. Also, the number of ranks which have to communicate with six different neighbors in the `MPI-Only` version is a lot larger. Communication has a larger impact, and being able to hide communication behind computation results in a larger potential for performance improvement as compared to the other setups. All hybrid approaches outperform the `MPI-Only` approach for all but the smallest computational domain sizes. While the `MPI-PthreadClassic-Nonblocking` approaches show better performance for mid sized computational domains, the `commtask` approach is better for small sizes and best for very large computational domains.

All in all it can be said that using the `commtask` approach in a three dimensional domain decomposition does not have the same, clearly visible, impact on performance as has been observed when applying it to the one dimensional domain decomposition described previously. It is able to improve hybrid implementations and shows increasing advantages with larger problem sizes on larger clusters.
The performance improvements cannot be attributed to the communication overlap as clearly and the observed performance improvements seem to be a result of the combination of communication overlap and work balancing inside the compute nodes.

Looking back at the results of the benchmark presented in Section 4.2, it is no surprise that the impact is less observable as with the one dimensional splitting. For communication based on sequential memory buffers, the results showed that nearly all the time spent in the communication functions is actually used for data transfer. When achieving data transfer overlap, these times were reduced to practically nothing. In contrast, when using `MPI` data types to create noncontiguous communication buffers, additional overhead was introduced. It has been shown that this overhead is related to the necessary data movement done inside the `MPI` library in order to create temporary sequential memory buffers, which are communicated. Therefore the time spent inside the library functions is, in large part, spent on local operations.
As the data types used in the three dimensional Jacobi algorithm are less uniform and different data types are used for different communication partners, it is not clear which approach and which optimizations are applied inside the `MPI` library, but it can be assumed with high certainty that only a (possibly small) part of the time spent in the library is needed for data transfer. As the additional overhead cannot be overlapped, the impact of using the `commtask` approach is reduced in this regard.
Nevertheless the combination of all aspects still provides improvements to the hybrid implementations as the available threads can share the load of necessary overhead and balance out the useful work at the same time in order to minimize imbalances and, therefore, idle times on the available cores.

5.4. Discussion

The proposed `commtask` approach, presented in Chapter 4, has been applied to a stencil algorithm, namely a three dimensional Jacobi algorithm, representing a wide range of programs which have stencil codes at their core. These stencil codes exhibit hierarchical parallelism, which matches both the hybrid hardware and hybrid programming models which have been discussed earlier in this work.

The computational domain of the computed problem can be split into three dimensional subdomains, which can be assigned to higher level parallelization units, i.e. the used `MPI` ranks. These in turn can internally split the assigned part of the computational domain again and assign these parts to the available threads when parallelized in a hybrid fashion.
Communication at the `MPI` layer is necessary due to the fact that required values from neighbor elements at the edge of the subdomains from the previous iteration have been computed by a different rank. Depending on the kind of stencil used, especially the order of the stencil (Section 5.1.2), these messages have different sizes. Depending on the used domain decomposition approach, the message buffers for these messages also have a different memory footprint. Two kinds of computational domain decompositions have been applied and evaluated: one dimensional splitting of the computational domain (Section 5.2) and a multi-dimensional decomposition (Section 5.3).

Applying a one-dimensional computational domain decomposition is easiest from a programming point of view. The resulting communication pattern uses message buffers which are sequential in memory, which has been shown to work very well with manual progression functions, as discussed in Chapter 4. Starting with an `MPI-Only` implementation using blocking communication in a communication phase, other different implementations have been created and evaluated. These include `MPI-Only-Nonblocking`, for which the communication dependent work and communication independent work have been separated. The blocking communication phase has been exchanged with nonblocking communication functions, surrounding the communication independent work. The same has been done for a hybrid `MPI-OpenMP` version, resulting in the combination of hierarchical parallelization and nonblocking communication in the `MPI-OpenMP--Nonblocking` implementation. Finally, all aspects discussed in regard to the proposed `commtask` have been implemented using a Pthread based `OpenMP` like runtime. Details on the implementations have been presented in Section 5.1.4.
For an extensive set of computational domain configurations, all implementations have been executed and evaluated on multiple modern HPC systems, including the LiMa Cluster and SuperMUC. For all combinations but the `commtask` implementation, the impact of increasing communication requirements can be observed. While this is expected for implementations using blocking communication, i.e. separate communication and computation phases, the results confirm what has been shown with the benchmark presented in the previous chapter. Not one of the used `MPI` implementations, which included Intel MPI, IBM MPI, MPT MPI and MPICH2, is able to provide real communication overlap even when making use of the corresponding `MPI` functions.
The results show that, while smaller single socket HPC systems like the used ICE Cluster and Woodcrest Cluster do not benefit from a hybrid implementation approach, new multi-socket systems like SuperMUC and LiMa can be used with higher efficiency when combining distributed with shared memory

parallelization. Nevertheless, the impact of increasing communication overhead for larger computational domains is still clearly visible. Only when looking at the presented results for the `commtask` implementation can all systems be used perfectly in regard to communication overlap. Perfect overlap of the entire communication, including data transfer, has been observed for all test scenarios. As for all cases, the communication was less time-consuming than the computation, all used cores have been optimally used for useful computation throughout the entire program execution.

Splitting the computational domain in multiple dimensions is improving the ratio of communication to computation and minimizes communication overhead. The implementation is more complex as multiple communication partners in multiple dimensions have to be considered. Additionally the data layout of the message buffers needed for the different neighbors is different for each dimension. The three dimensional computational domain is mapped to a one dimensional memory region and depending on the chosen communication dimension, the used message buffer is either sequential, consists of multiple (smaller) sequential memory regions, or many individual elements. The use of message buffers which are not sequential in memory introduces a lot of overhead in regard to communication (as shown in Section 4.2.6). Nevertheless, the advantage through minimizing the overall communication requirements is larger, resulting in performance improvements compared to a one dimensional domain decomposition approach. Due to the reduced percentage of the entire program execution used for actual message transfer, the impact of being able to hide communication behind useful work is reduced. The results showed that this is especially true when using a small number of compute nodes with little inter-node communication. Even when implementing the shared memory part on a lower level using Pthreads instead of `OpenMP`, the `MPI-Only` approach can outperform the hybrid versions. Nevertheless, applying the `commtask` approach on such configurations can at least improve the hybrid approach to match the `MPI-Only` counterpart.
The true potential of the `commtask` approach has been shown when using it together with a three dimensional domain decomposition on a larger part of a modern multi-socket HPC system, namely 27 nodes of the CoolMUC-2 providing 756 cores in total. As observed with the results of the one dimensional domain decomposition, the hybrid parallelization can make better use of the hardware than the `MPI-Only` approach, even when no communication overlap can be achieved. Especially for large computational domains with increasing communication demands, the results show that the `commtask` approach can optimize this further. Together with the observations made in earlier parts of this work, the results indicate that perfect communication overlap has been achieved and idle times of the available computational cores have been minimized.

For both computational domain decomposition approaches, manually adding the message progression function calls to different parts of the implementations, e.g. at the end of the computational loops at different levels (elements, lines, planes), has been done. In all cases this resulted in additional overhead and decreased performance without achieving the desired overlap.

The application of the `commtask` approach to real HPC algorithms works and is able to improve hybrid implementations. For cases where the communication buffers are sequential in memory the time spent in communication related functions is reduced to basically nothing and the communication successfully hidden behind useful computation. In cases where `MPI` data types are needed, the advantages of applying the `commtask` approach outweigh the introduced overhead when a lot of inter-node communication is necessary. Even when this is not the case, it improves other hybrid approaches to a point where it can at least match `MPI-Only` implementations.
Additionally, adding `MPI` awareness to the used `OpenMP` runtime offers opportunities in regard to optimizing message progression. As the necessary parameters, like number and timing of calls to the message progression functions, are dependent on the used HPC system and the provided software environment, they can be optimized by system experts and provided through system software without the need to change the parallel software and reduce its portability.

CHAPTER 6

Conclusions

6.1. Summary

During a given computational phase, two kinds of workloads generally exist in typical modern high performance computing applications: communication-dependent and communication-independent work. The overlap of communication with communication-independent work has been discussed and analyzed in detail in this work (Section 2.2). It has been shown that while the `MPI` standard offers the necessary functions which should be able to overlap communication with computation, it cannot be achieved with modern `MPI` implementations on modern HPC systems (Section 4.1).
Different reasons why this is the case have been discussed. Due to the fact that no calls to the `MPI` library are done during the computation phase, behind which the communication should be hidden, steps like hardware programming are not taken care of, and instead of being overlapped with computation, communication is moved to the communication termination function. The two possible ways discussed in literature to achieve overlap are the use of progress threads and manual progression. As discussed in this work, the use of progress threads is not always possible nor feasible. The focus of this work was manual progression, i.e. the use of progression functions throughout the computation phase.
The behavior of different `MPI` implementations and their capabilities to overlap communication without and with calls to progression functions has been benchmarked and analyzed in detail (Section 4.2). The results show that while no implementation is able to achieve overlap per default, it is possible through the use of progression functions. Nevertheless, it has also been shown that the correct use of these, in regard to timing and number of calls, is crucial to performance. Manual placement in `MPI-Only` parallelized codes is difficult when trying

to achieve overlap with minimal additional overhead. Even when achieving the overlap on one system, the same approach can behave differently on another.

A different approach to program for modern HPC systems is combining shared- and distributed-memory parallelization in a hybrid way. At a higher level, `MPI` is used for parallelization, placing one rank on each compute node or on each socket in the provided NUMA environment. At a lower level, each `MPI` rank parallelizes the assigned work in a shared-memory fashion using Pthreads or `OpenMP`. While this approach corresponds to the hybrid hardware setup of the used HPC systems, it is being discussed controversially in literature. Even for the same algorithm, it can be better or worse than an `MPI-Only` implementation depending on the used system or even on the chosen input data.
Its advantages have been discussed throughout this work and its potential in regard to communication overlap has been the focus of the presented approaches. Through `MPI` awareness, the lower level parallelization, i.e. the `OpenMP` runtime, can be notified of outstanding asynchronous communication requests. Together with knowledge about the used HPC system and its software environment, the `OpenMP` runtime can take care of scheduling and calling progression functions. A respective extension to the `OpenMP` standard, namely `commtasks`, has been proposed and discussed in detail (Section 4.4). Together with a new schedule for the `OpenMP` loop parallelization construct (Chapter 3), the proposed `commtasks` have additional advantages. Besides optimal use and timing of progression functions, NUMA awareness is being added to hybrid programs. Work balancing inside the `MPI` ranks is achieved through work stealing between the used `OpenMP` threads. Finally, idle times and parallelization overhead are minimized by scheduling communication related work steps with a high priority.
The proposed approaches have been implemented for the Jacobi Relaxation method, working on a three dimensional computational domain (Chapter 5). This algorithm represents a wide range of so called stencil codes as well as, in regard to this work, programs which can be parallelized in a hierachical manner. A large set of test scenarios has been executed on different HPC systems for many different parallel implementations: `MPI-Only` as well as hybrid `MPI-OpenMP` implementations, both using blocking and nonblocking communication functions, and the proposed `commtask` implementation.
The results show that perfect communication overlap can be achieved with the proposed approach and performance can be improved drastically especially for hybrid implementations. Resource usage is additionally optimized due to the fact that idle times are minimized on the used cores. At the same time, the ease of use for programmers is achieved through minimal changes to the existing `OpenMP` standard and the fact that optimizations in regard to the use of progression functions is moved to HPC system vendors and administrators, who can fine tune the proposed `OpenMP` extensions in the installed `OpenMP` runtime.

6.2. Future Work

It has been shown that the proposed `commtask` approach works well on the used HPC systems together with the used `MPI` implementations. The chosen algorithm for the proof of concept example was able to prove the potential and effectiveness of the approach, but also revealed shortcomings of the existing `MPI` implemenations in regard to communication using noncontiguous memory buffers. Future research can address this problem and extend the proposed `commtask` implementations accordingly. Also, it would be interesting to investigate how the approach can benefit different types of parallel programs, especially those following a master-worker scheme. As those have irregular communication patterns, communication awareness in the used `OpenMP` runtime can remove the burden of checking for communication requests from the application programmer.

While the focus of this work has been manual progression, the use of progress threads is also be possible in combination with the proposed `commtasks`. Instead of scheduling progression functions, the `OpenMP` runtime can interact with progress threads in cases where they are part of future `MPI` implementations. This would remove the overhead introduced by the necessity to call progression functions while keeping all benefits of the `commtasks`. Communication termination can be made known to the `OpenMP` runtime through the progress thread. Prioritizing communication related work, e.g. pre- and post-communication dependent work, can be done as proposed together with work balancing.

Finally, the newest version of the `MPI` standard introduced nonblocking collective operations. These require different steps from different `MPI` ranks at different times, such as forwarding messages to other ranks or even computational steps while accumulating data during operations such as `MPI_Iallreduce`. An `MPI` aware `OpenMP` runtime could recognize the need for action and schedule it to available threads with high priority through the use of the proposed `commtasks`, as well as counteract introduced overhead and imbalances by applying the proposed work stealing. Future research should be done to investigate where the bottlenecks lie in this regard and in which way the proposed `commtask` can overcome these.

Appendix

APPENDIX A

System Descriptions

This chapter describes all systems used throughout this work. All but one system are high performance computing (HPC) systems, installed either at the “Leibniz Supercomputing Centre”[1] (LRZ) or the “Regionales RechenZentrum Erlangen”[2] (RRZE). Three systems entered the TOP500[3] list: The SuperMUC Thin Nodes entered the list in June, 2012, on rank number four. The LiMa Cluster entered the list in November, 2010, on rank number 130 and the Woodcrest Cluster in November, 2006, on rank 124.

[1] `https://www.lrz.de`
[2] https://www.rrze.fau.de/
[3] `http://top500.org/`

A.1. SuperMUC Fat Nodes

The SuperMUC Fat Nodes are part of the SuperMUC Phase 1 installation at the Leibniz Supercomputing Centre (LRZ). They have been installed 2011 and consist of one island with 205 nodes.
The system details for the nodes and the used interconnect can be seen in the following table.

Processor Type	Intel Westmere-EX Xeon E7-4870 10C
Nominal Frequency [GHz]	2.4
Total Number Nodes	205
Total Number Cores	8200
Total Peak Performance [PFlop/s]	0.078
Total Linpack Performance [PFlop/s]	0.065
Total size of memory [TByte]	52
Processor per Node	4
Cores per Processor	10
Cores per Node	40
Memory per Core [GByte]	6.4
Interconnect Technology	Infiniband QDR
UMA/NUMA	NUMA

A.2. SuperMUC Thin Nodes

Also part of the SuperMUC Phase 1 installation at the Leibniz Supercomputing Centre (LRZ), the SuperMUC Thin Nodes have been installed 2012. They consist of 18 islands hosting 512 nodes each. This system entered the TOP500 list of supercomputers at rank number four in June, 2012.
The system details for the nodes and the used interconnect can be seen in the following table.

Processor Type	Intel SandyBridge-EP Xeon E52680 8C
Nominal Frequency [GHz]	2.7
Total Number Nodes	9216
Total Number Cores	147456
Total Peak Performance [PFlop/s]	3.2
Total Linpack Performance [PFlop/s]	2.897
Total size of memory [TByte]	288
Processor per Node	2
Cores per Processor	8
Cores per Node	16
Memory per Core [GByte]	2
Interconnect Technology	Infiniband FDR10
UMA/NUMA	NUMA

A.3. CoolMUC-2

Also part of the Linux Cluster at the Leibniz Supercomputing Centre (LRZ), the CoolMUC-2 has the same processors as the Phase 2 of the SuperMUC. The system details for the nodes and the used interconnect can be seen in the following table.

Processor Type	Intel Haswell EP
Nominal Frequency [GHz]	2.6
Total Number Nodes	384
Total Number Cores	10752
Total Peak Performance [TFlop/s]	447
Total size of memory [TByte]	24.6
Processor per Node	2
Cores per Processor	14
Cores per Node	28
Memory per Core [GByte]	2.3
Interconnect Technology	Infiniband FDR14
UMA/NUMA	NUMA

A.4. LiMa Cluster

The LiMa Cluster has been installed at the “Regionales RechenZentrum Erlangen” (RRZE) in 2010. In November of the same year, it entered the TOP500 list of supercomputers on rank 130.
The system details for the 500 compute nodes and the used interconnect can be seen in the following table.

Processor Type	Intel Westmere Xeon 5650
Nominal Frequency [GHz]	2.66
Total Number Nodes	500
Total Number Cores	6000
Total Peak Performance [TFlop/s]	64
Total Linpack Performance [TFlop/s]	56.7
Total size of memory [TByte]	11.7
Processor per Node	2
Cores per Processor	6
Cores per Node	12
Memory per Core [GByte]	2
Interconnect Technology	Infiniband QDR
UMA/NUMA	NUMA

A.5. ICE Cluster

The ICE Cluster has been installed at the Leibniz Supercomputing Centre (LRZ) in 2010. The system details for the 64 nodes and the used interconnect can be seen in the following table.

Processor Type	Intel Nehalem-EP Xeon E5540
Nominal Frequency [GHz]	2.53
Total Number Nodes	64
Total Number Cores	512
Total Peak Performance [TFlop/s]	5.2
Total size of memory [TByte]	1.5
Processor per Node	2
Cores per Processor	4
Cores per Node	8
Memory per Core [GByte]	3
Interconnect Technology	Infiniband DDR
UMA/NUMA	UMA

A.6. Woodcrest Cluster

The Woodcrest Cluster has been installed at the “Regionales RechenZentrum Erlangen” (RRZE) in 2006. In November of the same year, it entered the TOP500 list of supercomputers on rank 124. The system has been updated throughout the years.
The details for the system used in this work can be seen in the following table.

Processor Type	Intel Woodcrest Xeon 5160 2C
Nominal Frequency [GHz]	3
Total Number Nodes	212
Total Number Cores	728
Total Peak Performance [TFlop/s]	8.736
Total Linpack Performance [TFlop/s]	5.416
Total size of memory [TByte]	1.4
Processor per Node	2
Cores per Processor	2
Cores per Node	4
Memory per Core [GByte]	2
Interconnect Technology	Infiniband
UMA/NUMA	UMA

A.7. Dell Notebook Cluster

This system consists of two high end office notebooks, connected through direct Ethernet. Both notebooks were running the same Linux operating system (Ubuntu 12.04LTS) and used the MPICH (version 3.0.4) `MPI` implementation. The system details can be seen in the following table.

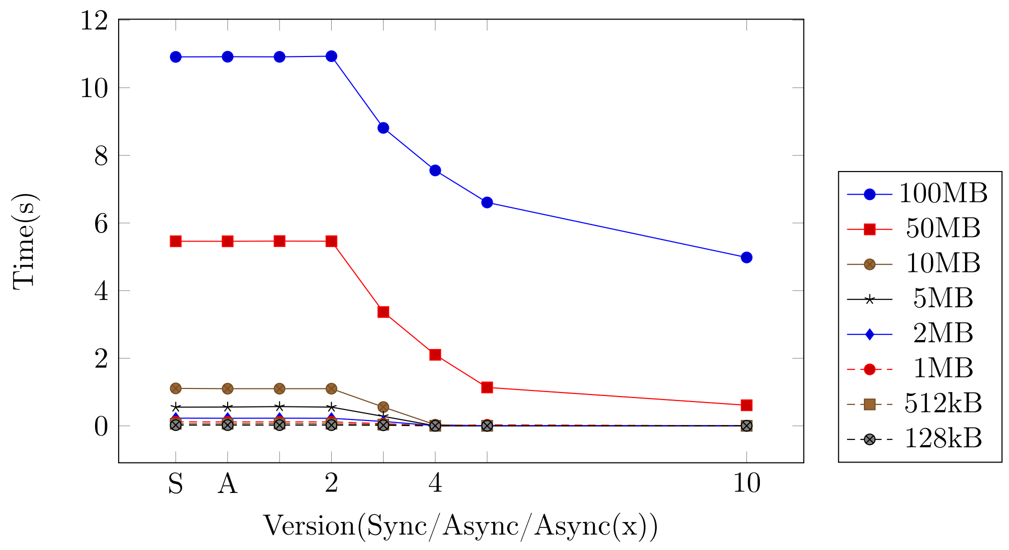
Processor Type	Intel Core i7-3740QM
Nominal Frequency [GHz]	2.7
Total Number Nodes	2
Total Number Cores	8
Total size of memory [GByte]	32
Processor per Node	1
Cores per Processor	4
Cores per Node	4
Memory per Core [GByte]	4
Interconnect Technology	Direct Ethernet
UMA/NUMA	UMA

APPENDIX B

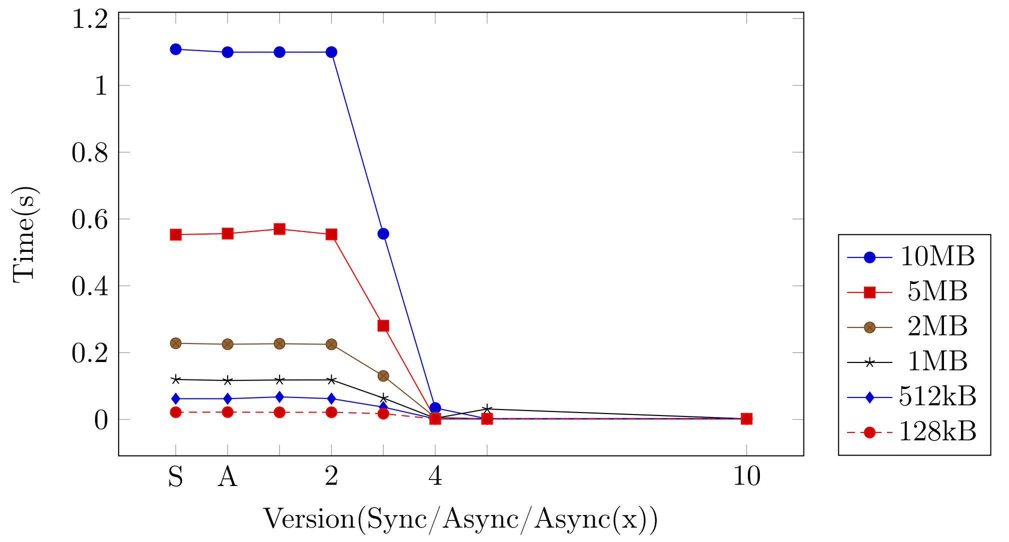
Benchmark: Asynchronous Communication Capabilities of MPI Implementations

B.1. Additional results

Detailed timing measurements as discussed in Section 4.2. The following measurements show the communication timer measurements corresponding to the results presented in Figure 4.10.

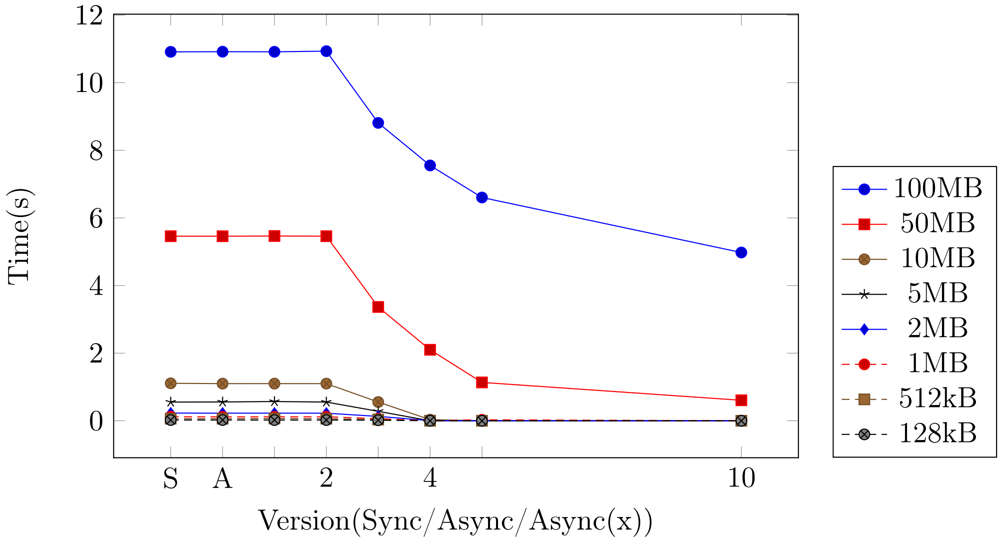


(a) 128kB to 100MB

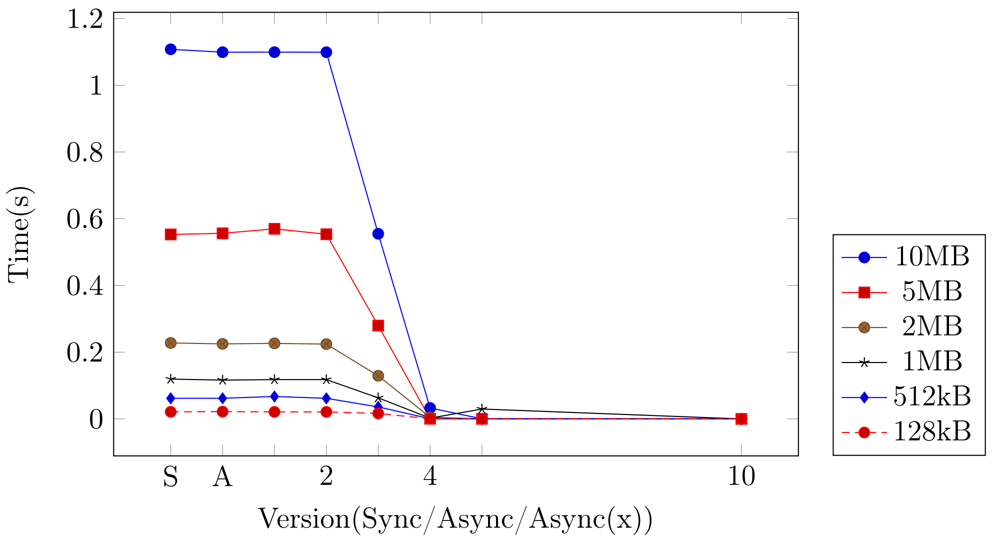


(b) 128kB to 10MB

Figure B.1.: Results micro benchmark new: ICE 2 nodes INTEL40 2MSGs no stripping COMMUNICATION time TOTAL only

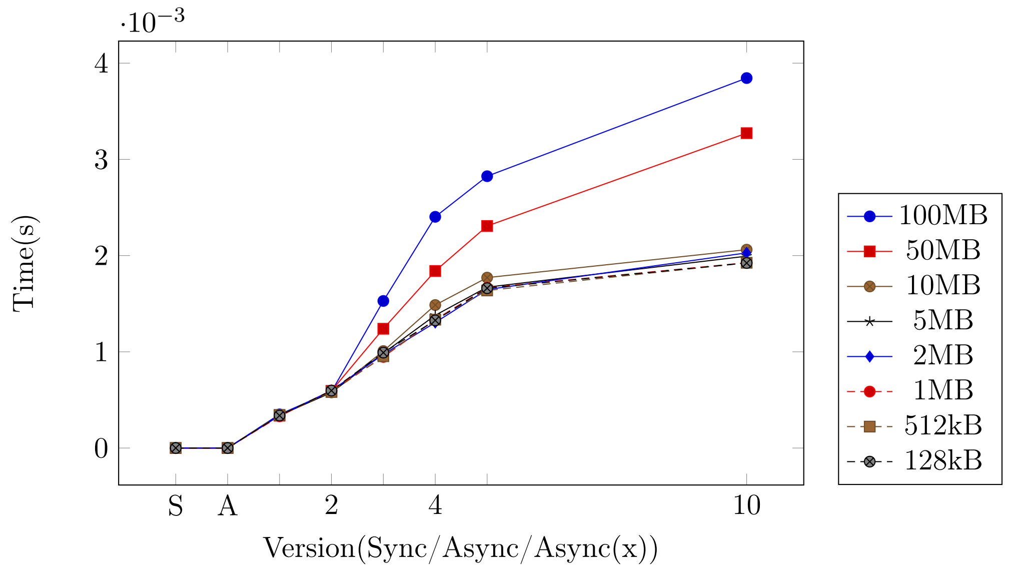


(a) 128kB to 100MB

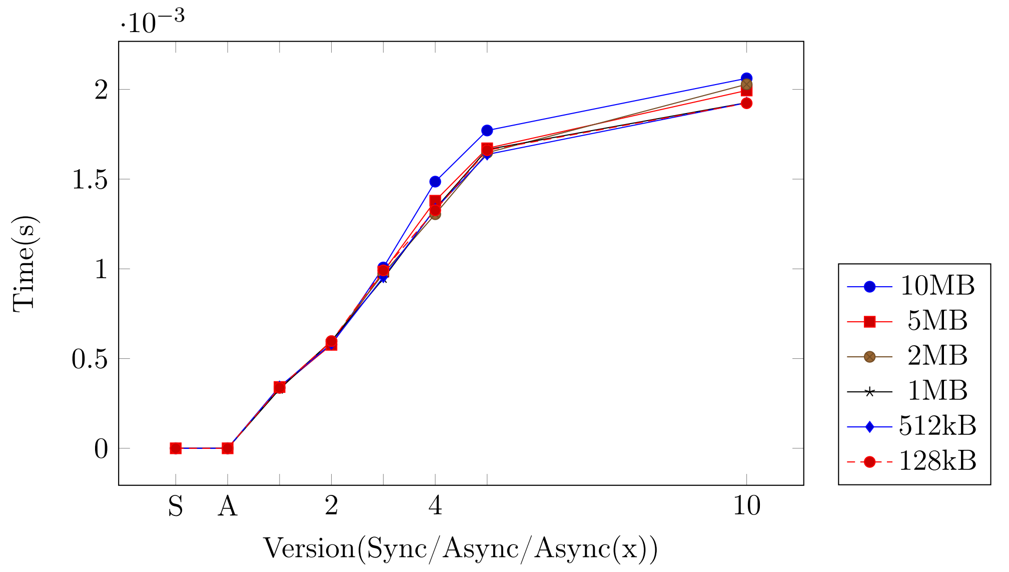


(b) 128kB to 10MB

Figure B.2.: Results micro benchmark new: ICE 2 nodes INTEL40 2MSGs no stripping COMMUNICATION time WAIT only

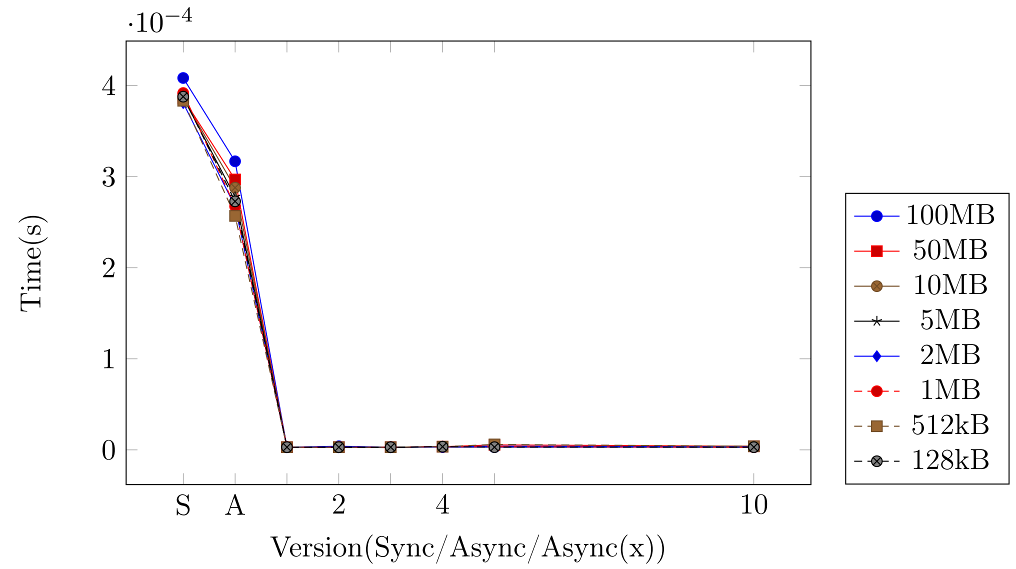


(a) 128kB to 100MB

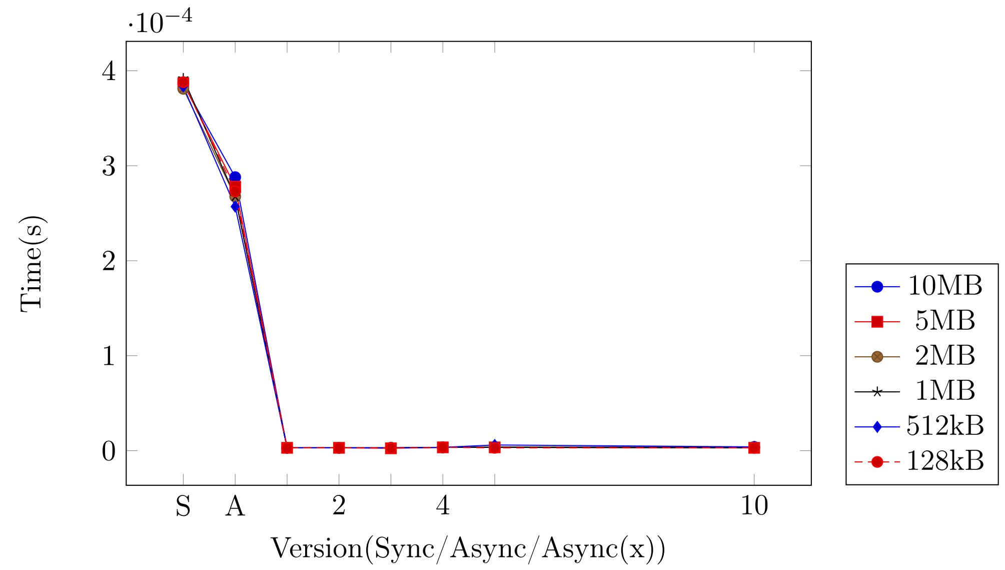


(b) 128kB to 10MB

Figure B.3.: Results micro benchmark new: ICE 2 nodes INTEL40 2MSGs no stripping COMMUNICATION time TEST only zoom



(a) 128kB to 100MB



(b) 128kB to 10MB

Figure B.4.: Results micro benchmark new: ICE 2 nodes INTEL40 2MSGs no stripping COMMUNICATION time INIT only

Bibliography

[1] F. Alessi, P. Thoman, G. Georgakoudis, T. Fahringer, and D. Nikolopoulos. Application-Level Energy Awareness for OpenMP. In *Proceedings of the 11th International Workshop on OpenMP, IWOMP 2015*, pages 219–232, Berlin, Heidelberg, 2015. Springer.

[2] R. Amorim, G. Haase, M. Liebmann, and R. W. dos Santos. Comparing CUDA and OpenGL Implementations for a Jacobi Iteration. In *Proceedings of the International Conference on High Performance Computing Simulation, HPCS '09*, pages 22–32, Washington, DC, USA, 2009. IEEE Computer Society.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[4] D. Bailey, E. Barscz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. NAS technical report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, 1994.

[5] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, page 10 ff., Washington, DC, USA, 2006. IEEE Computer Society.

[6] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham. Performance Analysis of Asynchronous Jacobi's Method Implemented in MPI, SHMEM

and OpenMP. In *International Journal of High Performance Computing Applications*, pages 97–111. Sage Publications, Inc., Thousand Oaks, CA, USA, 2012.

[7] R. A. F. Bhoedjang, T. Ruhl, and H. Bal. User-Level Network Interface Protocols. *Computer*, 31(11):53–60, 1998.

[8] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA Machines. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[10] S. W. Bova and G. F. Carey. A Distributed Memory Parallel Element-by-Element Scheme for Semiconductor Device Simulation. *Computer Methods in Applied Mechanics and Engineering*, 181(4):403 – 423, 2000.

[11] R. Brightwell, R. Riesen, and K. D. Underwood. Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications. In *International Journal of High Performance Computing Applications*, volume 19, pages 103–117. Sage Publications, Inc., Thousand Oaks, CA, USA, 2005.

[12] R. Brightwell, K. Underwood, and R. Riesen. An Initial Analysis of the Impact of Overlap and Independent Progress for MPI. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting, EuroPVM/MPI*, pages 370–377, Berlin, Heidelberg, 2004. Springer.

[13] R. Brightwell and K. D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *Proceedings of the 18th Annual International Conference on Supercomputing, SC 2008*, ICS '04, pages 298–305, New York, NY, USA, 2004. Association for Computing Machinery.

[14] D. Buettner, J. T. Acquaviva, and J. Weidendorfer. Real Asynchronous MPI Communication in Hybrid Codes through OpenMP Communication Tasks. In *Proceedings of the International Conference on Parallel and Distributed Systems, ICPADS*, pages 208–215, Washington, DC, USA, 2013. IEEE Computer Society.

[15] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, Supercomputing '00, page 12, Washington, DC, USA, 2000. IEEE Computer Society.

[16] P. Carribault, M. Perache, and H. Jourdren. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In *Proceedings of the 6th International Workshop on OpenMP, IWOMP*, pages 1–14, Berlin, Heidelberg, 2010. Springer.

[17] J. Cecilia, J. García, and M. Ujaldón. CUDA 2D Stencil Computations for the Jacobi Method. In *Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing, PARA 2010*, pages 173–183, Berlin, Heidelberg, 2012. Springer.

[18] B. M. Chapman, L. Huang, H. Jin, G. Jost, and B. R. de Supinski. Toward Enhancing OpenMP's Work-sharing Directives. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, pages 645–654, Berlin, Heidelberg, 2006. Springer.

[19] W.-Y. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, IPDPS 2009*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[21] G. Da Costa and J.-M. Pierson. Characterizing Applications from Power Consumption: A Case Study for HPC Benchmarks. In *1st International Conference on Information and Communication on Technology for the Fight against Global Warming, ICT-GLOW 2011*, pages 10–17, Berlin, Heidelberg, 2011. Springer.

[22] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, page 58 ff., Washington, DC, USA, 2005. IEEE Computer Society.

[23] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[24] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008*, pages 1–12, Washington, DC, USA, 2008. IEEE Computer Society.

[25] S. Didelot, P. Carribault, M. Pérache, and W. Jalby. Improving MPI communication overlap with collaborative polling. *Computing*, 96(4):1–16, 2013.

[26] H. Dursun, K. I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A Multilevel Parallelization Framework for High-Order Stencil Computations. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 642–653, Berlin, Heidelberg, 2009. Springer.

[27] A. Eichenberger, C. Terboven, M. Wong, and D. an Mey. The Design of OpenMP Thread Affinity. In *Proceedings of the 8th International Workshop on OpenMP, IWOMP 2012*, pages 15–28, Berlin, Heidelberg, 2012. Springer.

[28] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. OpenMPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2004*, pages 97–104, Berlin, Heidelberg, 2004. Springer.

[29] B. Goglin and S. Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. In *Journal of Parallel and Distributed Computing*, pages 176–188. Association for Computing Machinery, New York, NY, USA, 2013.

[30] R. Grabner, F. Mietke, and W. Rehm. Implementing an MPICH-2 Channel Device Over VAPI on InfiniBand. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 184 ff., Washington, DC, USA, 2004. IEEE Computer Society.

[31] R. Graham, S. Poole, P. Shamis, G. Bloch, G. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. ConnectX-2 InfiniBand Management Queues: First Investigation of the New Support for Network Offloaded Collective Operations. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid*, pages 53–62, Washington, DC, USA, 2010. IEEE Computer Society.

[32] G. Hager, G. Jost, and R. Rabenseifner. Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the Cray Users Group Conference*. Cray User Group, Inc., 2009.

[33] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein. Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid

MPI/OpenMP on Current Supercomputing Platforms. In *Proceedings of the Cray Users Group Conference*. Cray User Group, Inc., 2011.

[34] D. S. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC'00, Washington, DC, USA, 2000. IEEE Computer Society.

[35] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 5–12, Washington, DC, USA, 2008. IEEE Computer Society.

[36] C. Iancu, P. Husbands, and P. Hargrove. HUNTing the Overlap. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, PACT 2005*, pages 279–290, Washington, DC, USA, 2005. IEEE Computer Society.

[37] InfiniBand Trade Association. InfiniBand Architecture Volume 1 and Volume 2, 2015. `http://www.infinibandta.org/content/pages.php?pg=technology_public_specification`. [Online, accessed April 3, 2015].

[38] M. Jiayin, S. Bo, W. Yongwei, and Y. Guangwen. Overlapping Communication and Computation in MPI by Multithreading. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA 2006*, pages 52–57. CSREA Press, 2006.

[39] J. Ke, M. Burtscher, and E. Speight. Tolerating Message Latency Through the Early Release of Blocked Receives. In *Proceedings of the 11th International Conference on Parallel Processing*, Euro-Par'05, pages 19–29, Berlin, Heidelberg, 2005. Springer.

[40] C. Keppitiyagama and A. Wagner. Asynchronous MPI Messaging on Myrinet. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, page 8 ff., Washington, DC, USA, 2001. IEEE Computer Society.

[41] M. Kowarschik, U. Rüde, C. Weiß, and W. Karl. Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions. *Computing*, 64(4):381–399, 2000.

[42] R. Kumar, A. Mamidala, M. Koop, G. Santhanaraman, and D. Panda. Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting*, EuroPVM/MPI, pages 185–193, Berlin, Heidelberg, 2008. Springer.

[43] W. Lawry, C. Wilson, A. MacCabe, and R. Brightwell. Comb: A Portable Benchmark Suite for Assessing MPI Overlap. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 472–475, Washington, DC, USA, 2002. IEEE Computer Society.

[44] B. P. Lester. *The Art of Parallel Programming.* Prentice Hall, 1993.

[45] G. Liu and T. S. Abdel-Rahman. Computation-Communication Overlap on Network-of-Workstation Multiprocessors. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, New York, NY, USA, 1998. Association for Computing Machinery.

[46] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC'03, page 58 ff., New York, NY, USA, 2003. Association for Computing Machinery.

[47] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D. Panda, and P. Wyckoff. Microbenchmark Performance Comparison of High-Speed Cluster Interconnects. *IEEE Micro*, 24(1):42–51, 2004.

[48] J. Liu, J. Wu, and D. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS'03, pages 167–198, New York, NY, USA, 2004. Association for Computing Machinery.

[49] R. D. Loft, S. J. Thomas, and J. M. Dennis. Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC'01, pages 18–18, New York, NY, USA, 2001. Association for Computing Machinery.

[50] A. Mamidala, D. Faraj, S. Kumar, D. Miller, M. Blocksome, T. Gooding, P. Heidelberger, and G. Dozsa. Optimizing MPI Collectives Using Efficient Intra-node Communication Techniques over the Blue Gene/P Supercomputer. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW*, pages 771–780, Washington, DC, USA, 2011. IEEE Computer Society.

[51] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 5–16, New York, NY, USA, 2010. Association for Computing Machinery.

[52] K. Mehta, E. Gabriel, and B. Chapman. Specification and Performance Evaluation of Parallel I/O Interfaces for OpenMP. In *Proceedings of the 8th International Workshop on OpenMP, IWOMP 2012*, pages 1–14, Berlin, Heidelberg, 2012. Springer.

[53] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting, EuroPVM/MPI*, pages 104–115, Berlin, Heidelberg, 2009. Springer.

[54] Message Passing Interface Forum. MPI-1.0: A Message-Passing Interface Standard, 1994. `http://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps`. [Online, accessed August 8, 2011].

[55] Message Passing Interface Forum. MPI-3.0: A Message-Passing Interface Standard, 2012. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. [Online, accessed July 13, 2013].

[56] S. Moreaud, B. Goglin, R. Namyst, and D. Goodell. Optimizing MPI Communication within Large Multicore Nodes with Kernel Assistance. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW*, pages 1–7, Washington, DC, USA, 2010. IEEE Computer Society.

[57] J. Nieplocha, V. Tipparaju, M. Krishnan, G. Santhanaraman, and D. K. Panda. Optimisation and Performance Evaluation of Mechanisms for Latency Tolerance in Remote Memory Access Communication on Clusters. In *International Journal of High Performance Computing and Networking, IJHPCN*, pages 198–209, New York, NY, USA, 2004. Association for Computing Machinery.

[58] OpenMP Architecture Review Board. OpenMP application program interface. `http://www.openmp.org`, [Online, accessed January 10, 2016].

[59] OpenMP Architecture Review Board. OpenMP-3.0: Application Programming Interface, 2008. `http://www.openmp.org/mp-documents/spec30.pdf`. [Online, accessed March 30, 2011].

[60] OpenMP Architecture Review Board. OpenMP-3.1: Application Programming Interface, 2011. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`. [Online, accessed August 1, 2011].

[61] OpenMP Architecture Review Board. OpenMP-4.0: Application Programming Interface, 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`. [Online, accessed January 15, 2014].

[62] OpenMP Architecture Review Board. OpenMP-4.5: Application Programming Interface, 2015. `http://www.openmp.org/mp-documents/openmp-4.5.pdf`. [Online, accessed December 1, 2015].

[63] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. Volz, and C. Wong. High-Order Stencil Computations on Multicore Clusters. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, IPDPS 2009*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[64] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Proceedings of the 9th Symposium on High Performance Interconnects*, HOTI '01, page 125 ff., Washington, DC, USA, 2001. IEEE Computer Society.

[65] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella. Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. In *Proceedings of the Cray Users Group Conference*. Cray User Group, Inc., 2012.

[66] M. Pérache, P. Carribault, and H. Jourdren. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting, EuroPVM/MPI*, pages 94–103, Berlin, Heidelberg, 2009. Springer.

[67] M. Pérache, H. Jourdren, and R. Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, page 78–88, Berlin, Heidelberg, 2008. Springer.

[68] R. Rabenseifner. Communication Bandwidth of Parallel Programming Models on Hybrid Architectures. In *Proceedings of the 4th International Symposium on High Performance Computing, ISHPC*, pages 437–448, Berlin, Heidelberg, 2002. Springer.

[69] R. Rabenseifner. Hybrid Parallel Programming on HPC Platforms. `http://www.compunity.org/events/ewomp03/omptalks/Tuesday/Session7/T01p.pdf`, 2003. [Online, accessed Juli 10, 2013].

[70] R. Rabenseifner. Hybrid Parallel Programming: Performance Problems and Chances. In *Proceedings of the Cray Users Group Conference*. Cray User Group, Inc., 2003.

[71] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the Euromicro Conference on Parallel, Distributed, and Network-Based*

Processing, pages 427–436, Washington, DC, USA, 2009. IEEE Computer Society.

[72] R. Rabenseifner and G. Wellein. Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures. In *International Journal of High Performance Computing Applications*, pages 49–62. Sage Publications, Inc., 2003.

[73] H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Lieberman, M. van Waveren, and B. Whitney. Large System Performance of SPEC OMP2001 Benchmarks. In *Proceedings of the 4th International Symposium on High Performance Computing, ISHPC*, pages 370–379, Berlin, Heidelberg, 2002. Springer.

[74] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC'06*, page 17 ff., Washington, DC, USA, 2006. IEEE Computer Society.

[75] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. Panda. Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID '09*, pages 380–387, Washington, DC, USA, 2009. IEEE Computer Society.

[76] D. Schmidl, T. Cramer, C. Terboven, D. Mey, and M. Müller. An OpenMP Extension Library for Memory Affinity. In *Proceedings of the 10th International Workshop on OpenMP, IWOMP*, pages 103–114, Berlin, Heidelberg, 2014. Springer.

[77] A. G. Shet, P. Sadayappan, D. E. Bernholdt, J. Nieplocha, and V. Tipparaju. A Framework for Characterizing Overlap of Communication and Computation in Parallel Applications. *Cluster Computing*, 11(1):75–90, 2008.

[78] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing. In *Proceedings of the ACM/IEEE 2001 Conference on Supercomputing, SC'01*, page 49 ff., Washington, DC, USA, 2001. IEEE Computer Society.

[79] L. Smith and M. Bull. Development of Mixed Mode MPI/OpenMP Applications. In *Journal on Scientific Programming*, pages 83–98. IOS Press, Amsterdam, The Netherlands, 2001.

[80] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi. Identifying the Capability of Overlapping Computation with Communication. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, PACT '96, page 133 ff., Washington, DC, USA, 1996. IEEE Computer Society.

[81] V. Strumpen and T. Casavant. Exploiting Communication Latency Hiding for Parallel Network Computing: Model and Analysis. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 622–627, Washington, DC, USA, 1994. IEEE Computer Society.

[82] V. Subotic, J. Sancho, J. Labarta, and M. Valero. The Impact of Application's Micro-Imbalance on the Communication-Computation Overlap. In *Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP*, pages 191–198, Washington, DC, USA, 2011. IEEE Computer Society.

[83] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 32–39, New York, NY, USA, 2006. Association for Computing Machinery.

[84] C. Terboven, D. Schmidl, T. Cramer, and D. an Mey. Assessing OpenMP Tasking Implementations on NUMA Architectures. In *Proceedings of the 8th International Workshop on OpenMP, IWOMP*, pages 182–195, Berlin, Heidelberg, 2012. Springer.

[85] X. Teruel, M. Klemm, K. Li, X. Martorell, S. Olivier, and C. Terboven. A Proposal for Task-Generating Loops in OpenMP. In *Proceedings of the 9th International Workshop on OpenMP, IWOMP*, pages 1–14, Berlin, Heidelberg, 2013. Springer.

[86] R. Thakur and W. Gropp. Open Issues in MPI Implementation. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems and Architecture, ACSAC*, pages 327–338, Berlin, Heidelberg, 2007. Springer.

[87] R. Thakur and W. Gropp. Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In *Proceedings of the 14th European PVM/MPI Users' Group Meeting, EuroPVM/MPI*, pages 46–55, Berlin, Heidelberg, 2007. Springer.

[88] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the 39th International Conference on Parallel Processing Work-*

shops, ICPPW'10, pages 207–216, Washington, DC, USA, 2010. IEEE Computer Society.

[89] R. F. van der Wijngaart and H. Jin. NAS Parallel Benchmarks, Multi-Zone Versions. NAS technical report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003.

[90] H. D. Vasava and J. M. Rathod. Software Based Distributed Shared Memory (DSM) Model Using Shared Variables Between Multiprocessors. In *Proceedings of the International Conference on Communications and Signal Processing, ICCSP*, pages 1431–1435, Washington, DC, USA, 2015. IEEE Computer Society.

[91] S. Venkatasubramanian and R. W. Vuduc. Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. Association for Computing Machinery.

[92] B. Wang, D. Schmidl, and M. Müller. Evaluating the Energy Consumption of OpenMP Applications on Haswell Processors. In *Proceedings of the 11th International Workshop on OpenMP, IWOMP*, pages 233–246, Berlin, Heidelberg, 2015. Springer.

[93] G. Wellein, G. Hager, A. Basermann, and H. Fehske. Fast Sparse Matrix-Vector Multiplication for TeraFlop/s Computers. In *Proceedings of the 5th Conference on High Performance Computing for Computational Science, VECPAR 2002*, pages 287–301, Berlin, Heidelberg, 2003. Springer.

[94] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, COMPSAC '09, pages 579–586, Washington, DC, USA, 2009. IEEE Computer Society.

[95] S. Whalen, S. Engle, S. Peisert, and M. Bishop. Network-Theoretic Classification of Parallel Computation Patterns. In *International Journal of High Performance Computing Applications*, pages 159–169. Sage Publications, Inc., Thousand Oaks, CA, USA, 2012.

[96] J. White III and S. Boya. Where's the Overlap? - An Analysis of Popular MPI Implementations. In *Proceedings of the 3rd MPI Developer's and User's Conference*, Atlanta, Georgia, USA, 1999. MPI Software Technology Press.

[97] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific Computing Kernels on the Cell Processor. In *International Jour-*

nal of Parallel Programming, pages 263–298. Kluwer Academic Publishers, Norwell, MA, USA, 2007.

[98] T. Woodall, G. Shipman, G. Bosilca, R. Graham, and A. Maccabe. High Performance RDMA Protocols in HPC. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting, EuroPVM/MPI*, pages 76–85, Berlin, Heidelberg, 2006. Springer.

[99] W. Yu, D. Buntinas, R. Graham, and D. Panda. Efficient and Scalable Barrier over Quadrics and Myrinet with a new NIC-based Collective Message Passing Protocol. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 182 ff., Washington, DC, USA, 2004. IEEE Computer Society.

[100] W. Yu, D. Buntinas, and D. Panda. High Performance and Reliable NIC-based Multicast over Myrinet/GM-2. In *Proceedings of the 2003 International Conference on Parallel Processing, ICPP'03*, pages 197–204, Washington, DC, USA, 2003. IEEE Computer Society.