

The present work was submitted to the
Chair for High Performance Computing,
IT Center, RWTH Aachen

Investigating Scheduling and Load Balancing Characteristics of Task-based Programming Models for Hybrid HPC Applications

Master Thesis

Simon Convent
Student ID no.: 345104

Aachen, September 26th, 2019

Communicated by Prof. Dr. Matthias S. Müller

First examiner: Prof. Dr. Matthias S. Müller (*)
Second examiner: Prof. Dr. Michael Bader (**)
Advisor: Jannis Klinkenberg, M.Sc. (*)

(*) Chair for High Performance Computing,
IT Center, RWTH Aachen University
(**) Department of Informatics,
Technical University of Munich

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published and unpublished scripts are indicated as such. The paper has not been previously presented as an examination paper in any other form.

Aachen, September 26th, 2019

Abstract

In hybrid HPC applications, load imbalances that can be caused by either hardware or software, may have a significant performance impact. Software-induced load imbalances can be caused by the operating system or by applications with changing load distributions such as Adaptive Mesh Refinement algorithms. Unpredictable latencies in complex memory hierarchies or clock frequency boosts are causes for hardware-induced load imbalances. Using a task-based programming paradigm is an approach to dynamically achieve dynamic load balancing. There are established task-based frameworks for shared memory environments, but frameworks for distributed memory environments face the additional challenge of balancing the load across process boundaries. The goal of this thesis is to analyze and compare different scheduling and load balancing mechanisms used in task-based frameworks.

Three frameworks were considered: Charm++, StarPU and Chameleon. Charm++ is a message-driven framework that uses migratable objects. StarPU provides task-based programming capabilities with an a-priori cross-rank load balancing mechanism. Chameleon is a framework designed for fine-granular, reactive load balancing using task migration.

Using two benchmarks, empirical experiments were performed to test the frameworks scalability, their reaction to hardware- and software-induced load imbalances and the effects of task-granularity on the performance. It was found that Chameleon offers the most fine-granular load balancing capability, but also introduces overhead with a dedicated communication thread. Chameleon was superior to the other frameworks in experiments with hardware-induced load imbalances and achieved a speedup of up to 1.15 compared to the baseline implementation. Charm++ was found to be the most versatile framework with a range of load balancing strategies. It performed best in experiments with software-induced load imbalances in an iterative application where it achieved a speedup of up to 1.36 compared to the baseline implementation. StarPU was unable to provide the desired load balancing properties.

Contents

List of Figures	ix
1 Introduction	1
1.1 Motivation	1
1.2 Task-Based Programming	3
1.3 Focus of the Thesis	3
1.4 Structure of the Thesis	4
2 Background	5
2.1 Modern Processors	5
2.2 Message Passing Interface (MPI)	6
2.3 OpenMP	7
2.4 Task-Based Programming in Distributed Memory Environments	9
3 Related Work	11
3.1 Online Scheduling Algorithms	11
3.2 Apache Hadoop and Apache Spark	12
3.3 Partitioned Global Address Space	13
4 Scheduling and Load Balancing Characteristics	15
4.1 Load Balancing Strategy Categories	15
4.1.1 Global Load Balancing	15
4.1.2 Local Load Balancing	16
4.1.3 Hierarchical Load Balancing	16
4.2 Charm++ Framework	16
4.2.1 Scheduling	19
4.2.2 Load Balancing	20
4.3 StarPU	21
4.3.1 Scheduling	22
4.3.2 Load Balancing	23
4.4 Chameleon Framework	23
4.4.1 Scheduling	25
4.4.2 Load Balancing	26
4.5 Performance Model	27
4.6 Framework Comparison	27

5	Benchmarks	31
5.1	Pi Benchmark	31
5.1.1	MPI+OpenMP Version	31
5.1.2	Charm++ Version	32
5.1.3	Chameleon Version	33
5.1.4	StarPU Version	34
5.2	Shallow Water Equations Benchmark	34
5.2.1	Mathematical Model	35
5.2.2	Discretization	35
5.2.3	Partitioning	36
5.2.4	MPI+OpenMP Version	38
5.2.5	Charm++ Version	38
5.2.6	Chameleon Version	39
5.2.7	StarPU Version	39
6	Evaluation	41
6.1	Environment	41
6.2	Methodology	41
6.3	Pi Benchmark Experiments	43
6.3.1	Strong Scaling	43
6.3.2	Work-Induced Imbalances	43
6.3.3	Interference	45
6.3.4	Interference (Single Core)	46
6.3.5	Granularity	46
6.4	SWE Benchmark Experiments	48
6.4.1	Strong Scaling	48
6.4.2	Work-Induced Imbalances	48
6.4.3	Interference	50
6.4.4	Interference (Single Core)	51
6.4.5	Granularity	52
7	Conclusions and Future Work	55
7.1	Future Work	56
	Acknowledgements	57
	Bibliography	59

List of Figures

1.1	Adaptive 3D solution to the tumor angiogenesis problem	2
2.1	Architecture of Intel Xeon Scalable Processors	6
2.2	MPI code example	7
2.3	OpenMP worksharing example	8
2.4	OpenMP tasking example	8
2.5	Example executions of a workload	10
3.1	Example of a Spark Lineage Graph	13
4.1	Basic Charm++ Ping Pong code example	18
4.2	Charm++ example setup	19
4.3	StarPU Tasking example	22
4.4	Chameleon Tasking example	25
4.5	Example setup of a Chameleon program during runtime	26
5.1	Approximation of π using an integral	32
5.2	SWE simulation example	34
5.3	Cross section of the simulation scenario	35
5.4	Overview of the block-based decomposition of the grid	37
5.5	Overview of SWE simulation steps	38
5.6	Overview over distribution of blocks to ranks	40
6.1	Example execution of interference program	42
6.2	Strong scaling behaviour of frameworks (Pi)	43
6.3	Reaction to work-induced imbalances of frameworks (Pi)	44
6.4	Reaction to interference (Pi)	45
6.5	Reaction to single-core interference (Pi)	46
6.6	Reaction to task granularity of frameworks (Pi)	47
6.7	Scaling behaviour of frameworks (SWE)	48
6.8	Reaction to load imbalances of frameworks (SWE)	49
6.9	Reaction to interference (SWE)	51
6.10	Reaction to single-core interference (SWE)	52
6.11	Effects of task granularity on performance of frameworks (SWE)	53

1 Introduction

1.1 Motivation

Modern HPC applications are usually executed in distributed memory environments. In recent years, processor architectures[9] have increased in complexity and applications use a higher numbers of processors at the same time which poses new challenges. Applications want to use of the computing resources as efficiently as possible to maximize performance. The more processors are involved in a computation, the more important it becomes to have a good load balance to avoid idle time. Modern computers introduce numerous possibilities of slowdowns that may cause load imbalances. Processors are clocked at a base frequency but may increase its frequency dynamically depending on the number of cores used, the kinds of instruction used and current thermal conditions. The increase and decrease of the processor frequencies are transparent for the application and cannot be influenced which results in a less predictable performance of the processor. The memory hierarchy of processors becomes more complex with new technologies such as High-Bandwidth memory [21] and the use of more sophisticated caches. As a result, memory access latencies become more unpredictable resulting in potential load imbalances. Modern operating systems can handle many cores and processes simultaneously including I/O and main memory management. But the operating system itself may also need processor time for internal bookkeeping or cleanup [5]. Thus, the operating system may need to use one core for some time in which it is unavailable to the application. HPC clusters have many users executing different applications at the same time. Modern schedulers allow for nodes to be shared between applications, i.e. that some cores and part of the memory are used by one application and the rest by another application. The performance of each application depends on the other one since they may share resources such as main memory bandwidth, caches or network bandwidth and the thermal conditions may be influenced resulting in more unpredictable core frequencies.

In addition to these hardware-induced load imbalances, there may also be software-induced load imbalances. One example are HPC applications with adaptive loads, where the computational load dynamically changes over time and cannot be predicted. One example for such applications are Adaptive Mesh Refining algorithms [4]. These algorithms dynamically adapt the mesh granularity during execution and the adaption depends on the input data. Thus, it is not possible to predict the work imbalance beforehand, but has to be done dynamically during execution.

1 Introduction

Figure 1.1 is a depiction of an intermediate state of a simulation using an adaptive mesh refinement algorithm. The simulation tracks the flow of a chemical around a tumor and some areas need to be simulated more fine-granular than others. Each cube represents a similar computational load regardless of its size. Because the refining is performed adaptively at runtime, the load distribution is unknown upfront and a static partitioning of the domain among the processors may result in imbalances.

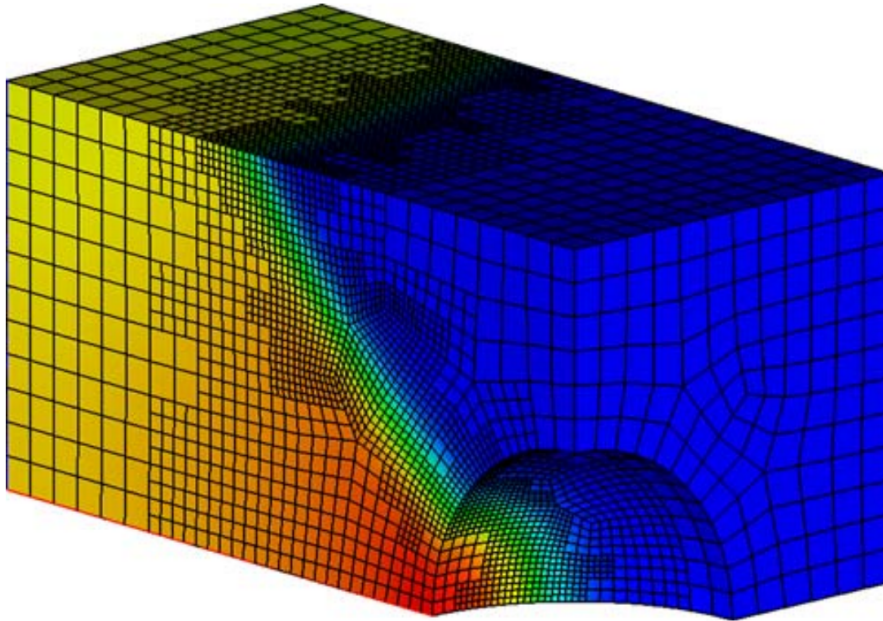


Figure 1.1: Adaptive 3D solution to the tumor angiogenesis problem using an adaptive mesh refining algorithm[12].

All of these unpredictable imbalances may result in a global performance decrease because all processors need to wait on the slowest processor at synchronization points, e.g. in a barrier. A global repartitioning of the data is a solution to achieve a good load balance, but is too expensive to execute as frequently as needed to compensate for fine-granular imbalances. Another solution to address imbalances are task-based programming models. Coarse-grained load balancing must be performed by the applications themselves but more fine-grained and unpredictable load imbalances may be handled in a more automatic fashion using a task-based programming framework.

All in all there is a need for task-based frameworks in modern HPC applications, that feature fine-granular dynamic load balancing capabilities. It is desirable to identify differences between these frameworks regarding their scheduling and load balancing characteristics and to identify possible usecases for which the frameworks are particularly well suited.

1.2 Task-Based Programming

The general idea of task-based programming is to split the workload into small pieces and execute them asynchronously using a set of workers. This approach is opposed to the common approach of statically partitioning the workload among the workers. In a task-based environment, each worker takes a task from a task queue, executes it and repeats these steps until all tasks have been processed. Because the distribution of the workload to processors is not static but dynamic, load balancing is achieved automatically given a small enough granularity of the tasks. Additionally, some algorithms are specifically suited for a task-based programming approach because they cannot be parallelized with loop-based approaches. One example for such an algorithm is Breadth-First graph search where the structure of the graph influences the order in which the nodes are visited and the overall number of visited nodes is not known beforehand. In general, a task consists of data and a function that is to be executed on that data. The frameworks implementing this paradigm however diverse in their usage and focus on different features. For shared memory environments, task-based programming has already been well-established with technologies such as OpenMP. In distributed memory environments however, additional challenges arise, such as inter-process task migration and load balancing. Three frameworks that provide solutions to these challenges and enable task-based programming in distributed memory environments are Charm++, StarPU and Chameleon.

1.3 Focus of the Thesis

It is the goal of this thesis to investigate scheduling and load balancing characteristics of these three frameworks.

More specifically, the following properties and their effect on performance are to be investigated:

1. **Scaling:** How well do the frameworks scale? What are limiting factors?
2. **Load Balancing Strategies:** What are the load balancing strategies used by the frameworks? What are their advantages and disadvantages?
3. **Reaction to Hardware-Induced Imbalances:** How do the frameworks handle imbalances caused by hardware?
4. **Reaction to Software-Induced Imbalances:** How do the frameworks handle imbalances caused by software? How large imbalances can the frameworks cope with? What are limiting factors?
5. **Granularity:** How does task size affect performance? Small tasks enable a more fine-granular load balancing but introduce overhead on the other hand. What is a good tradeoff?

6. **Suitability:** Which frameworks are suitable for which kinds of applications? For which kinds of applications are they not suitable and why?

To analyze the frameworks with regard to these properties, a theoretical analysis based on literature and documentation is conducted as well as an empirical analysis using two benchmarks.

1.4 Structure of the Thesis

The rest of this thesis is structured as following:

In chapter 2, there will be an introduction to basic concepts and technologies used in the thesis including MPI and OpenMP.

Then, an overview of related work is given in chapter 3. The focus here lies on technologies and concepts with similar challenges such as Online Load Balancing in general, Big-Data frameworks and PGAS frameworks.

In chapter 4, the three frameworks Charm++, StarPU and Chameleon are introduced. This chapter includes a theoretical comparison of the frameworks with regards to the properties described in the previous section.

To further underline the results of the theoretical analysis, empirical experiments using two benchmarks are conducted. These benchmarks are presented in chapter 5. One benchmark is synthetic and calculates an estimation of π . The other is a real-world application simulating wave propagation. Both have been ported to use each of the three frameworks considered in this thesis as well as a baseline version using MPI and OpenMP.

The experiments are conducted with different setups to analyze scaling behavior, reactions to imbalances and effects of different task granularities. The results as well as their interpretation are given in chapter 6.

Finally, chapter 7 offers conclusions drawn from the analysis and gives an overview of possible future work.

2 Background

This chapter provides some background information on concepts and technologies that are central to this thesis and will be referred to in the following chapters.

2.1 Modern Processors

The processors used in HPC clusters are constantly evolving and growing more and more complex. One observable trend in recent years has been the increase of the number of cores per processor [9]. The industry is determined to follow Moore's law and since the clock frequency of modern processors does not grow significantly anymore, the number of cores per chip increases steadily. Modern processors also have a boosting feature. If the temperature of the processor is low or not all cores are used, it may increase its clock frequency. The times when a processor boosts its frequency are not deterministic and offer limited control. Since these boost times cannot be controlled or predicted (especially in multi-user operation), some processors may execute workloads faster than others resulting in hardware-induced load imbalances.

Figure 2.1 gives an overview of a modern CPU, in this case an Intel Xeon Scalable Processor architecture. It consists of up to 28 cores (named Xeon in the figure), each of which has its own L1 cache integrated into the core. Additionally, each core has an L2 cache and an L3 cache that can also be used by other cores. To communicate, the cores are connected in a mesh, as depicted. Modern processors typically have multiple memory channels, in this case 6, split into 3 on each side.

As mentioned earlier in this section, there is a trend of a decreasing structure size in modern processors. This also means that small imperfections in the manufacturing process may induce a slight performance disadvantage compared to others. This may be due to a slightly higher power consumption and thus fewer boost times.

Another recent development is the use of High-Bandwidth-Memory (HBM) [21] or NVRAM [22], which increases the memory bandwidth but also increases the complexity of the memory hierarchy. This effect may also be a reason why some processors are faster than others leading to a load imbalance.

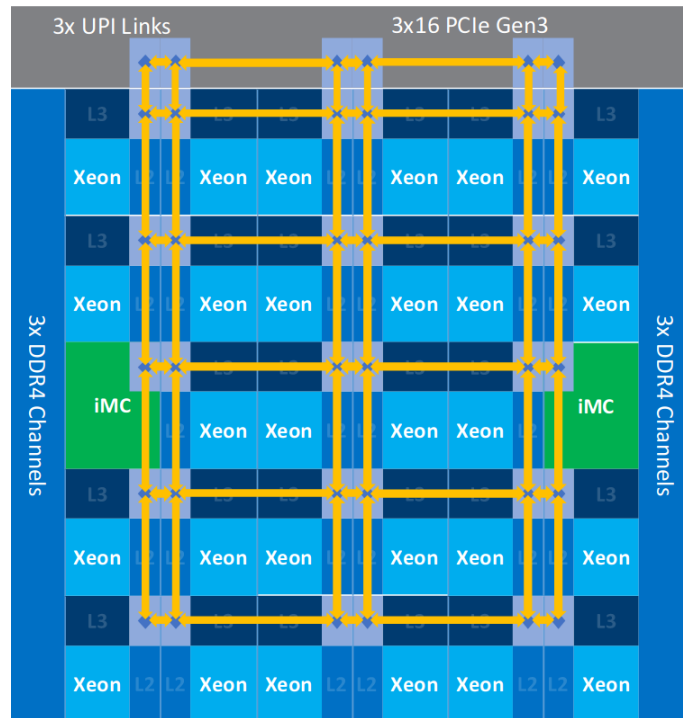


Figure 2.1: Architecture of Intel Xeon Scalable Processors [10]

2.2 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [15] is a standard to enable parallel programming for distributed-memory environments. It defines a set of routines to explicitly pass messages between processes. These processes are called ranks. There are multiple implementations of the standard that differ in performance and platform-specific optimizations. Some are designed to be generic and some are designed to achieve high performance on certain interconnects or certain models of processors.

When an MPI program is executed, the same binary is launched multiple times resulting in separate processes that may run on separate nodes. MPI provides routines that enable these processes to communicate and synchronize.

Generally there are three kinds of communication: Point-to-Point communication involves only two ranks where one rank sends a message that is received by the other. Collective Communication on the other hand involves all ranks within a communicator. One example is a broadcast operation, where one rank sends data to all other ranks. The third kind of communication is One-Sided communication, where a rank directly accesses the main memory of another rank.

Figure 2.2 provides an example of an MPI program using Point-to-Point communication. Before using the communication routines, MPI needs to be initialized and the overall number of ranks as well as the number of the current rank is usually fetched (lines 2-4).

The example is designed to be executed with two ranks. The ranks then alternate with sending and receiving messages (in this case just an integer) between each other until a certain number of messages has been sent. A call to `MPI_Send()` sends a message to the specified rank and does not return until that rank has called a corresponding `MPI_Recv()`. Communicators are a construct used to specify groups of ranks and this code uses the standard communicator `MPI_COMM_WORLD`, which includes all ranks that take part in the current execution.

```

1  int myRank, numRanks;
2  MPI_Init(&argc, &argv);
3  MPI_Comm_size(MPI_COMM_WORLD, &numRanks);
4  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
5
6  int otherRank = (myRank + 1) % 2;
7  int count = 0;
8  int message = 123;
9  while(count <= MAX_COUNT) {
10     if(count % 2 == myRank) {
11         MPI_Send(&message, 1, MPI_INT, otherRank,
12             0, MPI_COMM_WORLD);
13     }
14     else {
15         MPI_Recv(&message, 1, MPI_INT, otherRank,
16             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
17     }
18     count++;
19 }
20
21 MPI_Finalize();

```

Figure 2.2: MPI code example

2.3 OpenMP

OpenMP [18] is a language extension to the C/C++ and Fortran programming languages which enables directive-based parallel programming on shared-memory systems. Generally, OpenMP supports two major concepts of parallelization: work-sharing and tasking.

Worksharing is the distribution of work among threads. Typically the iterations of a for-loop are distributed using a single directive. By default, static scheduling is used, i.e. the iterations are statically distributed among the threads and the distribution cannot be changed dynamically. This type of scheduling has the advantage that it is simple and does not need any communication between the threads. However, it has the disadvantage that it does not support any loadbalancing, which is important if some iterations have a higher computational load than others. OpenMP also provides dynamic scheduling. Here, the iterations are grouped into chunks and each thread executes chunks until all chunks have been processed. The distribution of chunks to threads is nondeterministic and involves communication between threads but can handle load imbalances more effectively.

2 Background

Figure 2.3 is an OpenMP example code which scales an array with a factor 2. In line 3 is the OpenMP directive which creates a parallel region (*parallel*) and then divides the iterations of the loop (*for*) among the threads of the parallel region. Worksharing constructs are well-suited for the parallelization of codes using for-loops where the iteration count can be computed before the loop. Operations on vectors and matrices are one example for a usecase of worksharing constructs.

```
1 float* a; // Array
2
3 #pragma omp parallel for
4 for(i = 0; i < LENGTH; i++) {
5     a[i] *= 2;
6 }
```

Figure 2.3: OpenMP worksharing example

The other parallelization paradigm is tasking. Here, tasks are created and workers executed the tasks asynchronously following the task-based programming paradigm. An explicit data mapping is not necessary for standard tasks since OpenMP is intended for shared memory systems.

Figure 2.4 is an OpenMP tasking code example. First, a parallel region is created and then the *single* construct is used. Code within this construct is executed only by one of the threads of the parallel region. Two tasks are created, each of which prints some text. The code within the tasks is executed asynchronously, i.e. they are put into a task queue and the threads of the parallel region will execute them once they are idle. There are no guarantees about the order of the execution of the tasks. There is an implicit barrier at the end of the parallel region which gives the guarantee that all tasks have been executed when the parallel region is left.

Tasking in OpenMP is well-suited for algorithms that do not have a fixed iteration count, such as Breadth-First search on graphs. Additionally, tasking may be used to achieve automatic load balancing.

```
1 #pragma omp parallel
2 {
3     #pragma omp single
4     {
5         #pragma omp task
6         {printf("first task\n");}
7         #pragma omp task
8         {printf("second task\n");}
9     }
10 }
```

Figure 2.4: OpenMP tasking example

2.4 Task-Based Programming in Distributed Memory Environments

In the previous chapter, reasons for hardware- and software-induced load imbalances were given. They demand load balancing capabilities that are not static but dynamic. I.e. the load is not just distributed at the beginning, but is dynamically redistributed during execution in a fine-granular fashion. One solution is to use a task-based programming library. Technologies such as OpenMP [18] or Thread Building Blocks [20] already provide a way to use task-based parallel programming in shared memory systems. But for some HPC applications, shared memory systems do not provide the necessary computational capacity or not enough main memory. Task-based programming in distributed memory poses some challenges, however. In a shared memory system, there is no need for explicit data mapping, as the tasks are going to be executed in the same main memory space as they were created in. This may not be the case in a distributed memory setup, where the processes running on different nodes do not share the same main memory space. Thus, all tasks need to have an explicit data mapping specification at its creation.

When the runtime of the tasking library decides to migrate a task, its data needs to be explicitly sent from one node to another as well as a specification which function is to be executed on that data. Again, a tasking library in a shared memory environment does not need to do this as all tasks are already in the same main memory space.

The more nodes are involved in a program running in distributed memory, the more complex the load balancing decisions become. On a single node, i.e. a shared memory system, load balancing is straight forward as all threads repeatedly pop a task from the task queue and execute it. On more than one node however, in addition to the task execution, task migration decisions and the actual migrations need to be performed. These steps involve inter-node communication, which cause latencies.

In more advanced libraries it is also possible to specify dependencies between tasks, i.e. that certain tasks have to be completed before other tasks are allowed to be executed. This feature creates another level of complexity in distributed memory setups as tasks may depend on tasks executed on a different node which needs additional explicit communication between nodes.

To further illustrate the load balancing advantages of task-based programming in distributed memory, figure 2.5 depicts two scenarios: the initial situation and the desired goal. The top scenario is the execution of a workload using a traditional parallel program without dynamic load balancing. In both work phases, some ranks have finished their workload significantly earlier than the rank that finished last. This results in unused resources. As explained in the motivation section, there are numerous possible reasons for this behavior, namely software or hardware induced imbalances.

2 Background

The bottom scenario depicts an execution of the same workload using a task-based parallel program with task migration. The workload of each rank is split into multiple tasks which are then executed asynchronously and are migrated to ranks with a lower load.

As depicted in the figure, the total execution time is lower with the task-based program since the same workload is distributed more efficiently over the ranks resulting in an ideal use of the computing resources.

Note however, that this is an idealized setup which does not account for communication and task creation overhead. Also, it is not always possible to achieve an ideal schedule because the load imbalances are not known beforehand.

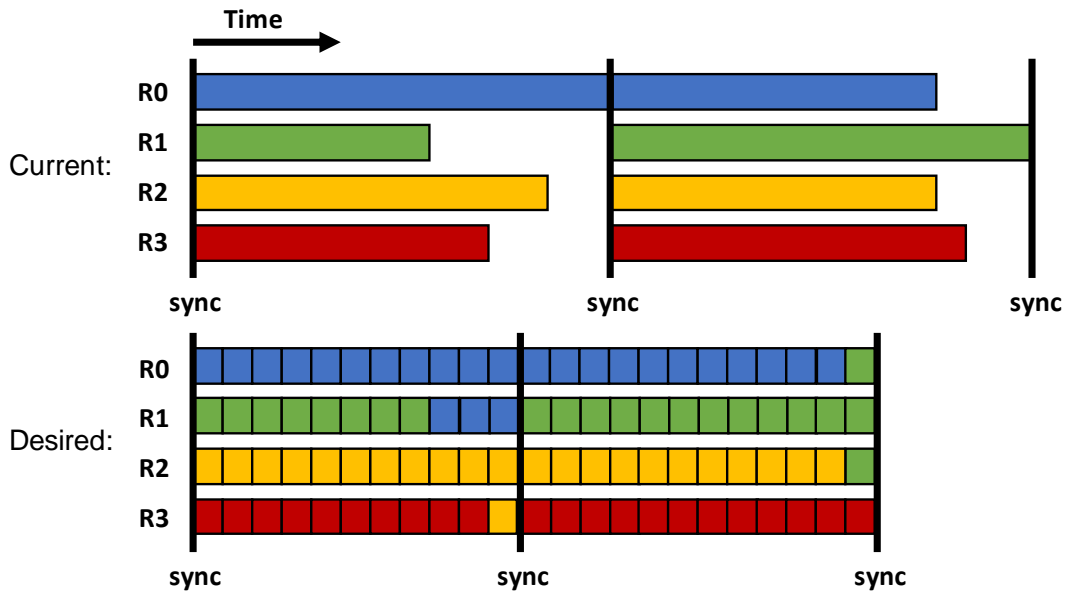


Figure 2.5: Example execution of a workload (with imbalances) by a traditional parallel program (top) and the same by a task-based parallel program with task migration (bottom)

3 Related Work

There has already been research on topics related to scheduling, load balancing and programming in distributed memory environments without explicit communication. This chapter gives a brief overview over related work.

3.1 Online Scheduling Algorithms

The Job Shop Scheduling Problem [14] describes the problem of assigning a given set of jobs to a given number of machines in order to achieve a schedule with a minimal makespan (i.e. a minimal time in which all tasks are executed). More specifically, the flexible Job Scheduling Problem is considered here, where any job can be executed on any machine. This problem has already been studied and has been proven to be NP-hard [1]. The problem was originally studied to optimize the makespan of jobs executed on physical machines in factories but has since been applied to a range of other usecases. The problem of scheduling tasks to nodes is one such usecase.

However, the traditional Job Shop Scheduling Problem assumes that all jobs and their cost (i.e. processing time) are known upfront. That is not necessarily the case when using task-based parallel programming. Since there are also other usecases where this is not the case, the so-called online version of this problem has been studied as well. Here, not all jobs are known upfront but are added during execution. There is no algorithm that is guaranteed to produce optimal schedules for the online version of the problem. Usually, when studying online algorithms, a competitive analysis is conducted which compares the online to the offline version and gives the ratio between the optimal offline solution and the optimal online solution in a worst-case scenario. For the Job Shop Scheduling problem, the online version is e -competitive [3] which means that in the worst case, an optimal algorithm for the online version can only achieve a minimal makespan that is e (Euler's number) times as great as the optimal makespan for the offline version.

For the usecases in this thesis, the tasks are all submitted almost simultaneously before they are executed which allows the scheduler to calculate a better schedule. But there are usecases, where tasks are also created dynamically with one example being Breadth-First search on graphs where tasks are created iteratively whenever a new node is visited. Here, the makespan will be closer towards the upper boundary of e times the optimal makespan.

Generally, the more tasks are created upfront and not dynamically, the better the tasks can be scheduled. However, even if all tasks are created upfront, the scheduler may not achieve an optimal schedule due to the fact, that the problem is NP-complete and the schedulers use heuristic algorithms.

3.2 Apache Hadoop and Apache Spark

In Big Data applications, frameworks such as Apache Hadoop[25] and Apache Spark[26] are used. Hadoop applications follow the Map-Reduce paradigm and the Hadoop runtime splits the Map and Reduce steps into independent tasks which are then executed by the worker processors. During execution of a Map Reduce job, the Hadoop runtime scheduler assigns tasks to nodes which execute them asynchronously.

Apache Spark was developed to address some of the shortcomings of Hadoop, for example the fact that Hadoop Map-Reduce jobs always write their output to the file system and subsequent jobs must read from the file system and cannot use cached in-memory data. Additionally, Spark support stream processing, i.e. continuous input and output while Hadoop does not support it. Apache Spark uses so-called Resilient Distributed Datasets (RDDs), which are immutable collections of key-value pairs stored in a distributed and fault-tolerant way. During the run of a Spark application, data is loaded into initial RDDs first and then subsequent RDDs are calculated using operators on previous RDDs. The RDDs and the operators form the so-called lineage graph of a Spark application. RDDs are partitioned and the partitions are distributed among multiple nodes. When an RDD is evaluated, the runtime creates a task for each of its partitions and then assigns tasks to nodes on which the needed partitions reside.

Figure 3.1 is an example of a lineage graph with four RDDs and a Map and Join operator. Each of the RDDs has 3 partitions and the arrows depict dependencies between these partitions. However, in a real application, the number of partitions per RDD is significantly higher to allow a better load balancing with many nodes as there is a task for each partition of RDDs that needs to be evaluated.

One major difference between these Big Data frameworks and task-based programming in general is the fact that Big Data frameworks also take the location of the data needed for the tasks into account. For Hadoop applications data is usually stored in an instance of the Hadoop Distributed Filesystem(HDFS), which stores files block-wise and distributes these blocks among the nodes of the filesystem. Spark applications usually store the RDDs in-memory but may also use the HDFS.

Each created task in a Hadoop or Spark application contains information on which data it needs to access and the scheduler will try to assign tasks to nodes which have a copy of the data needed. The task-based programming frameworks used in this thesis work in-memory and thus do not need the functionality of assigning tasks to nodes based on their I/O data dependencies.

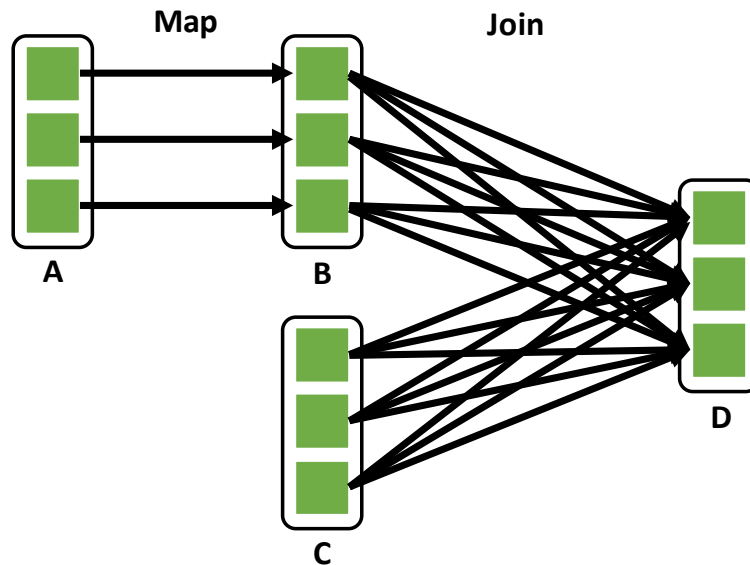


Figure 3.1: Example of a Spark Lineage Graph with four RDDs and a Map and Join operator

3.3 Partitioned Global Address Space

Task-based programming in shared-memory has already been well established with technologies such as OpenMP tasking [18]. Task-based programming in distributed-memory however is not as well established. To be able to develop applications that can be executed in a distributed memory environment but without the need for explicit communication by the application, the concept of Partitioned Global Address Space (PGAS) [6] was developed. PGAS programs have a global main memory address space that is partitioned among the processes, each of which owns a local part of the global address space. Each process can access the global main memory space and therefore has direct access to the main memory on remote processes. In this way, the inter-node communication is hidden from the application and instead is performed by the framework. A node-local main memory access is faster than a remote memory access due to latency and bandwidth limitations of inter-node communication. Thus, for better performance, remote main memory accesses should be avoided.

Task-based parallel programming in distributed memory environment and PGAS have similarities as they both are a solution to porting shared memory programs to a distributed memory environment while avoiding explicit inter-node communication. For both approaches, the inter-node communication can become a performance bottleneck and PGAS frameworks are a more general approach and do not require

3 Related Work

applications to use task-based programming. But on the other hand, the PGAS concept does not include the automatic load balancing capabilities that most task-based programming frameworks offer. A load balancing mechanism would have to be implemented by the application programmer.

4 Scheduling and Load Balancing Characteristics

There are different approaches to realize task-based programming in distributed memory environments. To give an overview of these approaches, several scheduling and load balancing techniques are presented in this chapter. They are analyzed and compared with respect to their characteristics and suitability for certain usecases. Frameworks considered in this thesis are: Charm++, StarPU and Chameleon. They are examined with a focus on which load balancing and scheduling techniques they use.

4.1 Load Balancing Strategy Categories

There are three major ways to trigger load balancing. Either it is triggered manually by the application, triggered in specified time intervals or triggered continuously, i.e. when one load balancing step is finished, the next one is started immediately.

When the load balancing is triggered, the frameworks need to collect load information and make the decision of how many tasks to migrate between particular ranks. Load information could be simply the number of tasks in the task queue but there are also more sophisticated metrics such as weighted tasks which have an indicator of how much computation time is needed to execute this task. Another way is to collect the execution time of previous tasks with the same kernel function or the same data mapping and to use these times to make an estimation of the needed execution time of future tasks.

The approaches of how to collect load information and how to balance the load can be grouped into three categories: global load balancing, local load balancing and hierarchical load balancing [27].

4.1.1 Global Load Balancing

Global load balancing strategies collect the load information of every rank. This is either done in a single rank or in every rank, i.e. every rank has the load information of every other rank. In the case that the load information is collected in a single rank, this rank makes all migration decisions which are then sent to the respective ranks that need to migrate tasks. In the other case where all ranks have the complete load information, a deterministic load balancing algorithm must be used so that all ranks calculate the same migration decision. Each node knows which tasks it needs

to migrate or receive and will act accordingly. Consequently, there is no need for further communication concerning the load balancing decision.

To collect the load information in a single rank, a linear amount of data needs to be sent which may become a bottleneck when a large number of ranks is used. If an appropriate load balancing algorithm is used, the load balancing will achieve the best performance, i.e. no other load balancing strategy could achieve a smaller makespan.

4.1.2 Local Load Balancing

The counterpart to global load balancing is local load balancing. Here, each rank shares load information only with a defined neighborhood of ranks and only receives load information from this neighborhood. For each rank, task migrations may only be performed within its neighborhood. The load migration decisions may not be the globally best decisions, but less communication is needed and the decision making can be performed asynchronously for each rank. Thus, each rank only has to wait until it received the load information of its neighbors in order to be able to make a migration decision and there is no global synchronization. This is an advantage especially when large numbers of ranks are used.

4.1.3 Hierarchical Load Balancing

For a hierarchical load balancing setup, the ranks are logically divided into equally sized disjoint sets, each of which has one logically defined leader. Each set performs load balancing similarly to a neighborhood in a local load balancing strategy. Additionally, all set leaders are divided into equally sized disjoint sets as well, each of which has a set leader. And within these sets, load balancing is performed as well. This scheme can be extended to a multi-level hierarchy of ranks where ranks migrate tasks within their assigned sets. Hierarchical load balancing has the advantage that little communication is needed for most ranks which is especially important when using large numbers of ranks. It is not guaranteed to achieve a globally best load balance, however.

4.2 Charm++ Framework

Charm++ [11] [19] is a programming framework that provides a message-driven parallel runtime model. To enable usage in a distributed memory environment, so-called chares are used which are specially defined migratable C++ objects.

A chare definition consists of three components: An interface description file, a header file and a main source file. The interface description defines a chare type and its so-called entry methods which are methods that other chares can call. A special compiler is used to cross-compile the interface description file to C++ source code, more specifically to a C++ class definition. In the chare header file, a class

that extends the generated C++ class is defined and its methods are implemented in the main source file. The generated C++ files and the provided C++ files can be compiled using a standard C++ compiler.

Instead of a `main()` method, a main chare is defined whose entry method acts as the `main()` method. The Charm++ runtime will instantiate the main chare on startup by calling its entry method. All further chares will then be instantiated by the main chare. When a chare is created, the Charm++ runtime will assign it to one of the processors that take part in the current execution. Processors in Charm++ are equivalent to ranks in MPI, i.e. they are independent processes that may run on different nodes, but are part of the same application. To the programmer, the placement of the chares on the processors is transparent and completely handled by the runtime. Only the number of processors needs to be specified at execution time. A chare has entry methods specified in its interface description file. Whenever an entry method of a chare is called, it is not executed immediately, but a message is sent to the processor in which the chare currently resides and is pushed onto the processors message queue. The arguments of the method call are also part of the message. A processor will continuously check whether there are messages in its message queue and will execute them, i.e. call the method of the corresponding chare defined by the message with the arguments contained in the message. In Charm++, processors only have one worker thread, that processes its messages. Although messages are not called tasks, they are very similar to them as they execute a method asynchronously and contain some data. Therefore, messages are considered to be equal to tasks in this thesis. The program is terminated when a call to a specified exit method occurs from one of the chares.

Figure 4.1 is an example code of a Charm++ program that implements a Ping Pong scheme using two chares. The `main.ci` file contains the interface definition of the main chare which consists of the main entry point and the `pong()` entry method. This file is cross-compiled to C++ header files, namely `main.decl.h` and `main.def.h`, which contain the code that makes the `Main` class migratable. In the `main.h` file are the forward declarations of the constructors. All entry points declared in the `main.ci` file have a corresponding constructor or method in the header file. The second constructor is required by Charm++ but is not used in this example. The other chare is the `ping` chare which is the counterpart of the `main` chare. It only has the creation entry method and the `ping` entry method. The implementation of the entry points of the chares is given in `main.cpp` and `ping.cpp`. The `Main` method of the main chare is equivalent to the main method of a standard C++ program. In this example, an instance of a `ping` chare is created in line 8. This call will invoke the creation entry point of the `ping` chare in lines 5-10 of `ping.cpp`. This method will print a message and then call the `pong` method of the main chare. To be able to call entry methods on chares without knowing which processor they are currently assigned to, so-called proxies are used, which always point to a chares location. The proxy of the main chare is defined in line 2 of the `main.ci` file and assigned in line 6 of `main.cpp`. Through the `extern` definition in line 1 of `ping.h`, the `ping` chare will also have access to the proxy. The `pong` entry method of the

4 Scheduling and Load Balancing Characteristics

main chare will print another message before terminating the application using the `CkExit()` method. The compiled Charm++ application can be used in a distributed memory environment by executing an MPI runner, since the Charm++ runtime can be configured to perform all its communication over MPI.

main.ci

```
1 mainmodule main {
2   readonly CProxy_Main mainProxy;
3
4   extern module ping;
5
6   mainchare Main {
7     entry Main(CkArgMsg* msg);
8   };
9 };
```

ping.ci

```
1 module ping{
2   chare Ping{
3     entry Ping();
4   };
5 };
```

main.h

```
1 class Main : public CBase_Main {
2   public:
3     Main(CkArgMsg* msg);
4     Main(CkMigrateMessage* msg);
5     entry void pong();
6 };
```

ping.h

```
1 extern CProxy_Main mainProxy;
2
3 class Ping : public CBase_Ping {
4   public:
5     Ping();
6     Ping(CkMigrateMessage* msg);
7 };
```

main.cpp

```
1 #include "main.decl.h"
2 #include "main.h"
3
4 // entry point
5 Main::Main(CkArgMsg* msg) {
6   mainProxy = thisProxy;
7
8   CProxy_Ping pingProxy = CProxy_Ping
9   ::ckNew();
10 }
11 Main::Main(CkMigrateMessage* msg) { }
12
13 void Main::pong() {
14   // print a message
15   CkPrintf("Pong\n");
16
17   // exit application
18   CkExit();
19 }
20 }
21 #include "main.def.h"
```

ping.cpp

```
1 #include "ping.decl.h"
2 #include "ping.h"
3
4 // entry point
5 Ping::Ping() {
6   // print a message
7   CkPrintf("Ping\n");
8   // trigger pong
9   mainProxy.pong();
10 }
11
12 Ping::Ping(CkMigrateMessage* msg) { }
13
14 #include "ping.def.h"
```

Figure 4.1: Basic Charm++ Ping Pong code example

Figure 4.2 gives an overview over an example setup of a Charm++ application at runtime using two processors. Each processor has its own message queue and a set of chares, each of which has some data. The main chare is the initial chare and creates the other chares using seeds, special messages which call the creation entry

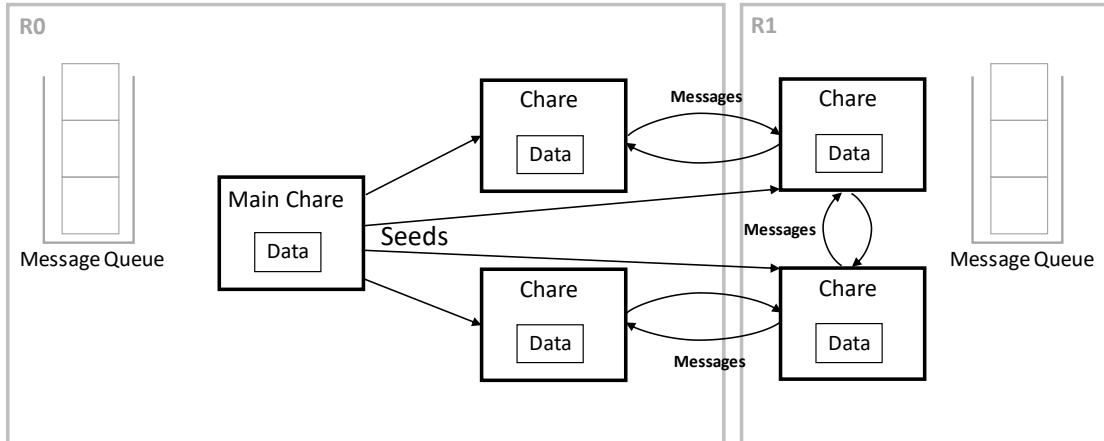


Figure 4.2: Charm++ example setup with two processors

methods. Chares can send messages to other chares which will then be added to the message queue of the processor that the destination chare is currently assigned to.

4.2.1 Scheduling

Each processor has its own message queue onto which messages are pushed and the processors will continuously get a message from the queue and execute its specified entry method. The processor will by default follow the first-in-first-out policy, unless dependencies between messages are specified. It is possible for the program to specify that for every call to an entry point, another entry point must have been called before. When such dependencies have been specified and the message, that is currently the next in the queue, is deferred until all dependencies have been met. The processor will go through the queue until a message has been found for which all dependencies are met. When load balancing is triggered, the processor interrupts its current execution and takes part in the load balancing by sending appropriate load information to other processors, makes decisions on potential chare migrations and performs the actual migrations. In Charm++, load information consists of recorded walltimes of entry methods on the same chare. I.e. walltimes are recorded for the execution of each message and are used as an estimation on how long subsequent executions of the same entry method on the same chare take. Due to the fact, that a processor only has one worker thread, the message queue are less likely to suffer from lock contention. Only if a lot of messages are sent to one processor, the lock may be contended. On the other hand, one processor needs to be launched for every core that is used which comes with overhead and with this setup, more inter-process communication is needed than in a setup with multiple worker threads per processor.

4.2.2 Load Balancing

The Charm++ framework provides extensive load balancing capabilities that can be configured to a programs need [19]. Load balancing in Charm++ programs means the permanent migration of chares from one processor to another. From then on, all messages sent to the chare are put into its processors message queue, until the chare is potentially remigrated to a different processor.

The load balancing can be triggered using one of three methods. It can either be triggered directly by the program using the `CkStartLB()` method which will start the load balancing immediately. Another option is to use the `atSync()` method. If all chares of a processor have called this method, load balancing is performed. The third option is to use periodical load balancing which will trigger the load balancing in specified intervals. When the load balancing is triggered, processors interrupt the execution of messages for the duration of the load balancing process. Therefore, the frequency of the load balancing directly influences performance.

There are also different load balancing strategies implemented in Charm++, some of which are introduced in the following.

GreedyLB and RefineLB

GreedyLB and RefineLB are global load balancing strategies. GreedyLB will order the messages by their estimated computational load and will assign the largest task to the least used processor, i.e. the processor which currently has the least estimated computation load. To do this, every rank needs to know the estimated computation load of every other rank which is why this strategy is a global load balancing strategy. As described above, the estimation of the computational load is based on timed previous executions of the same entry methods. RefineLB takes a different approach. Instead of focusing on the messages, it focuses on processors and migrates messages away from the most overloaded processors. Both GreedyLB and RefineLB are heuristics and are therefore not guaranteed to achieve the globally best load balance, even though they have access to the global load information. Also, since they are global load balancing strategies, the communication amount scales with the number of processors, which may become a bottleneck when using large numbers of processors. However, both load balancing strategies are simple to implement and have predictable behavior.

NeighborLB

NeighborLB is a local load balancing strategy. Each processors has a defined set of neighbors, which are only a subset of the processors. The processors then will try to average out the computational load in their neighborhood by migrating chares. This approach may also result in suboptimal load balances since the processors do not have access to the load information of processors outside of their neighborhood. If the majority of the computational load would be concentrated in just one processor, it would take several load balancing iterations for the chares to be propagated to

the logically most distant processor. This load balancing strategy is also simple but does not require the amount of messages sent per load balancing step that a global load balancing strategy requires.

HybridLB

Charm++ also implements a hierarchical load balancing strategy with HybridLB [27]. It is intended for setups with very large numbers of processors where both global and local load balancing strategies fail to provide effective load balancing. The processors are logically ordered into a hierarchy, i.e. a tree, and processors perform load balancing with their siblings as well as their direct parents and direct children. On the lower levels, GreedyLB is used and Refine LB at the root level. Although this approach is more complex than a global or local load balancing strategy, it has the advantage that it scales better and should therefore be used when large numbers of processors are needed.

4.3 StarPU

StarPU [2] [24] is a task-based programming framework that was originally developed for the use in heterogeneous systems, i.e. systems with CPUs and attached GPUs. The StarPU runtime can handle the automatic task migration between host and device including the data transfer. Additionally, support for distributed memory environments was added and StarPU programs can be compiled and executed using standard MPI implementations.

The central data structures in StarPU are so-called *codelets* which describe a task by defining a kernel function and the data mapping. To be able to use StarPU in a distributed memory environment and to allow task migration, each task needs to be registered on each rank. Registering a task involves defining a codelet, specifying the data mapping and then calling the task creation function. The data only needs to be present on one rank, but all tasks need to be registered on all ranks. In a StarPU application, each rank launches a set of worker threads that will then execute the tasks. The load balancing is performed a-priori at the time of the invocation of the `taskwait` function and is not rebalanced at runtime.

Figure 4.3 is an example of a StarPU application that scales a vector. The function `scaleKernel` is the function kernel as is given the mapped parameters through the `descr[]` variable. Once the values have been read, the actual computation is performed. `scaling_c1` is the codelet that describes a task by defining the kernel function, the number of parameters and their respective mapping types. The lower code section is part of the `main()` function after the initialization, which is omitted for the sake of readability. Each task needs three mapping specifications, each of which needs to be registered with the runtime and additionally registered as a piece of data that can be migrated over MPI. The two constants that specify the size of each task and the scaling factor are registered in lines 6-11 and they can be

used for every task. Only the part of the array that is handled by one task needs to be registered per task, which is done in lines 14-15. Then, the actual task is created with a specified MPI communicator, the codelet and the mapped data with the specification of their mapping types. Finally, in line 26, control is given to the runtime which will execute all tasks and return control back to the application afterwards.

```

1 void scaleKernel(void *descr[], void *_args) {
2     float* factor = (float*) STARPU_VARIABLE_GET_PTR(descr[0]),
3     float* array = (float*) STARPU_VARIABLE_GET_PTR(descr[1]),
4     int* array_size = (int*) STARPU_VARIABLE_GET_PTR(descr[2]),
5     for(int i = 0; i < *array_size; i++) {
6         array[i] *= *factor;
7     }
8 }
9
10 struct starpu_codelet scaling_cl =
11 {
12     .cpu_funcs = {scaleKernel},
13     .nbuffers = 3,
14     .modes = {STARPU_R, STARPU_RW, STARPU_R},
15 };

1 float vector[TASK_COUNT * TASK_SIZE];
2 float factor;
3 uint task_size;
4
5 starpu_data_handle_t vector_handles[TASK_COUNT];
6 starpu_data_handle_t size_handle;
7 starpu_variable_data_register(&size_handle, 0, &(task_size), sizeof(uint));
8 starpu_mpi_data_register(size_handle, 0, 0);
9 starpu_data_handle_t factor_handle;
10 starpu_variable_data_register(&factor_handle, 0, &(factor), sizeof(float));
11 starpu_mpi_data_register(factor_handle, 1, 0);
12
13 for(int i = 0; i < TASK_COUNT; i++) {
14     starpu_variable_data_register(vector_handles+i, 0, vector+(i*TASK_SIZE), sizeof(
15         float)*TASK_SIZE);
16     starpu_mpi_data_register(vector_handles[i], i+2, 0);
17
18     starpu_mpi_task_insert(
19         MPI_COMM_WORLD,
20         &scaling_cl,
21         STARPU_R, factor_handle,
22         STARPU_RW, vector_handles[i],
23         STARPU_R, size_handle,
24         0);
25 }
26 starpu_task_wait_for_all();

```

Figure 4.3: StarPU Tasking example

4.3.1 Scheduling

StarPU provides several scheduling policies, which determine how tasks are distributed among the worker threads of a ranks and which order they are executed

in. The eager scheduler has one task queue per rank and the worker threads take tasks from it concurrently. This approach has the disadvantage that with large numbers of worker threads, the queue may become a bottleneck because access to it must be mutually exclusive. However, it provides an automatic load balancing capability within the rank. The work stealing policy on the other hand assigns separate queues to each of the worker threads and distributes tasks to these queues. Whenever a worker thread is idle, it will steal a task from the worker thread with the most tasks. This approach avoids the potential contention of a central queue but introduces overhead by having to manage multiple queues. StarPU also offers additional scheduling policies such as the locality work stealing policy where tasks are not stolen from the queue with the highest load but from a neighboring queue. It is also possible to specify priorities of tasks which will be considered when using the priority scheduling policy.

As a more advanced feature, StarPU supports scheduling based on performance modeling. The execution of tasks is monitored and based on previously executed tasks, the scheduler makes estimations on how long new tasks will take to execute which helps the scheduler make decisions that allow for a lower makespan. This feature is especially effective when the task size varies and the computation time needed by a task can be estimated from the size of the data mapped to the task. It introduces the overhead of monitoring however and does not provide much benefit when tasks are roughly equally sized.

4.3.2 Load Balancing

When used in a distributed memory environment, StarPU uses several ranks, each of which acts like a StarPU instance in a shared memory system with the additional feature that task execution can be load balanced between ranks. All tasks need to be registered on all ranks, even if the task data resides on another rank. All ranks then process the so-called task tree, which is the set of all tasks with potential dependencies. The StarPU runtime then decides which tasks are processed by which rank and each rank only executes its subset of the task while skipping others. This load balancing strategy is therefore a global load balancing strategy. The decision where to execute which tasks is mainly based on where the data needed by the task resides and each task is assigned to a rank where minimal data needs to be fetched from other processors. Although StarPU provides automatic load balancing capabilities, this approach is not as reactive as other approaches such as Charm++ and Chameleon.

4.4 Chameleon Framework

Chameleon [23][13] is a framework developed by LMU Munich, Technical University of Munich and RWTH Aachen University. It leverages MPI and OpenMP to provide a task-based programming model similar to OpenMP tasks, but in a dis-

tributed memory environment. Using a standard MPI setup, a program consists of ranks and in Chameleon, each rank can launch worker threads which will participate in executing the tasks. There are two ways of creating tasks with Chameleon. One way is to use the OpenMP target offloading directives which were introduced with OpenMP version 4.0 [17]. Using the `target` construct, a special task can be defined which can be offloaded and executed on devices such as GPUs or FPGAs. Additionally, the `target` construct already provides clauses to define mapped data. In Chameleon, these tasks can also be offloaded to another rank. To do this, the compiler needs to support task offloading and must allow custom plugins to provide the interface to the Chameleon runtime. Currently, there only exists a plugin for the LLVM compiler suite. The other way to create tasks is to use the provided API, which offers the same functionality, but also works with other compilers which do not support plugins for offloading targets. Created tasks are always put into the task queue of the rank on which they are created and can also be created in parallel. After the creation of the tasks, a distributed taskwait is called by the application. This function is called by all ranks and will trigger the execution of queued tasks by the worker threads. Further, on a separate thread dedicated to communication, the Chameleon runtime continuously performs load balancing using the three steps: introspection, consolidation and migration. Introspection means the process of locally collecting load information, e.g. the local task count. During consolidation, the load information is exchanged between ranks and the migration decision is made. Afterwards, tasks are migrated if needed. Once a remote task has been executed, its data needs to be migrated back to its original ranks.

Due to the asynchronous execution and the possible migration of tasks, any MPI communication should be performed outside of tasks. Performing communication within Chameleon tasks could lead to incorrect and unpredictable results since the order of execution of the tasks and thus the communication-calls cannot be controlled. Also the source and target of point-to-point communication could be incorrect due to a possible migration of tasks. Collective communication would also pose a problem with migrated tasks since they require every rank to participate.

Figure 4.4 is an example of a Chameleon program that scales a vector by creating the needed tasks and waits until all tasks are executed. This example is the Chameleon equivalent of the StarPU example program given in the previous section. In the upper code section, the kernel function is defined, which takes the needed arguments and performs the scaling operation. The most important part of the `main()` method is displayed in the lower code section. The vector is logically divided into equally sized chunks and for each task, the mapping of its chunks is specified in lines 4-12. Afterwards, the task is registered with Chameleon and once the loop is finished, the `chameleon_distributed_taskwait()` method is called, which will trigger the task execution and introspection in the background.

Figure 4.5 depicts an example of a chameleon program during runtime. There are two ranks with 5 worker threads and one communication thread each. The task queue of the left rank contains 6 tasks while the other contains 3 tasks. Under the assumption that all tasks have a similar workload, one or several tasks will

```

1 void scaleKernel(float* factor,
2                 float* array,
3                 int* array_size) {
4     for(int i = 0; i < *array_size; i++) {
5         array[i] *= *factor;
6     }
7 }

1 float vector[TASK_COUNT * TASK_SIZE];
2 float factor;
3 for(int i = 0; i < TASK_COUNT; i++) {
4     chameleon_map_data_entry_t* args = new chameleon_map_data_entry_t[3];
5     args[0] = chameleon_map_data_entry_create(&factor, sizeof(float),
6       CHAM_OMP_TGT_MAPTYPE_TO);
7     args[1] = chameleon_map_data_entry_create(vector + (i*TASK_SIZE),
8       sizeof(float)* TASK_SIZE,
9       CHAM_OMP_TGT_MAPTYPE_TO | CHAM_OMP_TGT_MAPTYPE_FROM);
10    args[2] = chameleon_map_data_entry_create(&array_size, sizeof(int),
11      CHAM_OMP_TGT_MAPTYPE_TO);
12
13    cham_migratable_task_t *cur_task = chameleon_create_task(
14      (void *)&scaleKernel,
15      3, // number of args
16      args);
17    int32_t res = chameleon_add_task(cur_task);
18 }
19
20 chameleon_distributed_taskwait(0);

```

Figure 4.4: Chameleon Tasking example

be migrated from rank 0 to rank 1 in this setup. This however depends on the configuration, more specifically on the given imbalance threshold above which the program will start to migrate tasks.

4.4.1 Scheduling

When the taskwait method is called by the application, all threads of each rank will repeatedly get tasks from the task queue and execute it until the task queue is empty. There are two kinds of tasks, however: local tasks and remote tasks. Local tasks have been created on the same rank as they are executed and remote tasks are tasks that have been migrated to this rank from another rank. Chameleon gives remote tasks priority, i.e. they are executed first and only if no remote tasks are present on the current rank, local tasks are executed. Because remote tasks need to migrate data back to their original rank, it is important that they are executed as early as possible so that the data remigration and the execution of other (potentially local) tasks can be overlapped. In this way, an additional slowdown due to communication can be avoided because the communication overhead is hidden.

By performing task data remigration as early as possible, a minimal waiting time once all tasks have been executed can be assured.

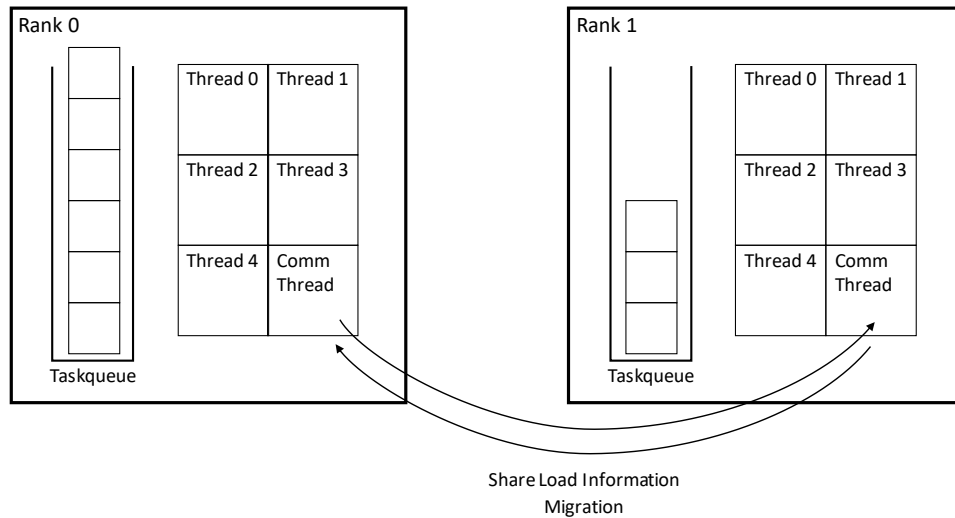


Figure 4.5: Example setup of a Chameleon program during runtime

4.4.2 Load Balancing

By default, Chameleon uses a global load balancing strategy. All ranks send their load information to all other ranks, which make migration decisions using a deterministic algorithm and then migrate tasks if necessary. The load balancing mechanism in Chameleon is predictive and reactive in the sense that load imbalances are detected and compensated as early as possible. The load balancing is performed continuously on a dedicated communication thread per rank. This thread can therefore not be used to execute tasks. On the other hand, this load balancing mechanism ensures responsiveness and reactivity. The default load balancing algorithm sorts all ranks by their estimated load, for example by their task count. Then, pairs of ranks are created by e.g. taking the two outermost ranks of the sorted list, i.e. the ones with the highest and lowest load, so that each pair consist of two counterparts. Load balancing is then performed within this pair with the rank with the higher workload migrating as tasks to the other rank. Migration is only performed if the number of tasks per rank does not fall below a configurable threshold. This ensures that late migrations are avoided, which may cause idle times. Additionally, migration is only performed when the imbalance between the two ranks is above another configurable threshold. Otherwise, the overhead of the task migration would outweigh the benefit of the load balancing. This pair-based approach ensures a low communication effort since each rank only communicates with one other rank. As with all global load balancing strategies however, communication to distribute load information may become a limiting factor when using large numbers of ranks, however. Chameleon also offers the capability for the programmer to customize its own load balancing strategy through the tools interface. Here, the programmer can

implement callbacks which will be called by the Chameleon runtime and make the task migration decisions.

4.5 Performance Model

A theoretical model to be able to generally predict performance of task-based parallel programs is desirable.

For this proposed producer-consumer model, some assumptions and simplifications need to be made. Assume that only two ranks are used and that the tasks are homogeneous, i.e. they all need the same compute time to execute and all take the same time to migrate. Also, assume a maximum imbalance between the ranks, where one rank has all tasks and the other none.

Let t be the compute time needed to execute one task. Let l be the time needed to migrate one task between ranks, i.e. the time needed for communication. Let c be the number of cores per rank. The rate of task consumption (i.e. execution) is given by $v = \frac{c}{t}$. The more cores are used, the faster the consumption and the more time a task needs, the lower the consumption. On the empty rank, the production of task (i.e. the migration from the other rank) is given by $p = \frac{1}{l}$.

To keep the empty rank busy, $p \geq v$ must hold which means that at least as many tasks are produced as consumed. When substituting p and v with their respective terms, $\frac{1}{l} \geq \frac{c}{t}$ must hold. Since c is given by the used architecture, the ratio between c and l is most important. The time needed to migrate tasks must be as low as possible and the time to execute a task as high as possible to be favorable to Chameleons load balancing approach. Also, the higher the core count, the more important this ratio becomes.

To apply this model to real applications, the times to migrate a task and to execute a task can be measured empirically. Together with the given number of cores, one can check, whether Chameleon is able to compensate load imbalances of the given application.

4.6 Framework Comparison

Three task-based parallel programming frameworks have been introduced, namely Charm++, StarPU and Chameleon. Each framework has its own distinctive set of features, advantages and drawbacks. A comparison of the frameworks is desirable to be able to decide which framework is the most suitable for a given type of application. All three frameworks offer a task-based programming capability to applications which can specify tasks by a function and some data and the frameworks will asynchronously execute these tasks. Although Charm++ does not call them tasks, its messages are very similar to tasks and can therefore be considered to be tasks.

Charm++ applications have a different compilation process than Chameleon or StarPU applications. The interface specification files need to be cross-compiled

using a special compiler and the resulting code must then be compiled using a standard compiler. The programmer must therefore write the application in C++. Chameleon and StarPU applications on the other hand need to be compiled and executed using the MPI wrapper compilers and runners. For programmers used to MPI+OpenMP development, these frameworks have less steeper learning curve compared to Charm++, which also has a special paradigm by executing everything using chares, not the traditional imperative and object-oriented paradigms. This also makes the integration of existing codes into a Charm++ application more cumbersome.

Charm++ only uses one thread per processor which avoid lock contention on the queue but also comes with overhead as there are more queues and also involves more inter-process communication. StarPU and Chameleon on the other hand have multiple threads per rank which guarantees automatic load balancing within each rank, but is also vulnerable to lock contention on the queue. For this case, StarPU also offers a multi-queue scheduling policy with work-stealing within each rank.

All three frameworks offer automatic load balancing capabilities, but in different variations. Charm++ offers a wide range of load balancing strategies including several global, local and a hierarchical strategy. This is especially useful when working with various numbers of nodes. Chameleon and StarPU each only provide a fixed global load balancing strategy, while Chameleon allows for custom strategies which can be provided by the programmer.

Charm++ also differs significantly from the other two frameworks with respect to task migration. In both Chameleon and StarPU, data is local to a rank and is migrated if needed, but always migrated back to the original rank once the task is completed. Charm++ chares on the other hand are migrated permanently. Both approaches have situations for which they are effective. In iterative programs such as stencil codes, the permanent migration is able to compensate higher load balances as the load is balanced during the first few iterations of the load balancing and is well balanced from then on. The remigration of tasks in Chameleon and StarPU is the reason why they can only handle lower imbalances.

Generally, Chameleon is the most fine-granular framework. It is reactive due to its continuous load balancing on a separate thread and can quickly compensate small load imbalances. It is not suited for bulk load balancing due to the fact that it remigrates data of migrated tasks. Chameleon can work well in setups with small, unpredictable load balances, such as Adaptive Mesh Refining applications or to compensate for hardware-induced load imbalances arising on a short time scale. It may also be combined with a bulk load balancing strategy executed by the application itself and then compensate for the more fine-granular unpredictable load imbalances. Charm++ on the other hand does not provide continuous load balancing but rather load balancing at points specified by the application or in intervals. Both StarPU and Charm++ block the task execution during the load balancing process while Chameleon performs the load balancing in the background to have overlapped communication and computation. Charm++ is not as reactive and cannot handle hardware-induced load imbalances as effectively, but also does not

need a separate communication thread. Additionally Charm++ can handle larger load imbalances, especially in iterative applications due to its permanent migration of chares. StarPU is well-suited for applications where the load imbalance is known at the time when the `taskwait` method is called. It will then distribute the workload among all ranks but is unable to react to load imbalances that occur during the execution of the tasks. It is therefore also not as reactive as Chameleon.

For all three applications, granularity of tasks play a vital role. There is a tradeoff between the ability to perform fine-granular load balancing with many small tasks and the overhead that comes with many tasks. It is important to find a good balance between the two.

Charm++ migrates complete chares, not just the data needed for every task. This is another disadvantage when trying to compensate for small imbalances because potential more data than needed is sent. Both StarPU and Chameleon define the data mapping on a per-task basis and thus, task migration only consists of sending the necessary data of that task.

5 Benchmarks

To evaluate the frameworks introduced in the previous chapter, two benchmarks are used. One is a synthetic benchmark which was designed to be well-suited for load balancing and can be configured with different degrees of load imbalances. It calculates approximations of the mathematical constant π . The other benchmark simulates wave propagation in water. It is a real-world application and the scenarios can also be configured to contain load imbalances.

5.1 Pi Benchmark

The mathematical constant π can be characterized by the following formula:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

Since π is an irrational number, computers can only calculate approximations of it. This benchmark computes such an approximation by partitioning the area under the function into slices and calculating the area of the slices.

The value of π is the size of the area below the function $\frac{4}{1+x^2}$ between 0 and 1. To approximate the size of the area, the interval $[0, 1]$ is divided into slices. For each slice, the value of the function is evaluated and multiplied by the width of the slice resulting in the area of one slice. Then the sum of the area of all slices is an approximation of π . The error of the approximation becomes smaller, the more slices are used. Figure 5.1 depicts the integral and its approximation using slices. The orange curve is the function $\frac{4}{1+x^2}$ in the range $[0, 1]$. Here, the integral is approximated using 20 slices represented by the green rectangles. The sum of the area of all slices is the approximation of π and the area between the slices and the function represents the error of the approximation, i.e. the difference between the approximation and the actual value of π .

The calculations for each slice are independent of each other allowing for a straightforward parallelization. To ensure the precision of the calculation, variables of the type `double` are used in all versions of this benchmark. A baseline version of the benchmark using MPI+OpenMP was implemented as well as a version for each of the considered frameworks.

5.1.1 MPI+OpenMP Version

To be able to compare the implementations of the benchmark with a traditional, more static approach without tasking, a baseline version using MPI and OpenMP

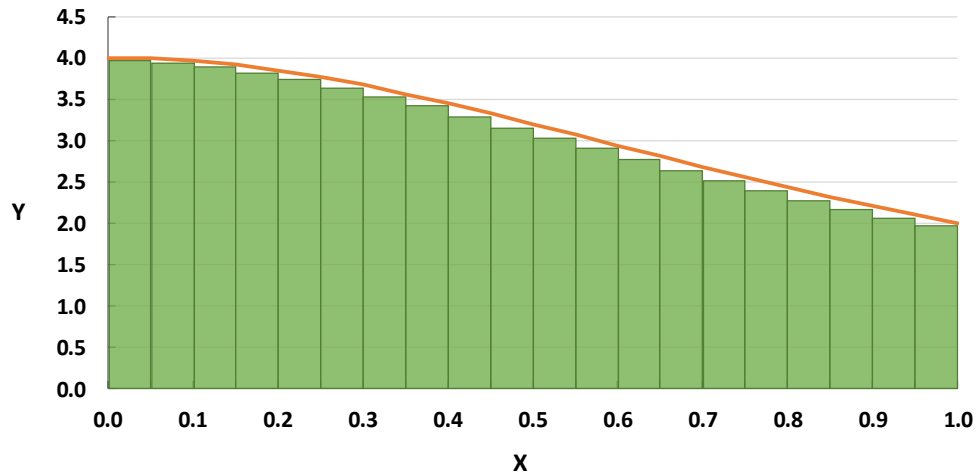


Figure 5.1: Approximation of π using an integral. The orange curve is the function $\frac{4}{1+x^2}$ and the total size of the green areas represent the approximation of π

was implemented. The slices that need to be evaluated are distributed statically among the ranks and each rank distributes its slices statically among its threads. Each thread calculates the size of the area for each of its slices and aggregates them into a single variable. To calculate the overall result, each rank performs an OpenMP sum reduction followed by a global MPI sum reduction. This version does not offer any load balancing capability but on the other hand introduces minimal overhead. To simulate load imbalances, the program can be configured to assign more slices to some ranks than others. Since no dynamic load balancing is performed, the rank with the highest workload will dictate the overall execution time and the other ranks will be idle some of the time.

5.1.2 Charm++ Version

The calculation for one slice only consists of evaluating the function at the respective points and multiplying the result with the width of the slice. In total, 4 floating point operations per slice have to be performed. Thus, using one chare per slice would introduce too much overhead and therefore sets of adjacent slices are grouped into so-called chunks, each of which is processed using one chare. In the Charm++ version, there are two kinds of chares: the main chare and chunk chares. The main chare creates the chunk chares and is also used for the final reduction and synchronization.

Three parameters are needed to define a chunk: the lower boundary, the slice width and the number of slices of this chunk. Since these values are known at the time

of the creation of the chunk chare, they are passed to the chare using the creation entry method and no other entry methods need to be defined. When a chunk chare has finished its calculations, it will send a message to the main chare containing the sum of the area of its slices and the chare ID. The main chare will sum up the results of the calculations and terminate the application when all chunk chares have sent their results.

The distribution of chares to processor elements is not influenced by the application but automatically determined by the Charm++ runtime. Since a chunk can be defined using just three 64-bit values, the communication effort is constant while the computational load per chunk can be adjusted with different slice counts. Experiments have shown that performance is best with chunk sizes of at least 100,000 slices which ensures a favorable ratio between computational workload and communication effort if a chare needs to be migrated. The time spent on communication would be insignificant compared to the time needed to complete the computational workload of a chunk.

To simulate load imbalances, it can be configured to use chunks of different sizes so that some chunks have a higher computational load than others.

5.1.3 Chameleon Version

For the Chameleon version, the approach of grouping slices into chunks is used as well. The chunks are evenly distributed among the ranks and on a rank level, one task is created for each chunk that needs to be processed. A task in this version is therefore similar to a chunk chare in the Charm++ version. It is defined by the three values lower boundary, slice width and number of slices which are calculated before the creation of the tasks and are mapped to the task using the Chameleon data mapping functionality. Since they are not needed afterwards, they are only mapped to the task as input parameters, i.e. should the task be migrated, they are only sent to the remote rank, but not back. In addition to the input variables, another variable is mapped that is needed to store the result of the calculations of that task. As opposed to the other variables, this variable does not need to be sent to the remote rank, but only back to the local rank once the task has been concluded.

With this approach, each rank has a set of local tasks which can be processed without any communication and if there are load imbalances, tasks can be migrated to remote ranks. Similar to the Charm++ version, chunks should have a size of at least 100,000 slices resulting in a favorable ration between computational load and communication effort for each task.

After all tasks have been executed each rank has an array of partial results which are summed up locally before a global MPI reduction is performed to get the global result.

Load imbalances are simulated in a similar fashion as in the Charm++ version by generating chunks of different sizes that will result in different computational workloads.

5.1.4 StarPU Version

Since StarPU offers a similar tasking approach as the Chameleon framework, the StarPU version is implemented in a similar fashion as the Chameleon version. The approach of creating tasks, each of which computes one chunk is kept as well as the final reduction of the results of all tasks. The main difference is the fact that all ranks register all tasks and the StarPU runtime makes the decision which tasks are executed on which rank. For each task, the variables are mapped in the same way as in the Chameleon version. The three values that define a chunk are mapped to the tasks and the result is mapped from the task.

5.2 Shallow Water Equations Benchmark

The Shallow Water Equations Benchmark [7] is a benchmark developed at the Technical University of Munich that simulates wave movements in a shallow water environment. It implements a Finite Volume model in a two-dimensional domain.

Figure 5.2 displays an example of an SWE simulation state. Here, the tsunami in the Tohoku region of Japan in 2011 was simulated.

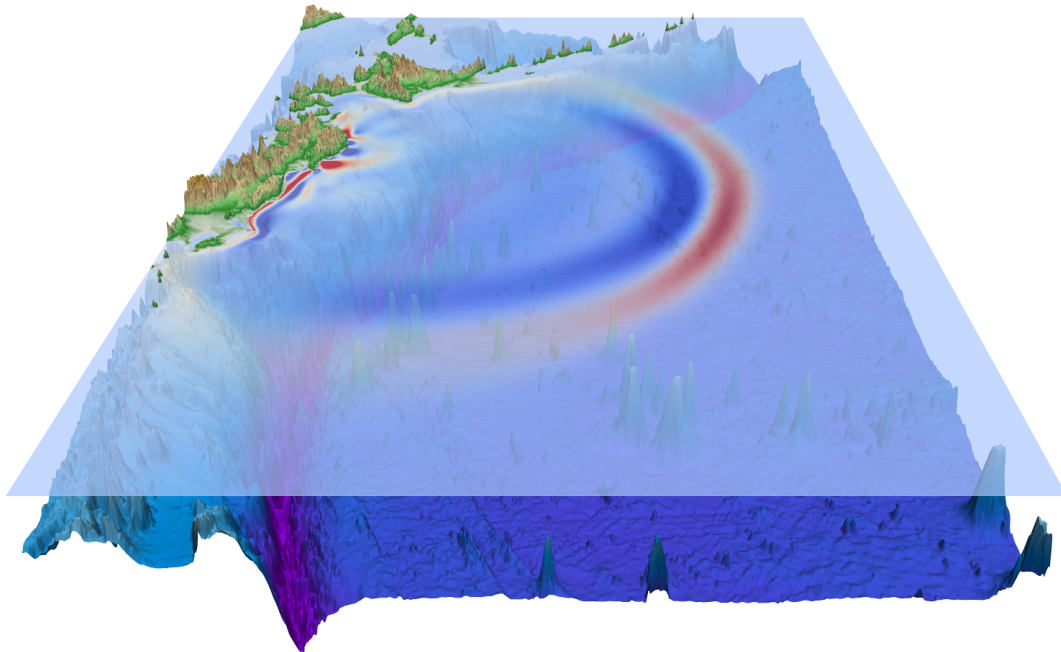


Figure 5.2: SWE simulation example using real-world data from the tsunami in the Tohoku region of Japan in 2011 [7]. The ground height and water height is augmented for the sake of visibility.

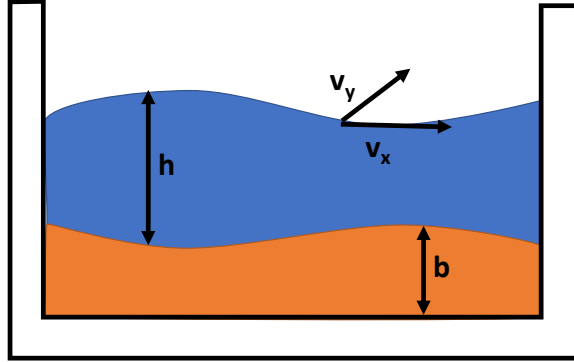


Figure 5.3: Cross section of the simulation scenario. The sea-floor is on the bottom, the water on top and the simulation boundaries on the left and right.

5.2.1 Mathematical Model

Let h be the height of the water, b the height of the sea-bottom. Furthermore, let v_x and v_y denote the horizontal and vertical momentum, respectively. Let g be the (constant) gravitational acceleration on earth. Then, the changes over time at a given point in the water can be modeled using the following system of partial differential equations [7]:

$$\begin{aligned} \frac{\partial h}{\partial t} + \frac{\partial(v_x h)}{\partial x} + \frac{\partial(v_y h)}{\partial y} &= 0 \\ \frac{\partial h v_x}{\partial t} + \frac{\partial(h v_x v_x)}{\partial x} + \frac{\partial(h v_y v_x)}{\partial y} + \frac{1}{2} g \frac{\partial(h^2)}{\partial x} &= -g h \frac{\partial b}{\partial x} \\ \frac{\partial h v_y}{\partial t} + \frac{\partial(h v_x v_y)}{\partial x} + \frac{\partial(h v_y v_y)}{\partial y} + \frac{1}{2} g \frac{\partial(h^2)}{\partial y} &= -g h \frac{\partial b}{\partial y} \end{aligned}$$

Figure 5.3 gives an overview of the variables in the simulation scenario setup. On the bottom is the sea-floor with varying height b , on top is the water with varying height h .

5.2.2 Discretization

Real-world seas are continuous, but for the simulation a discrete model is needed so that a computer is able to store the simulation data. To achieve this, the model needs to be discretized. A two-dimensional grid is used to approximate the continuous domain where the four values h , b , v_x and v_y are stored for every cell. Additionally,

the time is discretized into time steps and the simulation calculates the changes from one time step to the next in order to approximate continuous time progression. With the applied discretization, the simulation consists of the iteration of three computation steps: First, the horizontal numerical fluxes of each cell is computed by considering the h , b , v_x and v_y values of the cell itself and the four surrounding cells. Flux is a metric for the transfer of mass (in this case water) from one cell to another. This computation involves solving differential equations. Differential equations are to difficult to solve exactly and thus, numerical algorithms are used to compute approximate solutions which are called numerical fluxes. More specifically, an Augmented Riemann solver is used to solve differential equations in this benchmark. After this step, the global maximum of the horizontal numerical fluxes is computed, which is a value needed to determine the maximal time span of the current iteration step. In the second step, the vertical numerical fluxes are computed for each cell in a similar fashion as the horizontal numerical fluxes. The third step consists of applying the computed numerical fluxes to the unknown variables (h , b , v_x , v_y) resulting in the final state of the iteration. The numerical fluxes are applied with a factor depending on the time span covered by the current iteration step. In every simulation iteration, only the four directly adjacent cells are considered to calculate the updates of the current cell. In order for this concept to correctly approximate real-world behavior, the horizontal and vertical velocity can only be so large, that the water from one cell moves no more than one cell per time step. In other words, the time span of an iteration can only be so large that the fastest water movement does not move more than one cell. Only the maximum horizontal numerical flux is considered because the simulation assumes that the maximum vertical fluxes are not much greater than the maximum horizontal fluxes.

The input data for a simulation only consists of the two-dimensional arrays for the bathymetry and the water height. The horizontal and vertical fluxes are assumed to be 0 at the beginning. The scenarios can either be generated artificially or the benchmark can load real-world data from input files.

During the computation, the solver can detect dry cells, i.e. cells where the water height of the cell itself and all of its adjacent cells is 0. Then the solving is skipped because a dry cell does not have a horizontal or vertical flux nor a water height. This results in a possible load imbalance since certain areas of the grid may need significantly less computation time than others if they have a lot of dry cells. And there typically are no single dry cells, but rather dry spots on the grid.

5.2.3 Partitioning

In order to be able to execute the simulation in a distributed memory environment, the 2D grid needs to be partitioned among the ranks. The partitioning is also necessary to use a task-based programming approach. Here, the grid is divided into even blocks of a given size. To solve the differential equations in each cell, the current data from all surrounding cells is needed. In a shared memory environment, it is

simple to access the surrounding cells. In a distributed environment however, this access requires communication between ranks. So-called ghost-layers are used to minimize communication and simplify the implementation. Figure 5.4 depicts two blocks with ghost-layers and also the communication scheme that is used. The cells within the black squares are the actual cells of the blocks and the surrounding layers of green squares are the ghost-layers. The gray corner cells are allocated, but not used. A ghost-layer is an artificial layer of cells that are placed on the four sides of a block. During the simulation, they are used to store the data of the outer cells of the corresponding adjacent block. Thus, a block of size $nx * ny$ stores $(nx + 2) * (ny + 2)$ cells, but only performs the calculations for $nx * ny$ cells, omitting the outer layers on each side. When the ghost-layer exchange is executed, each block sends the outermost layer of its actual cells to the adjacent blocks in each direction who will write them to their respective ghost-layers. When the calculations are executed for the actual cells, the correct values of the adjacent cells can be used.

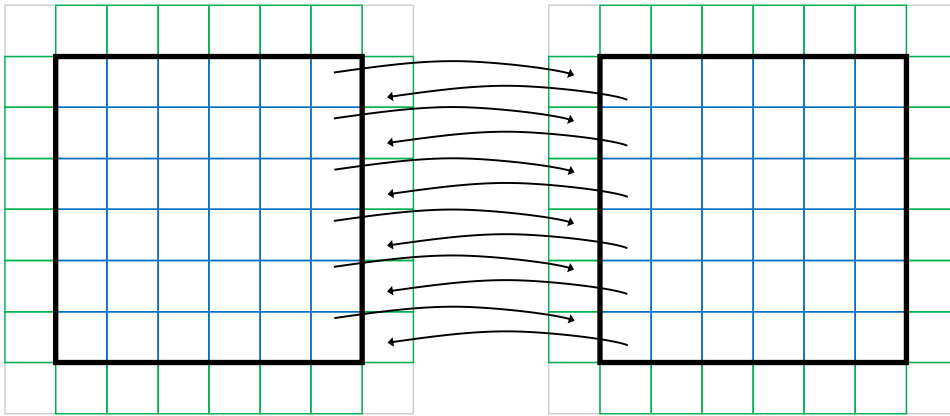


Figure 5.4: Overview of the block-based decomposition of the grid.

The steps in the simulation with the discretization and the partitioning are presented in figure 5.5. Initially, the input data needs to be loaded. Then, each block needs to fetch the content of its ghost layers from its adjacent blocks. After that, the necessary calculations for the horizontal numerical fluxes are performed, which is also called a horizontal sweep. The global maximum of the horizontal numerical flux is determined and the maximum wave speed which determines the time span of the current iteration is determined. Afterwards the vertical sweep is performed which is the calculation of the vertical numerical fluxes. The horizontal and vertical numerical fluxes are applied to the unknown variables (h, b, v_x, v_y) representing the current state of the simulation. This state can be written to an output file, if configured. This procedure is repeated until the desired simulation length is reached.

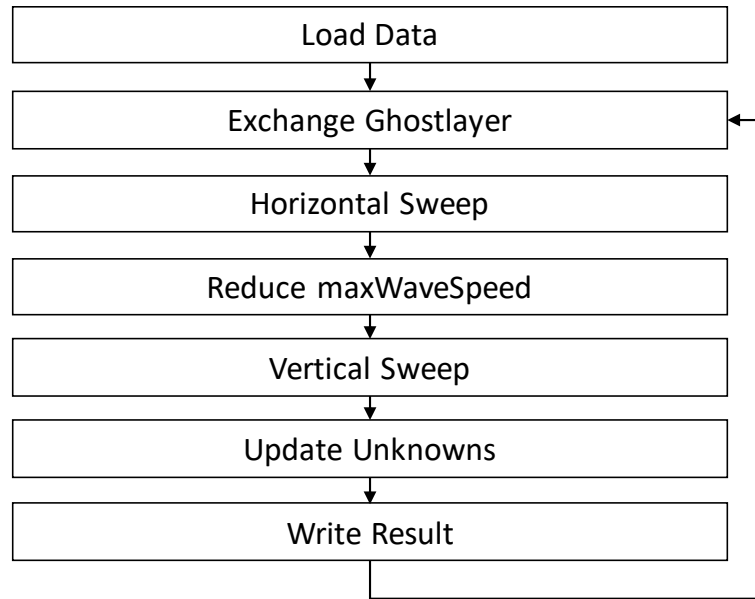


Figure 5.5: Overview of SWE simulation steps with applied discretization and block-based partitioning

5.2.4 MPI+OpenMP Version

As a baseline version, an MPI+OpenMP version [7] was used which assigns one block to each rank and then performs the calculations for each block in parallel using OpenMP `parallel` for constructs with static scheduling. The reduction to compute the time span of each iteration step, i.e. the maximum reduction of the horizontal fluxes, is achieved using an OpenMP maximum reduction within each block followed by a global MPI maximum reduction. To implement the ghost-layer exchanges before every iteration, asynchronous Point-to-Point communication is used with message tags that indicate the direction from which the ghost layer was sent. This version does not offer any load balancing capability but on the other hand introduces minimal overhead.

5.2.5 Charm++ Version

The Charm++ version was already implemented in the context of a bachelor thesis [16] and was adapted to fit the needs of this thesis.

There are two kinds of chares: the main chare and block chares. The main chare is used to create the block chares which contain the simulation data and perform the simulation calculations. Additionally, it is used for synchronization and to perform reductions.

Each block chare contains one block of the grid and defines entry methods for each

of the steps of the simulation. The initial data for a block is loaded in the creation entry method and the ghost-layer exchange is implemented using entry methods which accept specifically defined kinds of messages that contain one ghost-layer. Only when a chare has received all expected ghost-layer messages, it will perform the horizontal sweep and send its maximum wave speed to the main chare. The main chare will wait until it has received all values and then calculates the global maximum. Subsequently, it sends the maximum to all block chares which in turn will perform the vertical sweep, apply the updates and write the results before initiating the next iteration. Once the desired simulation length is reached, the main chare terminates the application.

In the context of this thesis, the Charm++ version of the benchmark was adapted to perform over-decomposition, i.e. to divide the grid into significantly more blocks than available processors used. This is important for the automatic load balancing performed by the Charm++ runtime. An analysis of the impact of block granularity on performance is conducted in the next chapter.

5.2.6 Chameleon Version

The Chameleon version applies the same over-decomposition of the grid as the Charm++ version. Each rank is assigned a set of blocks that are adjacent in the global grid. This allows for most ghost-layer exchanges to be performed in shared memory because both blocks are assigned to the same rank. For each the horizontal sweep, the vertical sweep and the updating of the variables, a task is created for each block by the rank that the block is assigned to. These three steps contain the computational workload and are therefore parallelized using the Chameleon framework. The ghost-layer exchange and the reduction of the maximal time span are not compute-intensive and involve MPI communication, which is why they cannot be executed using Chameleon tasks. Instead, these steps are implemented in a similar way as in the MPI+OpenMP version. The main difference is the fact, that in this version, most ghost-layer exchanges can be performed within shared memory and the MPI+OpenMP version only has one block per rank and thus no exchanges within a rank.

Figure 5.6 gives an overview over the distribution of blocks to ranks in a setup with 9 ranks. Each of the small squares represents one block. The green edges represent ghost-layer exchanges that can be performed within shared memory and the red edges represent ghost-layer exchanges across nodes.

5.2.7 StarPU Version

The StarPU version is similar to the Chameleon version because the two frameworks offer a similar tasking approach. The concepts of over-decomposition, assigning blocks to ranks and creating tasks for the computational expensive steps is kept. However, all tasks are created on all ranks, because the StarPU framework demands this in order to be able to migrate tasks. The data for each task is only present

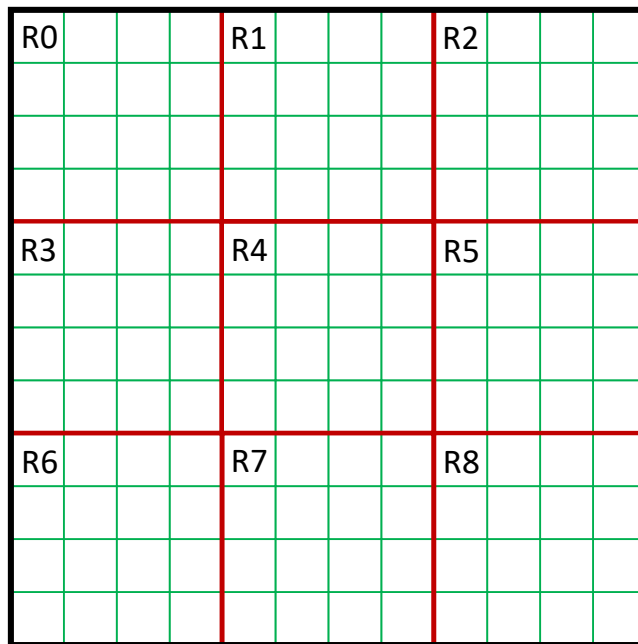


Figure 5.6: Overview over distribution of blocks to ranks

on the rank that the respective block is assigned to. The scheduler will then try to assign the tasks to a rank where the needed data is present. If there is a load imbalance and tasks need to be migrated, the StarPU runtime will migrate the data and execute the respective task on a different rank.

6 Evaluation

The intent of this thesis is to analyze task-based parallel programming models with respect to their scheduling and load balancing characteristics. After a theoretical analysis and comparison in chapter 4, the results are further underlined with empirical measurements using the two benchmarks introduced in the previous chapter.

6.1 Environment

Performance measurements were performed on the CLAIR 2016 system [8]. The compute nodes are dual-socket systems consisting of two Intel Xeon E5-2650v4 CPUs and 128 GBytes of main memory. The processors have a base frequency of 2.2 GHz and a maximal boost frequency of 2.9 GHz. Each node has a total of 24 cores with disabled hyperthreading and the nodes are connected with an Intel Omni-Path fabric. The operating system that is used is CentOS 7.6.

To compile libraries and benchmarks, the Intel Compiler 19.0.1 was used as well as the Intel MPI Library 2018 Update 4. The Chameleon library does not have versions and for the results in this thesis, the current development version as of September 2019 was used [13]. StarPU 1.2.8 was used for the StarPU experiments and Charm++ 6.9.0 for the Charm++ experiments.

6.2 Methodology

Task-based parallel programming frameworks offer an alternative paradigm than the traditional approach with static load partitioning using MPI and OpenMP. First of all, the performance and scalability of the frameworks need to be evaluated and compared to a traditional approach. This is done by performing standard strong scaling experiments for each benchmark using well-balanced scenarios. Due to runtime errors of the StarPU version of the SWE benchmark, StarPU is only used for the Pi benchmark experiments.

Both benchmarks can be configured to create or simulate software-induced imbalances of various degrees. The imbalances experiments consist of executing the benchmarks with the same node count and the same task granularity, but different degrees of load imbalances.

To simulate changes in the clock frequency of processors deterministically, a configurable interference program was designed which performs additional computations on the same processor as the benchmarks thus decreasing the compute resources.

6 Evaluation

Figure 6.1 shows an example execution of the interference program. Here, two ranks, each with two threads are used and the interference program will only interfere on one rank. Of every interval of 100 ms, the program will perform calculations during a configurable fraction of the interval resulting in load imbalances of different degrees. For experiments in this thesis, the interference program is only used in a setup with two nodes where the interference is executed on one of the nodes to create a load imbalance. Alternatively, to simulate OS jitter, the interference program can be configured to only use one thread on one node. For the experiments, the interference program is started in the background. The length of the time fractions of the interference is varied with the same node count and the same task granularity.

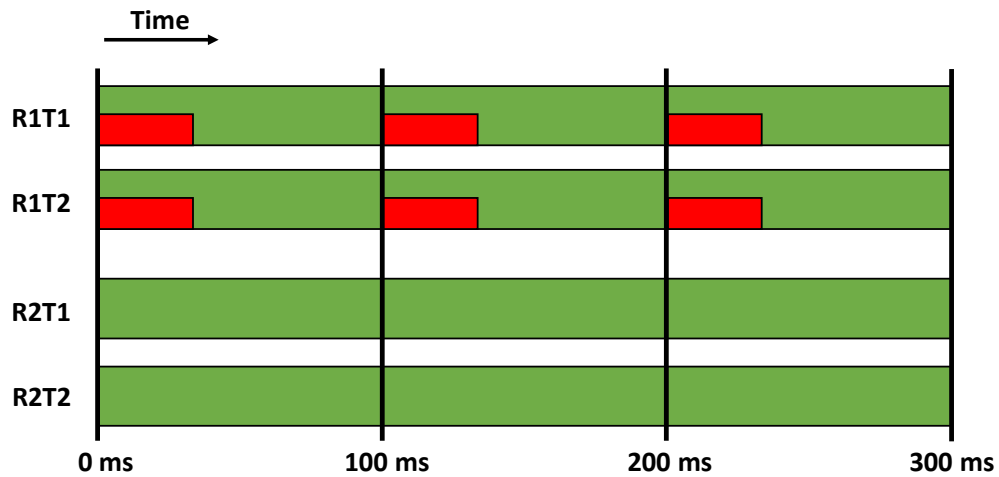


Figure 6.1: Example execution of interference program with 2 ranks and 2 threads each. The green blocks depict computations by the application while the red blocks depict interference. The interference program works with time windows of 100 ms and enables the interference in a configurable fraction of each time window.

To determine the task granularity with the best performance for each benchmark, experiments were conducted with task granularities. A load imbalance was configured for these experiments for both benchmarks in order to also measure how well the frameworks can compensate this load imbalance with varying task granularities. All experiments were executed 5 times and the median of the execution times is given as well as the minimum and maximum. MPI+OpenMP and StarPU experiments use 2 ranks per node and 12 threads per rank. Charm++ experiments use 24 processors per node and Chameleon uses 2 ranks, each with 11 worker threads and one communication thread. For Charm++, the load balancing strategy GreedyLB was used with a trigger interval of 1 second. MPI processes were pinned using `I_MPI_PIN=1` and threads were pinned using `OMP_PROC_BIND=close`.

6.3 Pi Benchmark Experiments

6.3.1 Strong Scaling

The results of the strong scaling experiments as well as the speedups compared to the baseline version are given in figure 6.2. The tasks are all equally sized to create a well-balanced setup. Since two threads per node are reserved for communication purposes in Chameleon, the Chameleon version has a lower performance than the MPI+OpenMP version, as expected. The Charm++ version closely matches the performance of the MPI+OpenMP version, which indicates a low overhead by the task-based approach of Charm++.

In general, all task-based frameworks scale well in this setup, especially Charm++. MPI+OpenMP does outperform the frameworks however, which is expected since the task-based paradigm introduces some overhead compared to a static worksharing setup.

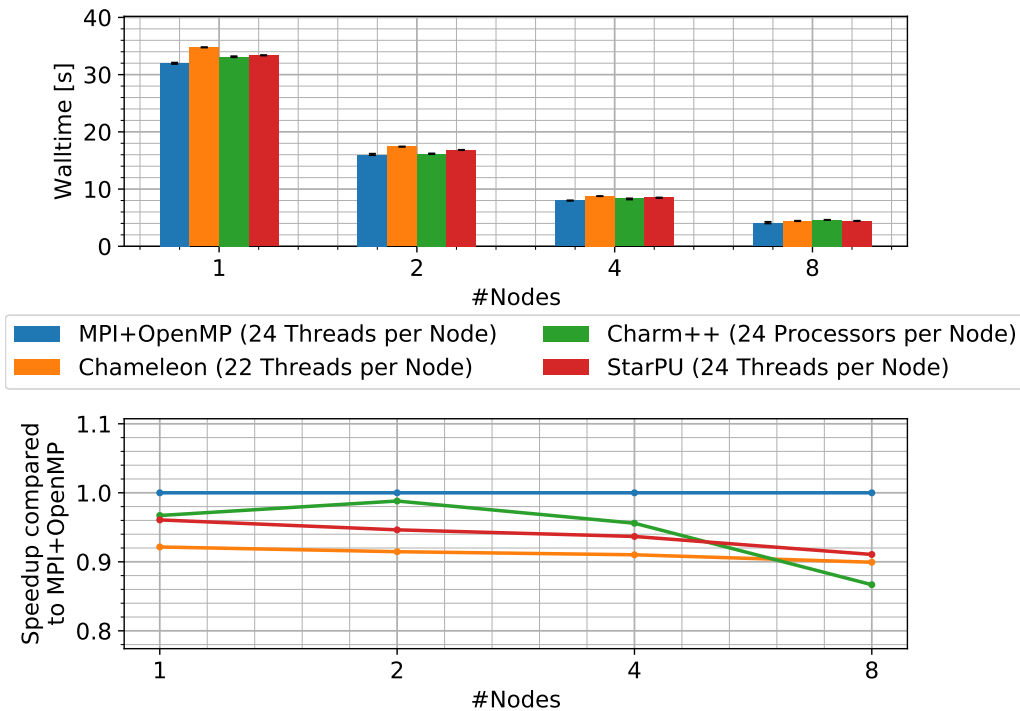


Figure 6.2: Strong scaling behavior of frameworks in Pi benchmark using different numbers of nodes. $4 \cdot 10^{11}$ slices were used with a taskcount of 8196.

6.3.2 Work-Induced Imbalances

Figure 6.3 presents the results of experiments with varying software-induced imbalances as well as the speedups compared to the MPI+OpenMP version. Two nodes were used, each of which is assigned the same number of chunks. But the chunks

6 Evaluation

have different amounts of slices. The ratio between the maximal number of slices of a chunk and the average amount of slices is varied.

Because the MPI+OpenMP version does not perform any load balancing, its performance decreases with increasing load imbalance. One node has the larger chunks and therefore takes longer to process them while the other node runs idle and waits once its load has been processed. Chameleon and Charm++ show consistent performance regardless of the imbalance. Because the Pi benchmark has tasks that are favorable for load balancing due to their high computational load and low migration effort, this behavior is expected. Without a load imbalance (with a ratio of 1.0), the MPI+OpenMP version is the fastest and the Charm++ and Chameleon have a lower performance due to overhead.

Charm++ achieves a speedup of up to 1.45 compared to MPI+OpenMP with a load ratio of 1.5, but only achieves 0.95 times the performance without a load imbalance. The speedups of the Chameleon version are lower with 1.3 in the experiment with a load ratio of 1.5 and a speedup of 0.9 in the experiment without load imbalances. StarPU shows lower performance than MPI+OpenMP and is unable to compensate the load imbalances.

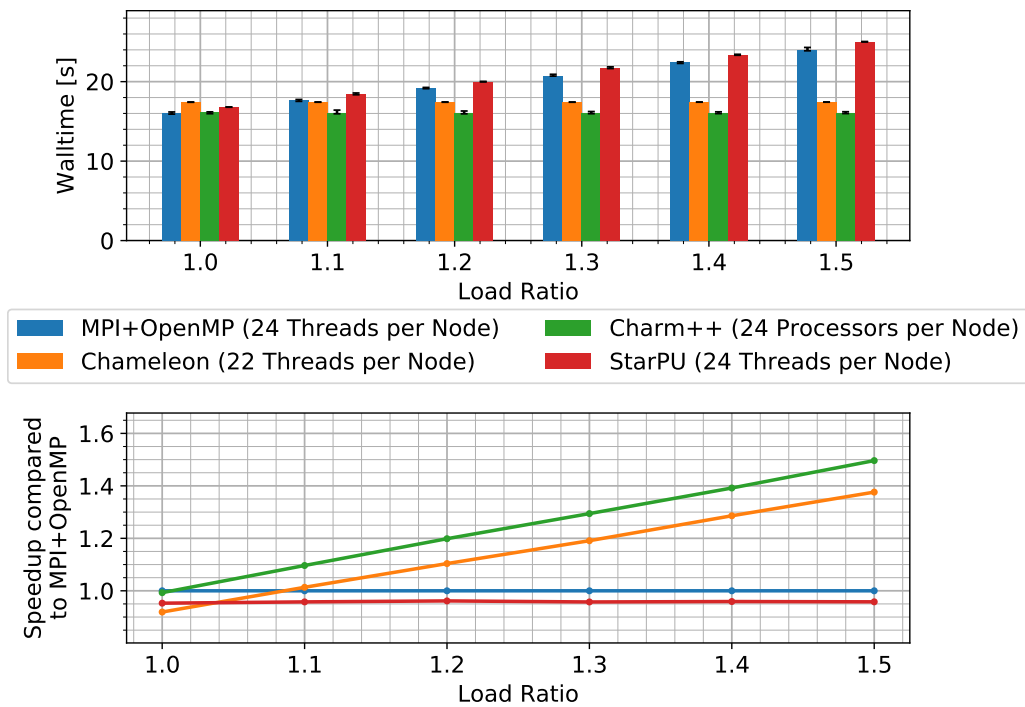


Figure 6.3: Reaction to imbalances of frameworks in Pi benchmark. $4 \cdot 10^{11}$ slices were used with a taskcount of 8196. Load ratio denotes the ratio between the maximal number of slices of a task to the average number of slices per task. All versions were executed using 2 nodes.

6.3.3 Interference

Figure 6.4 displays the results of the interference experiments for the Pi benchmark to simulate hardware-induced imbalances. Again, without interference, the MPI+OpenMP version has the best performance. With increasing interference fraction however, Chameleon shows better performance and starting at an interference fraction of 1.25. With increasing interferences the execution time for Charm++ and StarPU increases, similar to MPI+OpenMP. Generally, it is expected, that performance decreases because the interference program uses computing resources, which are then unavailable to the application. The interference program runs on only one of the two nodes and reactive load balancing, like Chameleon offers, uses continuous dynamic detection of emerging imbalances at runtime to make migration decisions. StarPU cannot compensate inter-node load imbalances once the taskwait function has been called and can therefore not compensate hardware-induced load imbalances. Charm++ cannot make estimations on the computational load of messages because these estimations are based on previous timed executions of the same entry method on the same chare. Since the entry methods of each chare are only executed once during the complete execution of the application, this estimation is not feasible.

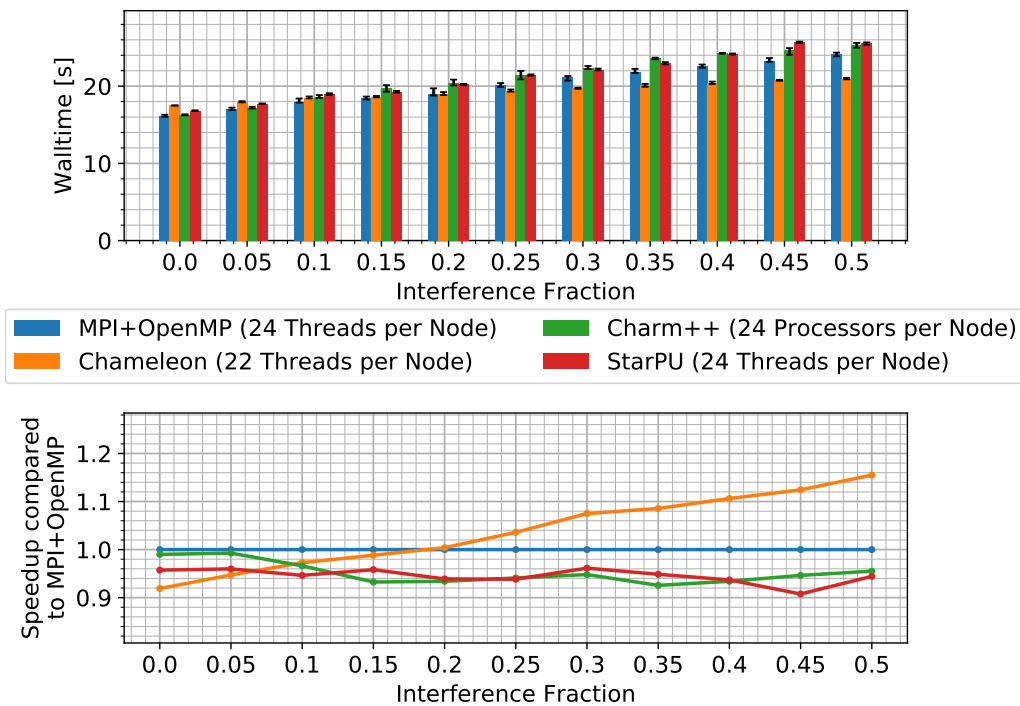


Figure 6.4: Reaction to interference of Pi benchmark. $4 \cdot 10^{11}$ slices were used with a taskcount of 8196. All versions were executed using 2 nodes. The interference application was run on one of the two nodes with varying time fractions in which interference was enabled.

6.3.4 Interference (Single Core)

The interference experiments were also executed with interference on just one thread on one of the nodes and the results are shown in figure 6.5. They are very similar to the results of the previous interference experiments. For MPI+OpenMP and StarPU, this is expected as they have a static load partitioning. Charm++ shows again, that it is unable to compensate fine-granular load imbalances. StarPU and Chameleon on the other hand are able to achieve a speedup of over 1.4 compared to MPI+OpenMP with an interference fraction of 0.5. Because the interference only affects one core in this setup, StarPU can use the other working threads on that rank to execute the tasks. The same is true for Chameleon which may additionally migrate tasks to the other rank.

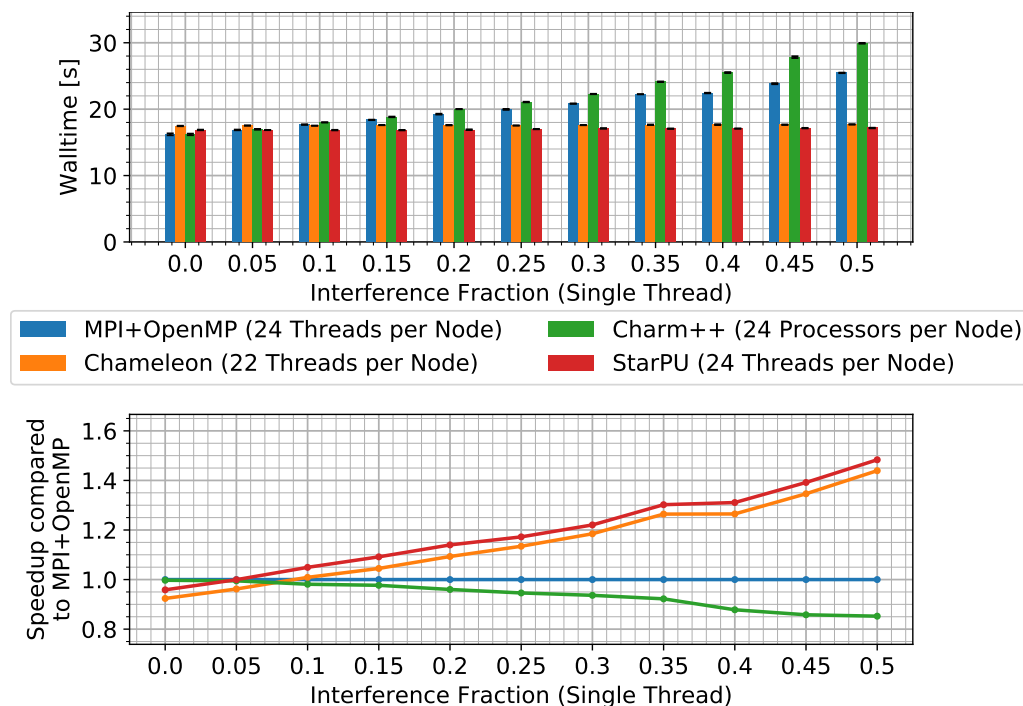


Figure 6.5: Reaction to single-core interference of Pi benchmark. $4 \cdot 10^{11}$ slices were used with a taskcount of 8196. All versions were executed using 2 nodes. The interference application was run on one core on one of the two nodes with varying time fractions in which interference was enabled.

6.3.5 Granularity

The granularity of tasks is of great importance for the performance of task-based programming frameworks. Experiments with varying task granularities are presented in figure 6.6. As the MPI+OpenMP version does not have a task-based approach, it is excluded from the granularity experiments. As expected, large task sizes have

a slight performance disadvantage since they result in schedules where some workers are idle at the end. Smaller task sizes avoid this problem but come with more overhead. This is especially true for the Charm++ version, for which the performance is significantly lower with less slices per task. Therefore, for all frameworks, performance is best in between the two extremes.

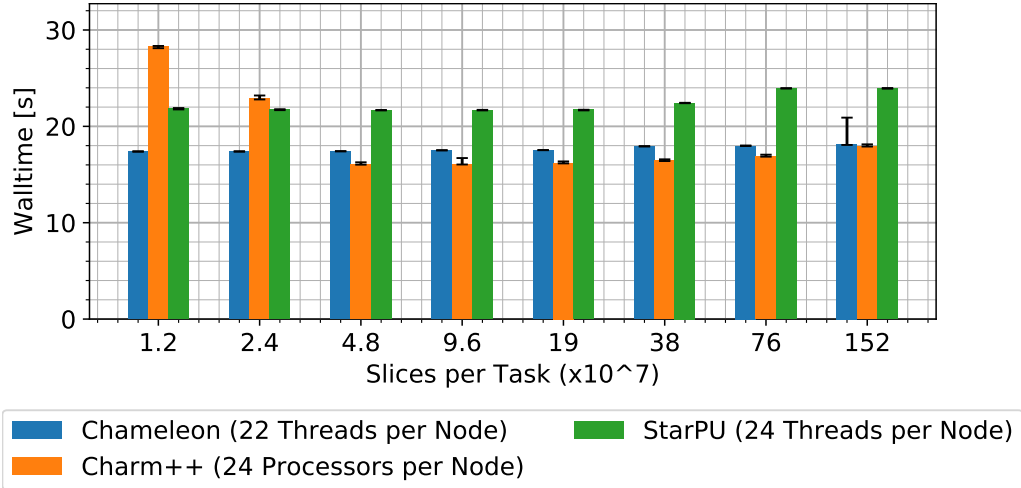


Figure 6.6: Reaction to task granularity of frameworks in Pi benchmark. 4×10^{11} slices were used with varying numbers of slices per task. A load ratio of 1.3 was configured and all versions were executed using 2 nodes.

6.4 SWE Benchmark Experiments

6.4.1 Strong Scaling

The strong scaling behavior of the frameworks using the SWE benchmark is presented in figure 6.7. All versions have similar walltimes. Since the used scenario is well-balanced, the MPI+OpenMP version is the fastest with all node counts. It does not have the overhead that comes with task creation and queuing. The performance of the Chameleon version is slightly below the MPI+OpenMP performance which can be explained by the fact that Chameleon reserves one thread per rank for communication. The Charm++ version shows similar behavior to the Chameleon version and also has a slightly lower performance than the MPI+OpenMP version.

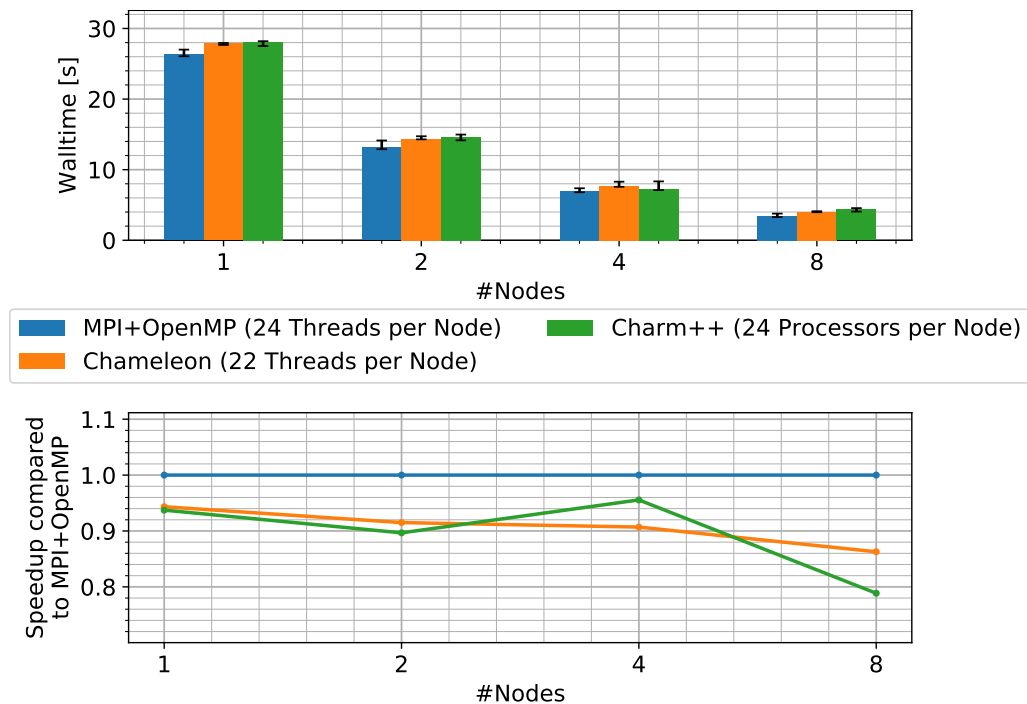


Figure 6.7: Scaling behavior of frameworks in SWE benchmark using different numbers of nodes. A well-balanced synthetic scenario was used with a grid size of 4096x4096, which was split into blocks of 128x128 cells and 200 simulation steps were executed.

The speedups of all frameworks are below 1.0, as they are slower than the MPI+OpenMP version. Generally, the frameworks scale well, but have a slowly decreasing speedup compared to the MPI+OpenMP version with increasing numbers of nodes.

6.4.2 Work-Induced Imbalances

The results of the work-induced imbalance experiments for the SWE benchmark are presented in figure 6.8. The node count is fixed to two nodes and the so-called

dry fraction is varied, which describes the fraction of dry cells in the scenario. In these experiments, the dry cells were all located on the left side of the grid. As a consequence, some parts of the simulation need less computation resulting in a load imbalance.

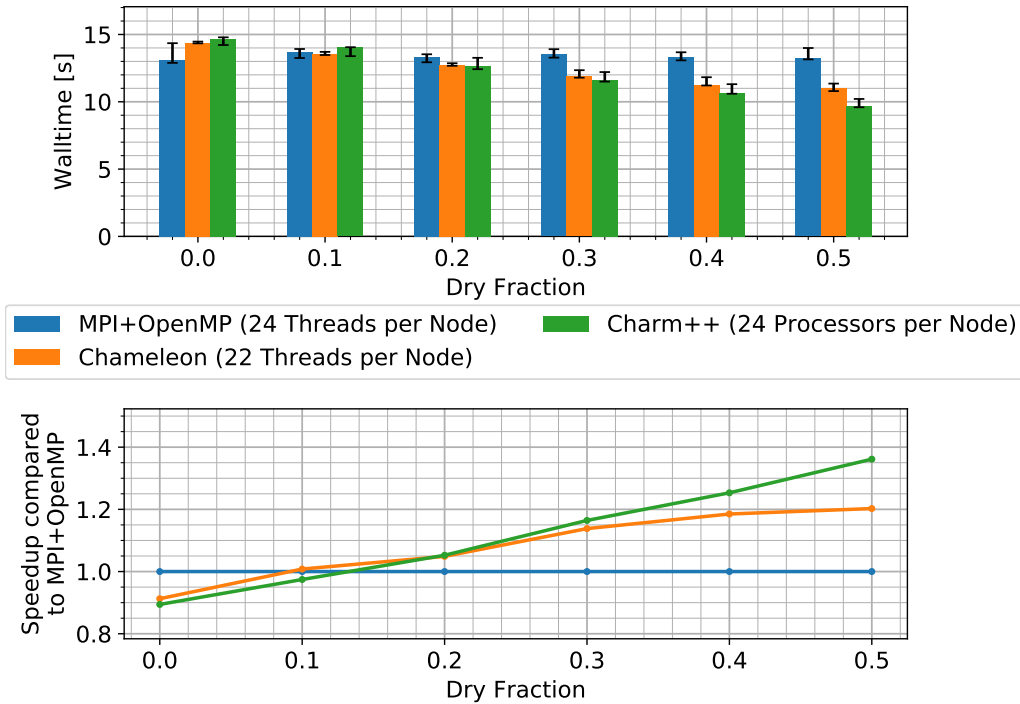


Figure 6.8: Reaction to load imbalances of frameworks in SWE benchmark. A well-balanced synthetic scenario was used with a grid size of 4096x4096, which was split into blocks of 128x128 cells and 200 simulation steps were executed. The dry fraction denotes the fraction of the cells that are dry (i.e. the water height is zero). The solver recognizes dry cells resulting in significantly fewer calculations. All versions were executed using 2 nodes.

The MPI+OpenMP version has no load balancing capabilities and is therefore expected to show a constant performance independent of the dry fraction because there is always at least one rank that has no dry cells in its block. This expectation is mostly confirmed since the performance in experiments with higher dry fractions is similar to the performance in the experiment without dry cells even though the overall computational load is lower.

The Chameleon and Charm++ versions also behave as expected. Without a dry fraction, their performance is below that of the MPI+OpenMP version, which is the same setup as in figure 6.7. Performance increases steadily with increasing dry fraction. At a dry fraction of 0.1, all three frameworks have nearly the same performance and with greater dry fractions, Chameleon and Charm++ have better performance than the MPI+OpenMP version with an increasing performance gap.

With dry fractions of 0.3 and higher, the measured Charm++ and Chameleon performances diverge while they previously have been nearly the same. This observation can be explained by the fact that Chameleon uses temporary task migration while Charm++ migrates chares permanently. Cells that are dry will also potentially be dry during the next iteration with the exception of cells that are placed close to the border between dry and wet cells. Generally, Charm++ is well-suited for iterative codes. The Chameleon version will detect the emerging imbalance as soon as some of tasks containing dry cells have been executed. Then it will start to migrate tasks from ranks with higher load to ranks with lower load and will transfer the results back to the original rank. When the imbalance is too high, the communication bandwidth limits the migration of tasks and the overall performance will stagnate as seen in the results. The Charm++ version on the other hand will detect the imbalances after the first execution just like the Chameleon version but will then migrate chares, i.e. blocks, permanently to ranks with a lower load. Since the load distribution does not change significantly over time, there is no need for further balancing in the next iterations. For high imbalances, it may take a few iterations to achieve a good balance but it will be reached eventually as the load distribution does not change. This property is the reason why the performance of the Charm++ version increases steadily even for higher load imbalances while the Chameleon performance stagnates.

One can see that both frameworks are 10% slower than MPI+OpenMP if there are no dry cells. Chameleon achieves a speedup of up to 1.2 while Charm++ has a speedup of up to 1.35 with a dry fraction of 0.5.

6.4.3 Interference

Experiments with interferences were also conducted for the SWE benchmark. Their results are presented in figure 6.9. In contrast to the results of the Pi benchmark, Charm++ outperforms Chameleon with higher imbalances in both cases. This can again be explained by the fact that Charm++ uses permanent migration and Chameleon temporary migration. The interference is always active on one node and thus, permanent migration has an advantage, especially when the ratio of computational load and migration effort per task is not favorable like with the SWE benchmark.

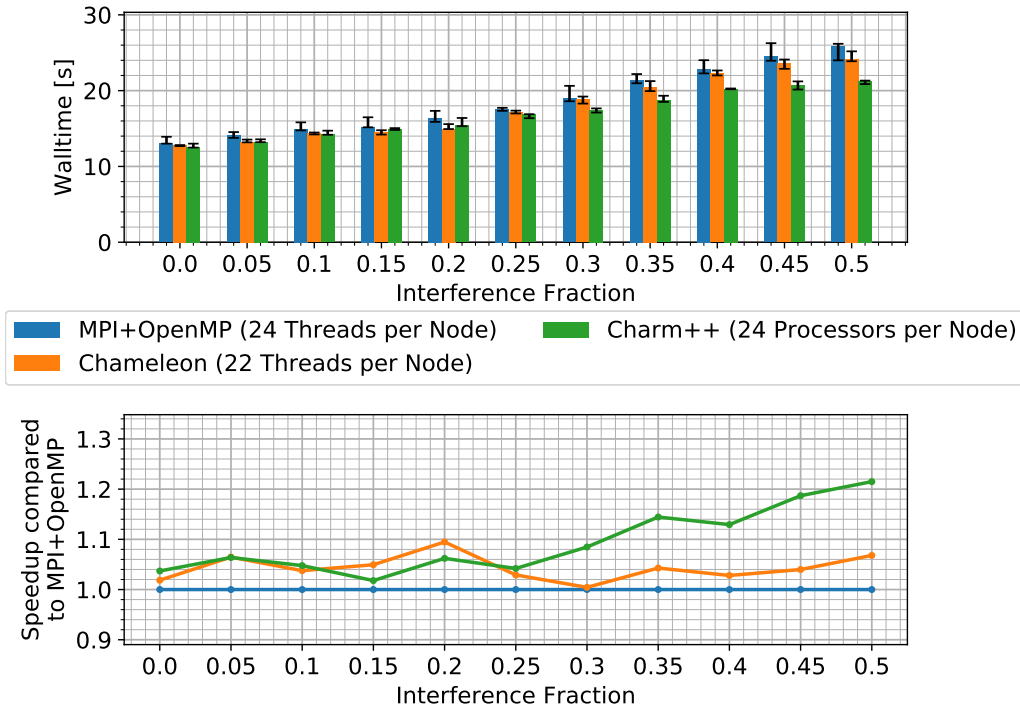


Figure 6.9: Reaction to interference of SWE benchmark. A well-balanced scenario was used with a grid size of 4096x4096, which was split into blocks of 128x128 cells and 200 simulation steps were executed. All versions were executed using 2 nodes. The interference application was run on one of the two nodes with varying time fractions in which interference was enabled.

6.4.4 Interference (Single Core)

The single-core interference results are given in figure 6.10. In contrast to the full interference experiments, Chameleon outperforms Charm++ here. Charm++ has lower performance than the baseline version which clearly indicates that it is unable to detect and compensate the load imbalance in this case. This may be because the interference program is enabled during the load balancing phase and because all ranks are blocked until it is finished, the interference negatively affects all other ranks as well. The speedup peaks at the interference fraction of 0.4 are caused by lower performance of the baseline version while the frameworks are more consistent.

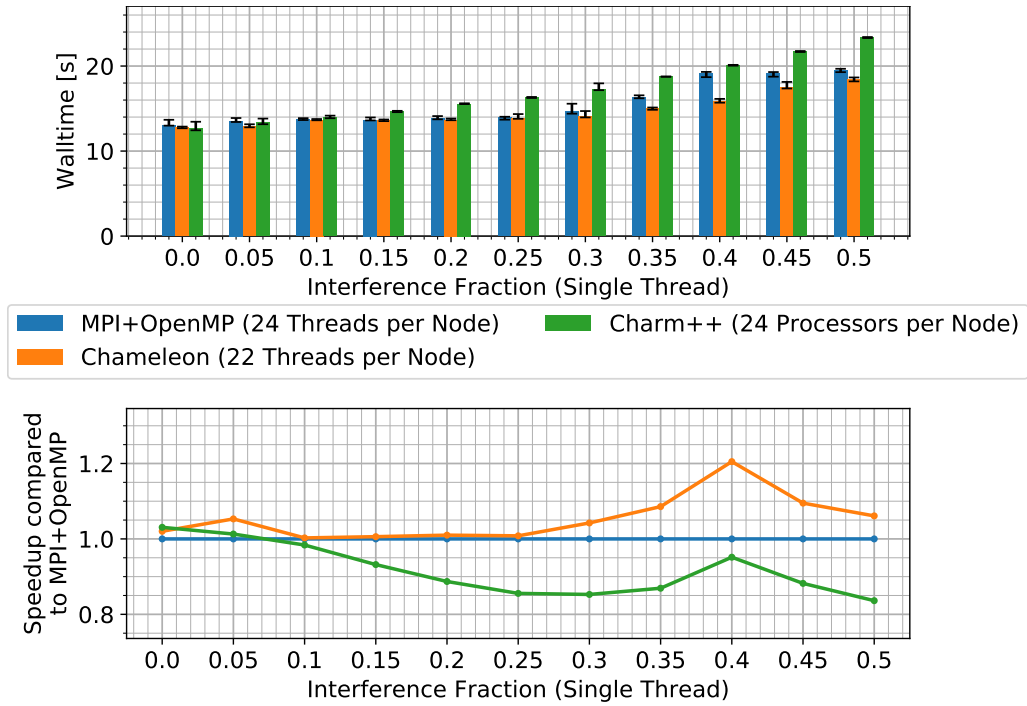


Figure 6.10: Reaction to single-core interference of SWE benchmark. A well-balanced scenario was used with a grid size of 4096x4096, which was split into blocks of 128x128 cells and 200 simulation steps were executed. All versions were executed using 2 nodes. The interference application was run on one of the cores on one of the two nodes with varying time fractions in which interference was enabled.

6.4.5 Granularity

Results of granularity experiments for the SWE benchmark are presented in figure 6.11. The MPI+OpenMP version is omitted again because it does not use a task-based approach. The results clearly show, that a blocksize of 128x128 has the best performance for both Chameleon and Charm++. Larger blocks cause idle time because the task scheduling cannot be fine-granular enough and smaller block sizes have too much overhead.

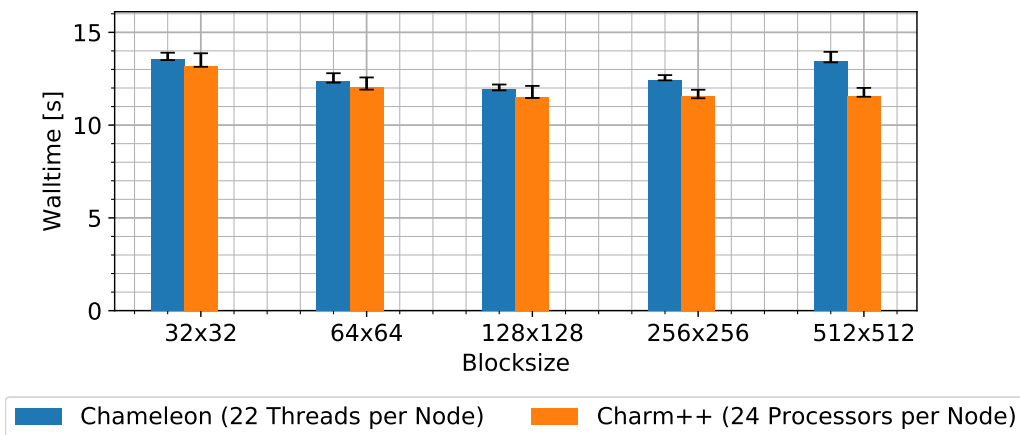


Figure 6.11: Effects of task granularity on performance of frameworks in SWE benchmark. A scenario with a dry fraction of 0.3 and a grid size of 4096x4096 was used and 200 simulation steps were executed. Different block sizes were configured and all versions were executed using 2 nodes.

7 Conclusions and Future Work

In this thesis, three frameworks for task-based parallel programming in distributed memory environments have been introduced and compared with respect to their scheduling and load balancing characteristics.

One advantage of task-based programming is the automatic load balancing capability of the frameworks. Local, global and hierarchical load balancing strategies were discussed. Global strategies work best with smaller numbers of nodes, local strategies scale better but may not achieve a good load balance in certain cases while hierarchical load balancing is the most scalable but also most complex form of load balancing in a distributed memory environment.

Charm++ is a programming framework that provides a message-driven parallel runtime model using migratable objects called chares. Chares can send messages to each other which causes asynchronous execution of functions, which can be interpreted as tasks. The framework offers various global, local and hierarchical balancing strategies that can automatically migrate chares and the load balancing can be triggered either manually or automatically in intervals. The migration of chares is permanent in Charm++.

StarPU is a framework that focuses on task-based programming in a hybrid setup with CPUs and accelerators, but also provides capabilities to support execution in distributed memory setups. The load balancing is performed using a global strategy, but only once when the `taskwait` method is called. Tasks are then distributed among ranks and there is no reactive load balancing.

Chameleon is a framework that enables task-based programming in distributed memory environments with a task model similar to OpenMP offloading tasks. It offers reactive, fine-granular load balancing due to a dedicated communication thread for each rank. Tasks that have been created on a particular rank are either executed locally or migrated to another rank by the Chameleon runtime with a concluding re-migration to the original rank. Load balancing is performed continuously in the background on the communication thread.

The frameworks were compared theoretically as well as empirically using a synthetic and a real-world benchmark. The synthetic benchmark has the property that each task has a constant, low amount of data and a comparatively high computational load. This is favorable for load balancing across ranks. The real-world benchmark has a less favorable ratio of the two properties, but it is a stencil algorithm, which creates tasks that operate on the same data in every iteration.

All three frameworks have shown good scalability in both benchmarks and offer a way to take the task-based programming paradigm from shared memory to dis-

tributed memory systems. Only StarPU has shown a significant overhead compared to a traditional approach using MPI+OpenMP.

StarPU was unable to compensate software-induced load imbalances while Charm++ and Chameleon achieved better performance than the baseline MPI+OpenMP versions of each benchmark with increasing load imbalances. For the synthetic benchmark with imbalances, Charm++ and Chameleon were able to successfully rebalance the load such that the execution time stayed the same regardless of the degree of imbalance.

Charm++ was found to be well-suited for iterative applications due to its permanent data migration. On the other hand, it was unable to compensate simulated hardware-induced imbalances effectively in a synthetic benchmark. Chameleon is particularly well-suited to balance temporary imbalances caused by hardware, but can only compensate smaller load imbalances in iterative algorithms.

StarPU was found to be unable to provide an effective load balancing mechanism for hardware- or software-induced load imbalances.

The effect of task granularities on performance was also examined empirically. As expected, large numbers of small tasks produce too much overhead while few large tasks cause idle time because the load balancing cannot be performed fine-granular enough. For every application there is a task size in between these two extremes, that has the best performance.

7.1 Future Work

One recent trend is the use of accelerators such as GPUs in High Performance Clusters. They may be integrated into the task-based programming frameworks by providing the functionality to automatically migrate tasks not just between ranks but also from hosts to devices. This approach would add another level of complexity but could allow applications to use the benefits of accelerators. The same properties of task-based parallel programming frameworks that were analyzed in this thesis would have to be analyzed for this new setup. StarPU already supports the execution of tasks on GPUs natively with the same automatic load balancing mechanisms as on the CPU version.

Task-based programming is suitable for algorithms with changing load imbalances such as Adaptive Mesh Refinement algorithms. It would be interesting to implement a version of such an algorithm for each of the frameworks and compare their performance under different setups with varying granularity, interference and refinement degrees.

In Big Data frameworks such as Apache Hadoop, task replication is used to compensate for stragglers, i.e. nodes which are significantly slower than others. If a straggler is detected, its assigned tasks are replicated to another node and the results are taken from the task that finishes first. Stragglers may also occur in distributed memory environments of HPC applications and the use of task replication and speculative execution could be investigated.

One disadvantage of Chameleon is the fact that, for each rank, a thread is reserved for communication purposes. Other approaches to achieve reactivity without this much overhead could be proposed and examined. Permanent migration of data or caching in Chameleon could also be investigated as it proved to be a valuable feature of Charm++. For Charm++ on the other hand, temporary migration of chares may be a feature with the potential to better compensate hardware-induced load imbalances.

Acknowledgements

Simulations were performed with computing resources granted by RWTH Aachen University under project thes0563.

Bibliography

- [1] D. Applegate and W. Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on computing*, 3(2):149–156, 1991.
- [2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.
- [3] A. Bar-Noy, A. Freund, and J. Naor. On-line Load Balancing in a Hierarchical Server Topology. *SIAM Journal on Computing*, 31(2):527–549, 2001.
- [4] M. J. Berger and P. Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.
- [5] P. De, V. Mann, and U. Mittaly. Handling OS Jitter on Multicore Multithreaded Systems. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.
- [6] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys (CSUR)*, 47(4):62, 2015.
- [7] Department of Informatics, Technical University of Munich. SWE - A Simple Shallow Water Code. 2013. <https://www5.in.tum.de/SWE/doxy/>, accessed on September 4th, 2019.
- [8] Fredrik Unger, NEC Deutschland GmbH. CLAIX. 2016. https://doc.itc.rwth-aachen.de/download/attachments/28344675/02.NEC_RWTH_Aachen.pdf.
- [9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.
- [10] Intel Corporation. Product Brief: Intel Xeon Scalable Platform. 2017. <https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-scalable-platform-brief.pdf>.
- [11] L. V. Kale and S. Krishnan. *CHARM++: A Portable Concurrent Object Oriented System Based On C++*, volume 28. Citeseer, 1993.

Bibliography

- [12] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. libMesh: a C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers*, 22(3-4):237–254, 2006.
- [13] Ludwig-Maximilians-University Munich (LMU), RWTH Aachen University, Technical University of Munich (TUM). CHAMELEON: A task-based programming environment for developing reactive HPC applications . <https://github.com/chameleon-hpc/chameleon>, accessed on September 26th, 2019.
- [14] A. S. Manne. On the Job-Shop Scheduling Problem. *Operations Research*, 8(2):219–223, 1960.
- [15] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. 2015. <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.
- [16] J. Olden. Performance Analysis of SWE Implementations based on modern parallel Runtime Systems. Bachelor Thesis, Technical University of Munich, 2018.
- [17] OpenMP Architecture Review Board. OpenMP Application Program Interface version 4.0. 2013. <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>.
- [18] OpenMP Architecture Review Board. OpenMP Application Program Interface version 5.0. 2018. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf>.
- [19] Parallel Programming Lab, Dept of Computer Science, University of Illinois. Charm++ Manual. <https://charm.readthedocs.io/en/latest/charm++/manual.html>, accessed on September 4th, 2019.
- [20] C. Pheatt. Intel® Threading Building Blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.
- [21] C. Pohl and K.-U. Sattler. Joins in a Heterogeneous Memory Hierarchy: Exploiting High-Bandwidth Memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, page 8. ACM, 2018.
- [22] D. Rinerson, S. W. Longcor, E. R. Ward, S. K.-R. Hsia, and W. Kinney. High-density nvram, July 12 2005. US Patent 6,917,539.
- [23] P. Samfass, J. Klinkenberg, and M. Bader. Hybrid MPI+ OpenMP Reactive Work Stealing in Distributed Memory in the PDE Framework sam (oa)^ 2. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 337–347. IEEE, 2018.

- [24] University of Bordeaux, CNRS, Inria. StarPU Documentation. <http://starpu.gforge.inria.fr/testing/master/doc/html/>, accessed on September 4th, 2019.
- [25] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.
- [26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.
- [27] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *2010 39th International Conference on Parallel Processing Workshops*, pages 436–444. IEEE, 2010.