UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1621

Advances in Task-Based Parallel Programming for Distributed Memory Architectures

AFSHIN ZAFARI

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018

ISSN 1651-6214
ISBN 978-91-513-0209-6
urn:nbn:se:uu:diva-338838

Dissertation presented at Uppsala University to be publicly examined in ITC/2446, ITC, Lägerhyddsvägen 2, Uppsala, Friday, 2 March 2018 at 10:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Doctor George Bosilca (Innovation Computing Laboratory, University of Tennessee).

Abstract

Zafari, A. 2018. Advances in Task-Based Parallel Programming for Distributed Memory Architectures. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1621. 42 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0209-6.

It has become common knowledge that parallel programming is needed for scientific applications, particularly for running large scale simulations. Different programming models are introduced for simplifying parallel programming, while enabling an application to use the full computational capacity of the hardware. In task-based programming, all the variables in the program are abstractly viewed as data. Parallelism is provided by partitioning the data. A task is a collection of operations performed on input data to generate output data. In distributed memory environments, the data is distributed over the computational nodes (or processes), and is communicated when a task needs remote data.

This thesis discusses advanced techniques in distributed task-based parallel programming, implemented in the DuctTeip software library. DuctTeip uses MPI (Message Passing Interface) for asynchronous inter-process communication and Pthreads for shared memory parallelization within the processes. The data dependencies that determine which subsets of tasks can be executed in parallel are extracted from information about the data accesses (input or output) of the tasks. A versioning system is used internally to represent the task-data dependencies efficiently. A hierarchical partitioning of tasks and data allows for independent optimization of the size of computational tasks and the size of communicated data. A data listener technique is used to manage communication efficiently.

DuctTeip provides an algorithm independent dynamic load balancing functionality. Redistributing tasks from busy processes to idle processes dynamically can provide an overall shorter execution time. A random search method with high probability of success is employed for locating idle/busy nodes.

The advantage of the abstract view of tasks and data is exploited in a unified programming interface, which provides a standard for task-based frameworks to decouple framework development from application development. The interface can be used for collaboration between different frameworks in running an application program efficiently on different hardware.

To evaluate the DuctTeip programming model, applications such as Cholesky factorization, a time-dependent PDE solver for the shallow water equations, and the fast multipole method have been implemented using DuctTeip. Experiments show that DuctTeip provides both scalability and performance. Comparisons with similar frameworks such as StarPU, OmpSs, and PaRSEC show competitive results.

Keywords: parallel programming, task-based programming, distributed memory system, scientific computing, hierarchical data, hierarchical tasks

Afshin Zafari, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0209-6
urn:nbn:se:uu:diva-338838 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-338838)

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I A. Zafari, M. Tillenius, and E. Larsson. Programming models based on data versioning for dependency-aware task-based parallelisation. In: *IEEE 15th International Conference on Computational Science and Engineering (CSE)*, Nicosia, 2012, pp. 275–280.[1]

Contributions
The author of this thesis implemented the distributed memory architecture parts of the programs used in the experiments, performed the corresponding experiments and contributed with writing the related sections.

II A. Zafari, E. Larsson, and M. Tillenius. DuctTeip: An efficient programming model for distributed task parallel computing. Available as arXiv:1801.03578, 2018.

Contributions
The author of this thesis did the implementation and performed the experiments. The analysis and design of the programs was developed in discussion between the authors. The manuscript was written in close collaboration with E. Larsson.

III A. Zafari. TaskUniVerse: A Task-Based Unified Interface for Versatile Parallel Execution. In: R. Wyrzykowski, E. Deelman, et al. (Eds.), Parallel Processing and Applied Mathematics–12th International Conference, PPAM 2017, Krakow, Poland, September 10–13, 2017, Lecture Notes in Computer Science, Springer, 2018, 14 pp., to appear.[2]

Contributions
The author of this thesis is the sole author of this paper.

IV Afshin Zafari, Elisabeth Larsson, Marco Righero, M. Alessandro Francavilla, Giorgio Giordanengo, Francesca Vipiana, Giuseppe

[1] Copyright © 2012 IEEE, reprinted with permission.
[2] Copyright © 2017 Springer, reprinted with permission.

Vecchi. Task Parallel Implementation of a Solver for Electromagnetic Scattering Problems. Available as arXiv:1801.03589, 2018.

Contributions
The author of this thesis did the implementation, performed the experiments and contributed to the manuscript.

V A. Zafari, E. Larsson, Distributed Dynamic Load Balancing for Task Parallel Programming. Available as arXiv:1801.04582, 2018.

Contributions
The author of this thesis did the implementation, performed the experiments, developed the analysis and design of the programs in discussions with the second author, and contributed to the manuscript.

Reprints were made with permission from the publishers.

Software

The software library DuctTeip, for distributed task based programming, is developed as part of this work. It is available at https://github.com/afshin-zafari/DuctTeip.

Contents

1. Introduction to parallel programming

Around the year 2000 it became clear that speeding up computations relying on Moore's law is no longer possible. Further enhancements of the computational speed were made possible by introducing changes in the architecture in the direction of increasing the number of cores in a single processor or via external accelerators. This requires a new way of programming to obtain the most of the available power from the available processors (cores) within a modern computer system. The classical programming paradigms of distributed memory message passing or shared memory OpenMP-type programming need to be supplemented with new techniques for fine grained parallelization. Due to the complexity and size of the scientific computing applications, the change in the computer architecture makes their parallel implementation challenging.

Thread-based parallel programming requires additional considerations from the programmer, not only for identifying the parts of the program that can be parallelized, but also for controlling concurrent accesses to the data accessed by multiple threads to preserve the correctness of the results. The proposed parallel programming approaches during the past two decades, attempt to relieve the programmer from as much as possible of the technical aspects above while maintaining high performance.

Viewing the choice of a specific parallel programming approach as an investment in the software being developed, implies constraints on the obtainable performance and the scalability of the approach. Indeed, approaches with simple concepts and smooth learning curves are acceptable to a wider spectrum of programmers. To survive in the long term, an approach should also address concerns like extendability, reconfigurability and interoperability of the software in order to fulfill the future needs and to easily adapt to new technologies. Scientific computing applications not only request a parallel programming approach to be general in terms of algorithmic flexibility but also to support complex data structures, and high accuracy of the results. Section 1.1 provides a comparison between the existing approaches for parallel programming in terms of the aforementioned constraints.

1.1 Approaches

The development of parallel programming environments has followed several directions. The first direction encompasses various programming paradigms

(frameworks) such as POSIX Threads (Pthreads), OpenMP [34] and the message passing interface (MPI) [31] that mainly provide concurrency mechanisms via an application programming interface (API). Using Pthreads or OpenMP, the programmer can fork multiple threads of execution in a program and using synchronization mechanisms, control concurrent accesses to the data used by those threads. The MPI standard also provides APIs for creating multiple processes and communicating data among them in a distributed memory environment. When using specific APIs in this way, the programmer is responsible for finding the the available parallelism and choosing an efficient concurrency control mechanism.

The second direction is based on the idea to simplify parallel programming by making the detailed technical concerns of parallelization as transparent as possible to the programmer. To this end, some programming frameworks, like Intel TBB [35], SkePU [20], FastFlow [6], HPX [25] and GrPPI [19] use object oriented features, e.g., general parallel design patterns as skeletons or parallel iterators for different types of containers.

A third direction is the development of programming languages that contain inherent parallel constructs. For example, OpenMP and UPC [45] extend the C or Fortran programming languages to support synchronizations and concurrency control. New languages also, like Chapel [15], X10 [47] and CoArray Fortran (CAF) [39] were invented with parallelization in mind and improved the readability, compactness and simplicity of the source code which after compiling can run in different parallel models. This new family of languages uses the Partitioned Global Address Space (PGAS) model for parallelism in which data can be defined and viewed in a global memory address space. This view of data enables the program to access the data as in a shared memory programming style, while the corresponding concurrency controls and communication are inserted transparently by the compiler or source-to-source translator.

A fourth direction in the development of parallel programming environments uses the dependencies of different operations (tasks) in a program and extracts the available parallelism automatically. In these approaches, a graph of dependencies is constructed implicitly or explicitly whose nodes are the operations and the directed edges determine the execution precedence of the operations. Intel TBB [35], Charm++ [16], Microsoft Task Programming Library (TPL) [32] and Google TensorFlow [24] provide interfaces to the programmer to define the dependencies between operations according to the dependency graph. The task-based parallel programming frameworks OpenMP (version 3.0 and later) [34], StarPU [8], OmpSs [14], SuperGlue [41], PaRSEC [13], DuctTeip [49] and Chunks and Tasks [36] abstract the operations to tasks and internally construct the dependency graph (implicitly or explicitly) from the input or output direction of the tasks' arguments. Using the interfaces provided by these frameworks, the programmer can define tasks and specify read or write accesses for each argument. The runtime system can determine the ready-to-run tasks by finding safe concurrent accesses to the data argu-

ments. The PaRSEC task-based parallel programming framework, provides a special script language for describing algorithms in terms of tasks and their input/output arguments. This script is then processed for constructing a parametric task graph (PTG) from which the successors of every task can be found. The runtime system schedules the tasks onto available processors maximizing the data locality in both shared and distributed memory environments.

This thesis is a contribution to the task-based parallel programming research area and includes novel ideas and solutions that are also implemented in the DuctTeip library.

1.2 Task-based parallel programming

Over the past decade, task-based parallel programming has gained a broad and steadily increasing acceptance from scientific application experts. Task-based programming possesses several advantages that make it a viable approach when aiming at implementing and running large scale complex applications on modern computer hardware. First, the asynchronous execution of tasks offers potential performance advantages compared with static approaches, for example, DuctTeip shows better performance than ScaLAPACK as explained in Paper II. Second, its inherent flexibility makes it applicable and suitable for parallel implementations of numerous types of scientific computing problems, such as solving partial differential equations (PDE) [44], Fast Multipole Method (FMM) solutions to N-Body problems [3, 23, 48], simulation of stochastic discrete events [10, 11], the Conjugate Gradient method for solving system of linear equations [4], Finite Element Method (FEM) applications [33], chemistry applications [18], seismic applications [29], image processing [12] and using sparse data structures in direct solvers [27], and matrix multiplication [37].

The above achievements rely on an abstract view of tasks and data which not only allows efficient instantiation of various types of data and operations in a program, but also enables designing and implementing generic algorithms for task scheduling regardless of the actual operations that the tasks are representing. The StarPU [7] framework, e.g., benefits from this and allows the scheduler component of the framework to be chosen among various existing or new schedulers which differ only in the implemented algorithm, or even to construct a scheduler [30] that combines other schedulers modularly, in a tree structure.

The abstract view of data in the task-based programming paradigm empowers the detection of available parallelism in an algorithm by tracking the data usage among tasks and discovering which subset of the tasks can run without conflicting accesses to data. The task abstraction represents any operations in a program. The data abstraction represents any piece of information in a program that may be accessed by several threads and therefore needs to be

protected from concurrent or out-of-order accesses such that the correctness of the final result is ensured. In order for the runtime system to determine which tasks are ready to run, information about both which data is accessed by a task and in what way (read/update/write) needs to be provided by the user.

The information about the tasks and their data arguments can be used by the runtime systems for optimizing task scheduling and execution. The most useful information in this context is the data location that can be used to perform any required data movement among the distributed processors (as in DuctTeip, PaRSEC and OmpSs) and/or accelerators (as in StarPU). In these cases, the data may need to be moved from the memory space of one processor to another to be processed at the destination or to be returned back as the results. In addition to the data location and data usage information, DuctTeip uses the information about the remote data requests in a program to implement an efficient communication strategy with improved bandwidth usage and less work load on the underlying communication library, as described in more detail in Section 2.3. Information about the tasks can also be used for prioritizing the execution of some tasks over others (as in SuperGlue and StarPU), or choosing the proper worker for executing tasks regarding such constraints as processor work load, computational intensity of the task or energy efficiency. Knowledge about resource consumption can be incorporated into the scheduling algorithm to reduce contention as in [43].

The abstract view of tasks and data also allows different frameworks to cooperate in scheduling and running tasks in different parallel environments, [1, 27, 49]. The TaskUniVerse framework in Paper III, generalizes this abstraction to a unified interface that contains all the common functionalities of the task-based frameworks. In this interface, a program can use generic tasks and data for describing an algorithm, and a central dispatcher distributes them to any existing compliant framework for task scheduling and execution (more details are given in Section 4). This interface can make the synchronization, communication and concurrency controls transparent to the application program.

2. The DuctTeip task-based parallel programming framework

DuctTeip is a framework for distributed task-based parallel programming. It uses the concept of *data* for representing shared variables in a program. The data is defined by the programmer using interfaces provided by the framework.

A *task* is an abstract view of the operations on the data. The data that is used by a task form the *arguments* of the task. Tasks should also be defined within the framework through provided interfaces. The *task definition* includes *registering* the read/write accesses to its data arguments and providing a *kernel* which is a callback mechanism used by the framework for performing the computations within the task. Once a task is defined, a new instance is created every time that type of task is needed in an algorithm. *Task generation* refers to both *creation* and *submission* of tasks and is the responsibility of the programmer. It is the responsibility of the *runtime* to control the life cycle of the tasks by storing, scheduling and executing them, as well as cleaning up the finished tasks.

Every data and task belongs to (is owned by) a specific process of those that execute the program in a distributed memory environment. The data *ownerships* are determined by the programmer. When a task's owner is different from the owner of its data arguments, at runtime the *remote data* is transparently *transferred* from its owners to the owner of the task. A task can run only when all its local and remote data arguments are *ready*. A remote data is ready when it is received by the process that owns the task waiting for it.

Data can be partitioned into smaller pieces *hierarchically*. When a *parent* data in this hierarchy is ready, all of its child data are also ready. Tasks can also have hierarchical relationships. When a parent task becomes ready to run, it can submit *subtasks* as children. A parent task is finished when all of its child subtasks are finished. DuctTeip runs in a Single Program Multiple Data (SPMD) model where multiple instances of the program execute the same code for task generation. This SPMD execution model ensures that the task-data dependencies are globally available to all the running instances of the program.

2.1 Data and dependency tracking

The DuctTeip framework uses data-driven task scheduling, where task dependencies are found and tracked through the data used as arguments of the tasks.

In order to generalize and distinguish data in the programs, an abstract *data* class is defined for representing any type of data used by the tasks. The *data* class encapsulates all the required information that is necessary for processing and communicating the data. The *data* class can be used to represent scalars, multi-dimensional arrays, tree or graph structures and any other complex data structure that may be needed by the different tasks. The *data* class can also be used inside other data types to build combined and even more complex data structures needed by the program. Like any other data types, arrays of *data* can also be defined to have an indexed access to different *data* variables. In the DuctTeip framework, a Fortran-like `(i,j)`-indexing interface is defined for the *data* class to provide access to the element at row `i` and column `j` of the array.

Each *data* object has a *version* number which counts the number of read and write accesses to the data both at task submission time and at task execution time. All accesses to a *data* object should be registered by the programmer into the framework that transparently computes task dependencies on specific versions of *data* and tracks the version numbers of the *data* to find which dependencies are fulfilled. The dependency of a task on a particular version of *data* indicates how many read or write accesses should be completed before the *data* becomes ready for that particular task. During the task execution, whenever a task is finished, the version number of all its *data* arguments are incremented and the new versions of the *data* arguments become ready for further read or write accesses. Tasks can run only when all of their *data* arguments are ready. By examining the version numbers of the *data* objects, the runtime finds all the tasks whose data are ready and schedule them onto the available hardware resources to run in parallel.

Figure 2.1 illustrates the versioning system of a program. To the left, program lines that use the *data* `u`, `x`, `y` and `z` are shown, and the rows of the table to the right show the accesses to the variables corresponding to each line.

The 'r' and 'w' letters in the table, indicate the read and write accesses, respectively, to the *data* specified in the columns of the table. The superscript and subscript numbers in r_a^b and w_a^b mean that the task should wait for the version a or higher of the *data* before it can run, and after the access is done the version is changed to b. For example, the read access r_3^4 of `z` at line 5 should wait for a version of at least 3 for `z` before it can be performed, and should change to 4 after the access. Looking at the columns of the table, we see that the version number (superscript) of each *data* is incremented at every access (initiated to zero at the program start).

To determine the version number for which a read access should wait (a in r_a^b), the versioning system looks backward on the accesses and selects the version number of the *data* at the last write access (i.e., b of the previous w_a^b). For example, the read accesses of `y` at lines 3–5 should wait for version 2, determined by the last write access at line 2. Similarly, to determine the version number that a write access has to wait for (a in w_a^b), the versioning

Line no.	Program	Accesses and versions: u	x	y	z
1	`z = x + y + u`	r_0^1	r_0^1	r_0^1	w_0^1
2	`y =` f_1`(x,z)`		r_0^2	w_1^2	r_1^2
3	`x =` f_2`(y,u)`	r_0^2	w_2^3	r_2^3	
4	`z = u + y`	r_0^3		r_2^4	w_2^3
5	`x = y - z`		w_3^4	r_2^5	r_3^4
6	`y =` α `* z + y`			w_5^6	r_3^5
		p_0	p_1	p_2	p_0
		Owners			

Figure 2.1. Illustration of the versioning system. **Left:** The sequential lines of a sample program. **Right:** The version numbers of the *data* variables for the read and write accesses at the corresponding lines of the program. The superscript numbers show the version number after the access is done. The subscript numbers show the version numbers to wait for before the access can be done. The *data* belong to different processors p_0, p_1 and p_2.

system looks backward on the accesses and uses the version number of the *data* at the last read access or the last write access, whichever happened later. For example, the write access to `x` at line 5 should wait for version 3 which is determined by the write access at line 3, and not by the last read access at line 2. Using versions, the system can find which tasks can run in parallel regardless of the actual operations (the f_1, f_2, `+`, `-` and `*`). For example, when the versions of the *data* variables `u`, `x`, `y` and `z` are respectively 0, 2, 2, and 2 (which happens after the execution of line 2), lines 3 and 4 can run in parallel.

2.2 Task submission and execution

In the task-based parallel programing approach, the tasks represent the operations to be performed on data. The *task* class in the DuctTeip framework encapsulates all the required information for scheduling and running tasks. The tasks should register their read or write accesses of their *data* arguments, which is usually done during *task* construction. The programmer can also define subclasses of the *task* class to provide different types of operations on the *data* arguments. There are virtual methods (`run` and `finished`) that are called by the framework when the task is ready to run or is finished. Once a *task* class is defined in a program, new instances of it can be created for the corresponding operations in the program and be submitted to the runtime. There is also a `simulate_run()` method of a *task* which is called at runtime in simulation mode and lets the programmer skip the computational kernel of the tasks if

needed. In simulation mode the runtime takes every *data* as one byte size and together with skipping the kernels, this execution mode can be used for analyzing the *task* and *data* distribution and communication of an application in shorter time than running the actual computations.

The *tasks* in the DuctTeip framework can have hierarchical parent-child relationships where a parent task does not finish until all its children are finished. The kernel of the parent task creates instances of child tasks and submits them as its children to the runtime. As soon as the last running child is finished the parent task is finished too.

2.3 Communication

In the DuctTeip framework both the *data* and *task* objects are connected with *locations* and can be communicated between locations when needed. The location here refers to a process that runs the task-based program using the DuctTeip framework. The initial location of the *data* can be assigned by the programmer to any process and the locations of the *tasks* are transparently determined at runtime.

Usually the data of a problem are partitioned into smaller pieces and distributed over the available processes that run the program. The DuctTeip framework provides interfaces for partitioning one- and two-dimensional *data* into pieces and distributing them over a virtual grid of processes. The location of a *task* is determined as the location of its output *data* argument, and in case of multiple output *data* a user-defined resolution method can be used instead. For example, in Figure 2.1, the *data* are owned by the processes p_0, p_1 and p_2. If t_i, $i = 1, \cdots, 6$ are the tasks corresponding to the lines 1–6 of the program, then the tasks t_3 and t_5 run in p_1 which is the owner of `x`. Similarly, t_1 and t_4 run in p_0 and t_2 and t_6 run in p_2.

When a *task* is submitted to the framework, the runtime can deduce the required communication of the *data* arguments that are not located at the same location as the task. Requesting to read a specific version of a *data* object by a process from another one is handled by the *listener* class in the DuctTeip framework. On task submission, the runtime creates a *listener* at the *data* owner for every *data* access that requires the *data* to be sent to the remote owner of the *task* that needs the *data* . For example, to run t_2 in the example shown in Figure 2.1, a listener for version 0 of `x` is created at p_1 and a listener for version 1 of `z` is created at p_0. Since p_2 owns the output data of t_2 (i.e., `y`), t_2 runs in p_2 and the `x` and `z` data corresponding to these listeners are sent to p_2.

During the execution time, when the version of a *data* argument is upgraded after finishing a *task*, the process that executed the task sends the new version of the output *data* to all the remote requesting processes by checking the received *listeners*. Using *listeners* allows the runtime to detect duplicate requests

for the same version of the data from a single process and thus to avoid redundant communication of the same data to the same destination processes. For the example in Figure 2.1, t_3 and t_5 run in p_1 and both request version 2 of y from p_2. In these cases, the runtime sends the *data* only once but increments the version number of the *data* for each listener. The slight overhead of holding the *listener* information (a tuple of owner, data, version and requester) is compensated by avoiding large amounts of unnecessary communication. For example, in a Block LU factorization of a matrix A with $n \times n$ blocks where every block is a *data*, at each step k of the algorithm, the factorized A_{kk} is used by all the $A_{k+1:n,k}$ and $A_{k,k+1:n}$ blocks resulting in $2(n-k)$ redundant read accesses (the number of reads by the blocks in the corresponding row and column), and each block in the updated $A_{k+1:n,k}$ and $A_{k,k+1:n}$ blocks is used for updating the remaining sub-matrix $A_{k+1:n,k+1:n}$ which results in $(n-k-1)$ redundant read accesses per updated block. Therefore, the total cost of unnecessary communication to run n steps of the algorithm with block-cyclic ownership of data, is $\mathcal{O}(n^2)$ for factorizing the whole matrix (i.e., $k = 1, \ldots, n$).

2.4 The DuctTeip runtime system

The runtime of the DuctTeip framework administrates different activities during the execution of the program including managing memory needs, handling concurrency, controlling the computations and communication and managing transition of program states.

All the *data* definitions and *task* submissions of a program should be enclosed by initialization and finalization of the DuctTeip runtime. On initialization, the program's main thread creates a new thread for administration purposes and continues running the user code. All the required processing of the *data* and the *tasks* at runtime occur in the administration thread concurrent with task submission in the main thread.

The administration thread uses asynchronous communication for sending and receiving messages using MPI in the background[1]. Since during the development of the DuctTeip framework there was no fully functional multithreaded support in the common MPI implementations, the administration thread uses non-blocking MPI calls and periodically checks the state of pending requests for sends and receives. Whenever any of the initiated communication requests complete, the administration thread carries out the consequent tracking of the task dependencies and the task execution.

The hybrid distributed and shared memory parallelization is enabled by allowing the *task* kernels to run in parallel using shared memory parallelization techniques. To gain high performance and productivity, DuctTeip enables us-

[1]The functionality provided by the MPI interface is encapsulated in one class and the signature of the MPI calls is made transparent to the other parts of the framework, so it can be replaced with any other communication system if needed.

ing the SuperGlue task-based framework for parallelization at the shared memory level. The kernel of a *task* divides the *data* arguments into smaller pieces and creates instances of SuperGlue tasks and then submits them to the SuperGlue runtime as sub-tasks (children). The parent-child relationships of the tasks can be handled by the SuperGlue subtask feature or alternatively by the DuctTeip runtime. In either case, when a *task* is finished then its `finished()` method should be called to notify the runtime.

2.5 Memory management

In the DuctTeip framework, to avoid unnecessary performance overhead from memory management operations, the memory for the contents of every *data* object is managed efficiently as follows. The memory is allocated at contiguous addresses to avoid packing/unpacking in communication. The communication message of a *data* object contains a meta-data header part that holds useful information about the *data*, such as the version number and the owner process. The size of the allocated memory for the *data* object is large enough to include both the header and the content parts of the *data*. Using different offsets, this memory location can be used for both the communication message buffer and the data contents in the computations.

There are also cases where a new location for an already allocated *data* object is needed. One such case is when new versions of the *data* are received while the older ones are still in use by some *tasks*. In these cases, the runtime keeps the received version temporarily in a new memory location until the in-use memory of the older version is released. To save the time spent on allocating and deallocating memory, the runtime creates a pool of preallocated memory locations to provide the memory used and requested by the *data* objects. The pool is constructed with a predefined number of memory slots when the *data* objects are defined and can be enlarged transparently when more positions are requested than available. The memory items in the pool can be requested when new memory allocation is needed and can be returned back to the pool when they are no longer needed.

3. Dynamic load balancing (DLB)

Writing a program to run in parallel does not necessarily lead to optimal utilization of the computational resources of the underlying hardware, particularly when the work load among the execution threads is not evenly distributed. Load-balancing is, therefore, a concern in both algorithm design and implementation of a parallel program. At algorithm design, the work load distribution is statically determined and encoded in the program. Dynamic load balancing is also possible, where the program can redistribute the work load at run-time. An overview of various issues related to dynamic load balancing is given in [5]. While static load balancing can take advantage of information from a specific application in efficiently redistributing the work load, dynamic load balancing is independent of the application and balances the work load without assuming any specific foreknowledge about the application.

Dynamic load balancing can be implemented based on data migration [9, 28] or work migration [26, 36]. Migration of computational resources is also investigated in [40, 21, 38, 22], where the resources (threads or CPUs) not used by a blocked MPI process within a computational node are used for running other MPI processes within the same node [21]. This approach alone cannot provide global load balance, but can improve utilization within the node, and can adjust global imbalance depending on the mix of processes at the node.

Whether data migration or work migration is used, information about the current work load distribution over the participating threads or processes is needed, to move the extra work/data from the overly loaded processes to the less loaded ones. In shared memory environments, the current work load is globally visible to all the threads and an optimal migration can be decided easily. In distributed memory environments, however, both the knowledge about the current work load of processes and the migration itself need communication between participating processes which consequently require efficient techniques in the design and implementation of a dynamic load balancing approach. In addition, meeting the generality requirement for a programming framework like DuctTeip introduces more challenges for the dynamic load balancing implementation. In this part of the thesis, we describe some of these challenges and our solutions that are implemented in the DuctTeip framework.

3.1 Distributed dynamic load balancing

The location of the *data* and *tasks* that is initially set at the task generation time may not necessarily lead to en even distribution of the work load among

all processes. Furthermore, the load in a complex algorithm can vary between processes over the execution time for a given data distribution. Therefore, balancing the work load at runtime can result in shorter execution time by migrating *tasks* from the busy processes to the idle ones. A process is considered busy (idle) when the number of its ready-to-run tasks is greater (less) than a predefined threshold. A DLB procedure starts with finding a pair of idle-busy processes. When a pair of busy-idle processes is found, a number of ready-to-run *tasks* (which is not greater than the threshold amount) together with their *data* arguments are migrated from the busy process to the idle one and the output *data* of the migrated *tasks* is returned back to the busy process when the migrated *tasks* are finished.

Before the migration starts, the pairs of busy and idle nodes should be formed. To determine the current work load of every process, DuctTeip refrains from using non-scalable all-to-all collective operations like `MPI_Allgather` with communication costs of $\mathcal{O}(PlogP)$ for P processes. Instead, the busy and idle processes independently look for the other parties by communicating with randomly selected processes. This approach makes no assumption on the virtual topology of the processes but assumes fixed communication cost for any process pair. In cases where, for example, the communication costs differ with the network distance between the processes, the idle or busy processes look for the other parties among their nearest neighbors.

$$\mathscr{P}(k) = \frac{\begin{pmatrix} P-K \\ n-k \end{pmatrix}\begin{pmatrix} K \\ k \end{pmatrix}}{\begin{pmatrix} P \\ n \end{pmatrix}}. \tag{3.1}$$

The random procedure of finding idle (busy) processes follows the hypergeometric probability distribution (3.1), which describes the probability of k successes of n tries without replacement in a configuration of totally P processes out of which K are idle (busy). The probability of at least one success ($k = 1$) out of n tries is $1 - \mathscr{P}(0)$ (the complementary probability of failure), which increases with n, as depicted in Figure 3.1 for different K/P ratios. Obviously, when the K/P ratio is low due to a small number of idle (busy) processes, the $(P-K)/P$ ratio would be large for the busy (idle) processes. Therefore, when a busy (an idle) process searches for an idle (a busy) process with low chance due to small $K/P = \alpha \ll 1$, there would be high chance for an idle (a busy) process to find a busy (an idle) process due to large $(P-K)/P = 1-\alpha$. This reasoning is used in DuctTeip to set an upper limit for the number of tries (n) in finding idle/busy processes (according to Figure 3.1, five tries is a reasonable choice even for $\alpha = 50\%$). The idle (busy) processes try to find a busy (idle) process by sending specific messages to randomly chosen processes. Each idle (busy) process tries to find a busy (idle) process up to a certain number of failures and then becomes silent for a predefined time interval, while it

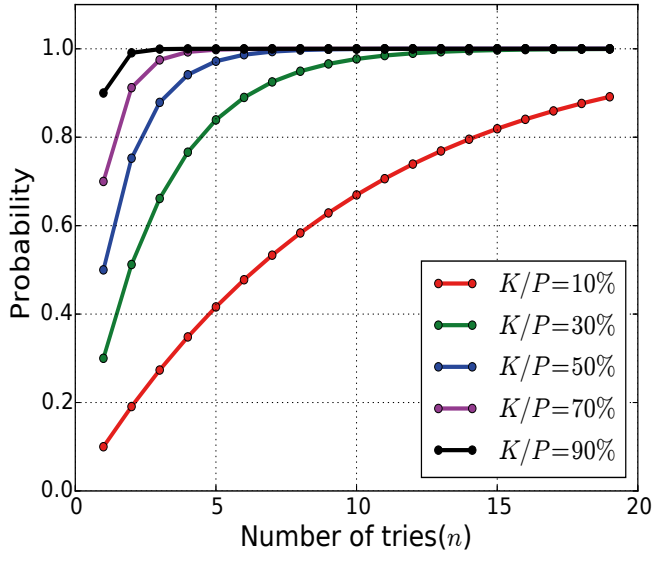

Figure 3.1. The probability of success in a random search for finding at least one idle (busy) process when there are $P = 100$ processes in total with K of them being idle (busy). After five tries ($n = 5$), there is more than 80% chance for a successful search for $K/P \geq 30\%$.

may still be found by other busy (idle) processes. By gathering performance measures (e.g., the average speed) for the computations and communication at runtime and assuming that all processes are running on similar hardware, the busy process can select *tasks* for migration whose outputs *data* are estimated to be received earlier than if the *tasks* would run locally and compute their output *data* locally.

3.2 Dynamic load balancing in practice

Dynamic load balancing in distributed memory architectures requires explicit communication between processes both for managing the workflow and for migrating *tasks* and *data*. The workflow of a typical load redistribution consists of finding the idle/busy process pair, exporting some *tasks* and their *data* arguments from the busy process to the idle process and running the imported *tasks* and returning their output *data* from the idle process to the busy process. The size of communicated messages for the step of finding the idle/busy process is fixed and independent of the application. Therefore, the communication cost of this step of the DLB workflow is constant. However, communication of *data* (for exporting and returning/importing them back) in the other steps of the workflow depends on the size of the *data* being communi-

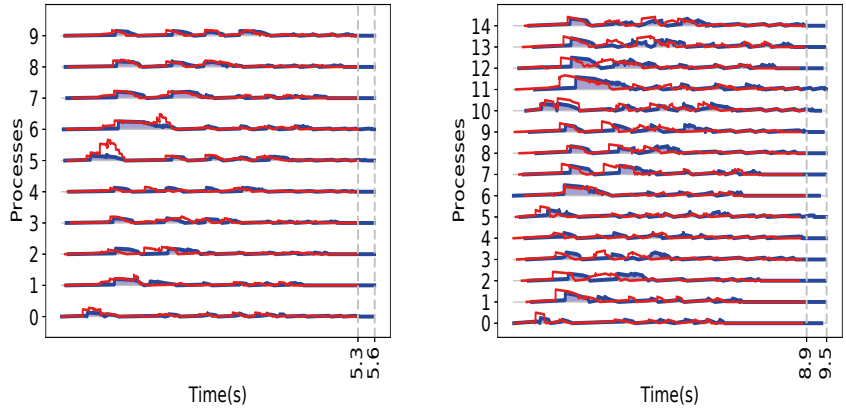

Figure 3.2. The work load of the Cholesky factorization application without DLB (filled blue curves) and with DLB (thin red lines). The highest work load is 10 and the threshold is taken as 5. To the Left: Factorizing a 20000×20000 matrix using 10 processes in a 2×5 process grid (5.3% speedup by using DLB). To the right: Factorizing a 30000×30000 matrix using 15 processes in a 3×5 process grid (6.3% speedup by using DLB).

cated. Therefore, different numbers and sizes of the migrated *data* even within an application require different times for communication. This together with the random selection of finding the idle/busy processes, make the result of the DLB procedure nondeterministic. To illustrate the unpredictability of the DLB success, examples of successful DLB attempts for two configurations of an application are shown in Figure 3.2 and examples of successful and unsuccessful DLB attempts are presented in Figure 3.3.

In Figure 3.2, the work loads (the number of ready-to-run tasks) over time of a block Cholesky factorization application are shown. The application is executed by 10 computational nodes in a virtual 2×5 process grid. Since in this factorization, the matrix blocks are communicated between processes along both the rows and columns of the matrix, using non-square process grids results in an asymmetric/imbalanced communication and computational work load among processes. Another experiment with 15 computational nodes arranged in a 3×5 process grid is conducted and its results are shown also in Figure 3.2. Using DuctTeip with DLB for these experiments results in a 5.3 and 6.3%, respectively, shorter execution time compared with not using DLB. Figure 3.3 shows experiments where the results of the DLB attempts for one application, with the same set of parameters, are different. The Cholesky factorization application in this experiment is executed by 11 processes forming a 11×1 process grid, and hence each process updates the blocks of a row left of the main diagonal of the input matrix. Although this arrangement of processes has higher success probability for the DLB attempts due to an imbalanced

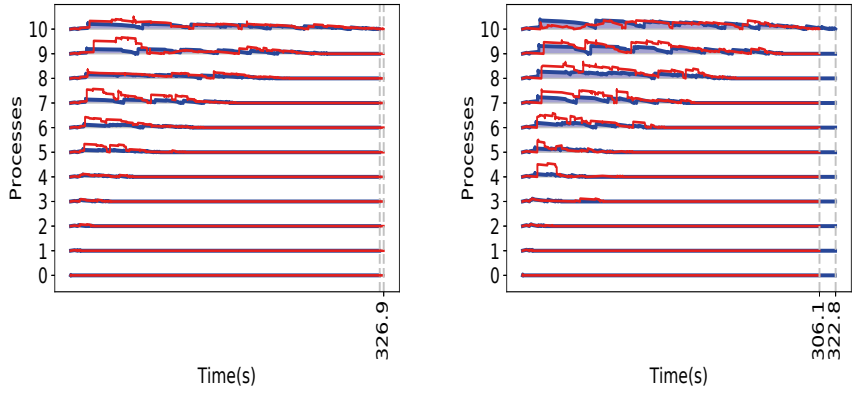

Figure 3.3. The work load of the Cholesky factorization application for an 110000×110000 matrix using 11 processes in an 11×1 process grid without DLB (filled blue curves) and with DLB (thin red curves). Two instances of the application execution with DLB are presented, one unsuccessful (left) and one successful (right) with 5.1% shorter execution time using DLB.

distribution of blocks, the total execution time is only reduced in one of the attempts (right figure).

4. The unified interface for task-based programming

The abstract view of tasks and data used in the task-based parallel programming framework can be generalized to a unified task programming interface to enable different runtime systems to cooperate with each other for scheduling and executing tasks on different underlying hardware. Using such an interface, a program can be written once in terms of generic tasks and data and then be executed by different types of computer hardware using corresponding task-based runtime systems which have implemented the interface.

The TaskUniVerse (TUV) interface that we provide divides the software development into three layers: the application layer, the technical layer and task-based frameworks, as shown in Figurer 4.1. The interface in the middle layer decouples the application layer at the top from the task-based frameworks at the bottom which in turn hide the technical details of parallel programming for the underlying hardware. In this model, the taskified versions of the operations are provided by the middle layer to the application layer via ordinary subroutines while on the other side of the middle layer, the generic tasks move back and forth to the frameworks or their wrappers.

The unified interface defines protocols for working on generic data (*g-data*) and generic tasks (*g-tasks*). The program can use the interface for defining, partitioning and passing *g-data* as arguments to *g-tasks*. Registering the accesses of the *g-tasks* to their *g-data* arguments and submitting the *g-tasks* are also defined in the interface. There are also protocols for notifying the dispatcher (described later in this section) that a *g-task* is ready to run or is finished. To have a hierarchy of *g-tasks* a generic *splitter* can be used to further divide a ready-to-run *g-task* into child *g-tasks* and submit them to the runtime system.

A central customizable dispatcher object in the unified interface bridges the application program and the runtime systems and is responsible for orchestrating the available compliant runtime systems for managing the *g-data* and the *g-tasks*. Every compliant runtime system only talks to and hears from the dispatcher for working with the *g-data* and the *g-tasks* and is not aware of the other available runtime systems. The dispatcher forwards the ready-to-run notifications from one runtime to another or it can split the ready *g-tasks* and call the submit interface of another runtime. The flow of the *g-tasks* and the *g-data* can be customized by the programmer through extending or replacing this dispatcher object.

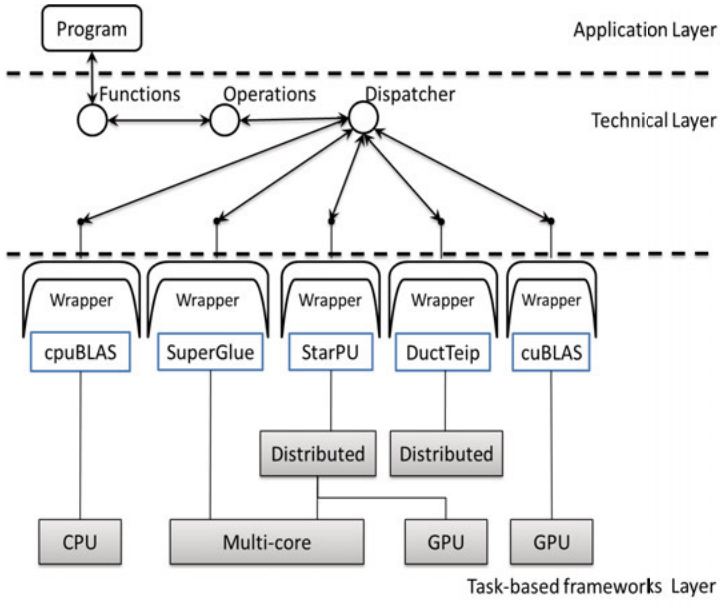

Figure 4.1. An overview of the TUV model. The TUV interface with task-based frameworks unifies the cooperation of the Dispatcher and any framework run-time system. A single program in the Application Layer can be executed in various parallel environments using combinations of frameworks (or their wrappers).

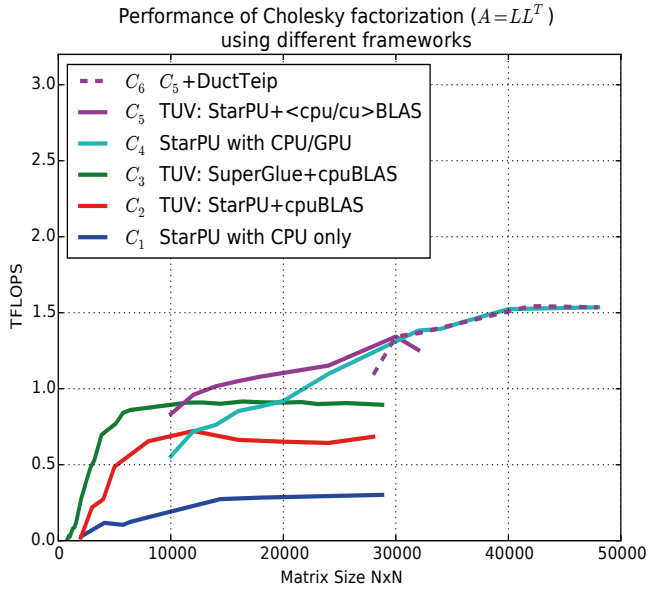

Figure 4.2. Comparison of mixing different frameworks for running a single Cholesky factorization program.

In the unified programming model, the three layers are decoupled from each other, resulting in (potentially) independent software development for application domain experts, vendors and framework developers. It can also make the application software more tolerant to future changes in the underlying hardware.

4.1 Implementation and experiments

To verify the feasibility of the unified interface, we developed the wrappers around the frameworks depicted in Figure 4.1 and implemented the unified interface for combining them. A single program for Cholesky factorization is written and executed in one multi-core computational node with/without GPUs using several configurations (C_1–C_9) of the mixed frameworks as shown in Figures 4.2 and 4.3. In configurations C_1–C_3 matrices up to 30000×30000 elements are factorized using the StarPU framework without the TUV interface (C_1), the StarPU wrapper within TUV (C_2) and the SuperGlue wrapper within TUV (C_3). For factorizing larger matrices in one computational node, parts of the computations are executed in the GPUs by using the StarPU framework without the TUV interface (C_4), the StarPU wrapper within TUV (C_5) and the DuctTeip wrapper and StarPU wrapper within TUV (C_6). In Figure 4.3, the performance of different frameworks is compared for Cholesky factorization of matrices in a distributed memory environment. The factorization is performed by the StarPU framework (C_7), the DuctTeip and StarPU wrappers within TUV (C_8) and the DuctTeip and SuperGlue wrappers within TUV (C_9).

These experiments show that the program is not only independent of the underlying frameworks, but it can also gain the best performance by using different mixtures of frameworks. This enables the program to easily adapt to future versions of the frameworks (probably with improved performance) and provides more freedom to the framework developers to upgrade their libraries without affecting the legacy software as long as the frameworks stay compliant with the unified interface.

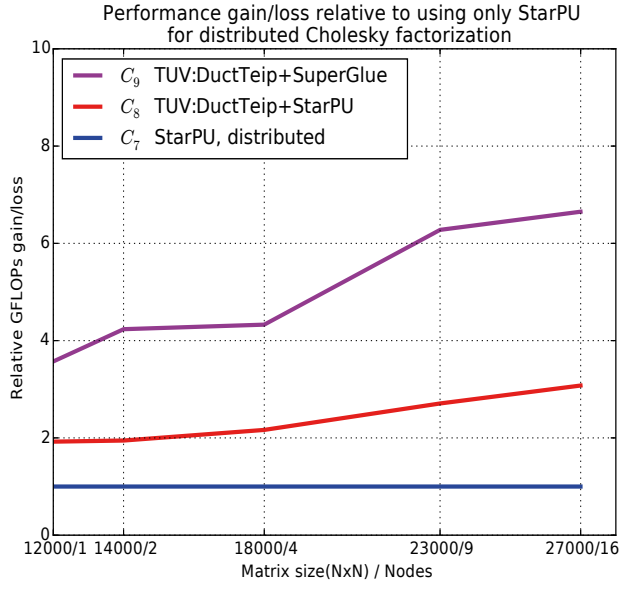

Figure 4.3. Comparison of mixing different frameworks for running a single Cholesky factorization program in distributed memory environment.

5. Benchmarks

The concepts and design of the DuctTeip framework are evaluated through different types of applications in terms of non-trivial task-data dependencies, different data and task types and different granularities. The Cholesky factorization is used as a complex task-data dependency algorithm, which benefits from the ease of implementation and the flexibility of the task parallel programming model.

To assess the correctness and performance of the framework in using different data types and iterative algorithms, a shallow water time-dependent PDE solver with a sparse discretization matrix is used. A third benchmark is the implementation of a Fast Multipole Method (FMM), where an ongoing work for implementing 3D problems for distributed memory architectures is in progress after a successful pilot study for 2D problems for the shared memory architectures.

5.1 Benchmark 1: The Cholesky factorization

To verify the correctness of extracting the dependencies of *tasks* and *data*, the Cholesky factorization algorithm is used with various numbers of *tasks* and *data* in two hierarchical levels both in distributed and shared memory environments. The input matrix of the algorithm is partitioned into $B \times B$ blocks at the first level of the hierarchy and the blocks are used as *data* objects in the DuctTeip runtime. Every block in the first level is also divided into $b \times b$ blocks that are used by the SuperGlue runtime at the next level. When a *task* object at the first level is ready to run, its kernel creates the SuperGlue tasks that use the partitioned data at the second level of the hierarchy. This hierarchy of data enables the program to choose block sizes that are efficient for the communication and computations separately.

The Cholesky factorization implementation shows that using task-based programming enables the programmer to describe the algorithm easier by relieving her from finding the potential parallelism in different parts of the algorithm. The efficiency of the design is also verified by achieving high performance and scalability of executing the Cholesky application on more processors as depicted in Figures 5.1 and 5.2. Frameworks with different programming models and implementations are chosen for executing the Cholesky factorization in distributed memory environments. ScaLAPACK is a distributed

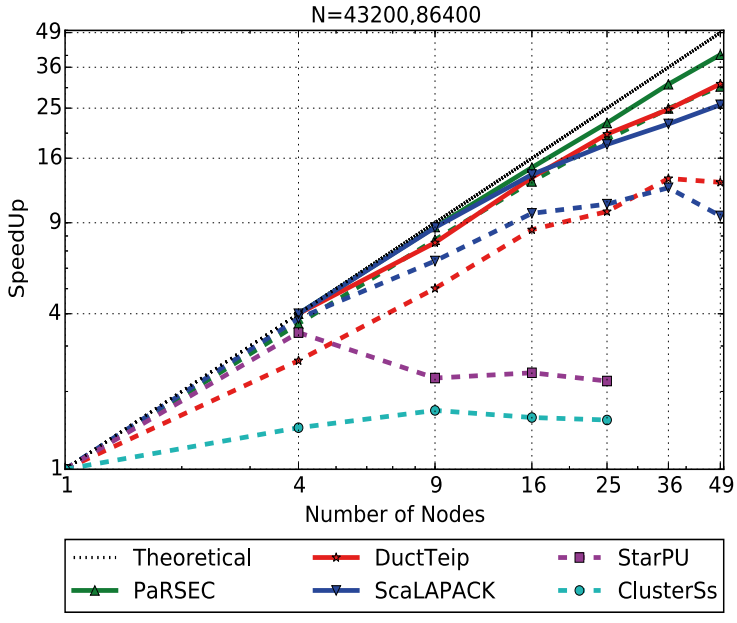

Figure 5.1. Comparison of Speedups for Cholesky factorization benchmark obtained by using different frameworks.

version of LAPACK and is used as a reference even though it is not a task-based framework. In PaRSEC, an algorithm is written in a special scripting language and a parametrized task graph (that can be used to generate the successors of a task) is extracted and embedded into the generated C source code. DuctTeip and ClusterSs use hierarchical partitioning of tasks and data. ClusterSs uses a centralized task submission from one master process to all other participating processes. The figures show that the techniques used in DuctTeip (hierarchical partitioning, decentralized task submission and efficient communication) result in better performance and scalability than what is achieved for StarPU and ClusterSs here.

The task-based Cholesky factorization benchmark is also used for verifying the feasibility and usefulness of the unified interface for task-based programming, introduced in Section 4. In this set of experiments, the input matrix is also partitioned into two hierarchical levels and the different runtimes use the data only within a particular level. The DuctTeip, SuperGlue and StarPU runtimes that are used in these experiments are not aware of each other when working with the data or scheduling and running tasks in different levels. The Cholesky benchmark is written once in terms of generic data and tasks and is executed by several combinations of the runtimes in different parallel computing environments. The fruitfulness of the programmer's choice for combining different runtimes to achieve the best performance is also elaborated and demonstrated in these experiments.

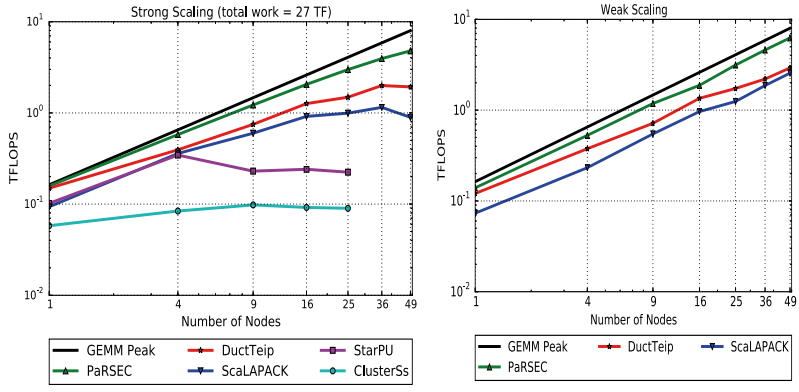

Figure 5.2. To the left: Strong scaling of Cholesky factorization benchmark using different frameworks. To the right: Weak scaling of Cholesky factorization benchmark using different frameworks.

5.2 Benchmark 2: The shallow water equations

The solution of the shallow water equations is challenging both in terms of choice of methods and for parallel implementation. The solution used for verifying DuctTeip is originally implemented in [44] using the SuperGlue framework for parallelization in both distributed and shared memory environments. The DuctTeip implementation uses SuperGlue for the shared memory parallelization and handles the distributed communication using the DuctTeip *data* and *task* objects described in Section 2.1 and 2.2.

The main programming challenges when solving the shallow water equations are the sparse structure of the difference operator resulted from finite difference discretization methods, that limits the computation to communication ratio, and the time-step iterations. To avoid losing available parallelism (as found in [44]) by partitioning the system matrix regarding its average number of non-zeros per row, the matrix is partitioned into 2D blocks in two hierarchical levels which leads to a different density of elements in the blocks. The elements of the operator in every block are stored in the compressed sparse row (CSR) format in contiguous memory locations and since the difference operator is only being read throughout the execution, no *data* object is needed for the blocks. The fourth order Runge-Kutta scheme is used for time-stepping and has the following structure:

```
f(F1,H);                    // F1 = f(H)
add(H1, H , 0.5*dt, F1);    // H1 = H + 0.5*dt*F1
f(F2,H1);                   // F2 = f(H1)
add(H2, H , 0.5*dt, F2);    // H2 = H + 0.5*dt*F2
f(F3,H2);                   // F3 = f(H2)
add(H3, H ,     dt, F3);    // H3 = H +     dt*F3
```

```
f(F4,H3);                  // F4 = f(H3)
step(H,F1,F2,F3,F4);       // H  = H + dt/6*(F1+2*F2+2*F3+F4)
```

The right hand side function `f(F,H)` consists of sparse matrix-vector multiplications of differentiation matrices and the vectors of unknowns `H1`–`H4`. The data distribution of the matrix and vectors is shown in Figure 5.3.

To implement the time-step iterations, a customized *data* object is defined that holds only the time-step number and is owned by (located at) every process, hence no send or receive is needed for it. To run the time-steps sequentially, one *task* for each time-step is defined that writes to a single instance of this *data* type. When a time-step *task* is ready to run, it creates child tasks for solving the PDE for a specific time-step. These *tasks* basically implement sparse matrix-vector products and vector updates in the Runge-Kutta method. The vectors are also partitioned into two hierarchical levels which are used by the DuctTeip and SuperGlue tasks at each level.

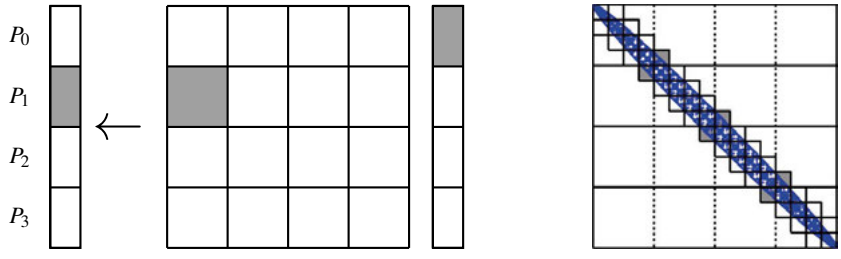

Figure 5.3. Left: One block row of each data structure is assigned to each process P_i. The shaded blocks illustrate input and output in the block matrix multiplication. Right: An actual matrix structure together with a partitioning into level 1 blocks is shown. Tasks are only generated for blocks with non-zero elements. Tasks associated with the shaded blocks require communication between different processes.

The speedup for the shallow water benchmark as shown in Figure 5.4, is close to linear for a small number of computational nodes and gradually moves away from linear when the number of computational nodes increases. Larger problem sizes show improved speedup due to larger task sizes. Figure 5.5 shows an execution trace for the largest problem size using two computational nodes. This figure shows that there are no gaps (idle time) between tasks or between different time-steps (different colors in the figure) thanks to the dependency-aware task scheduling of both DuctTeip and SuperGlue. More specifically, when all required data of a task are ready, the task runs independently of other tasks in the current time-step or another.

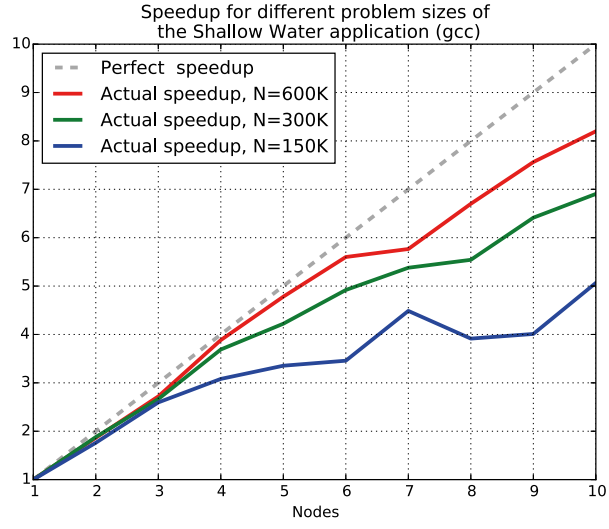

Figure 5.4. Speedup for the Shallow Water benchmark.

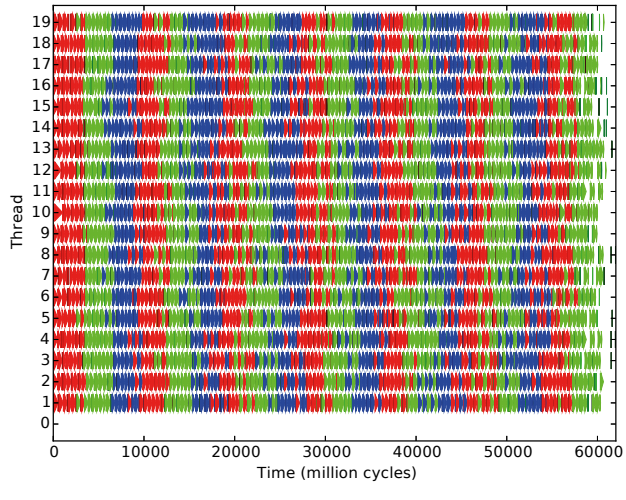

Figure 5.5. The execution trace of one node running the Shallow Water benchmark with the largest problem size. Each triangle shows execution duration of a task and different colors present different time steps.

5.3 Benchmark 3: The Multi Level Fast Multipole Method in 2D

The multilevel fast multipole method (MLFMM) approach to solve N-Body problems reduces the computational complexity of the problem to $\mathcal{O}(NlogN)$ by aggregating the impacts of groups of particles into a single equivalent pole and provide approximations for the impacts of poles on each other. This approach is used here for solving a simplified computational electromagnetic (CEM) problem that resembles problems that arise in antenna design, where the characteristics of an electromagnetic field is to be found by summing up the pairwise interactions of all points in the field. In this approach, the whole domain is enclosed in a 2D box which is partitioned into smaller boxes through bisections in each dimension. This partitioning continues up to a certain number of levels or up to a certain lower limit on the number of points per box. These boxes are held in an quad-tree data structure to enable efficient implementation of the parent-child and near or far relationships among boxes.

Based on the tree structure, the computations in this approach are performed both at every level and from one level to another. The algorithms designed for these computations start with the bottom most level and perform near-field point-to-point computations (P2P), then the result of the child boxes are aggregated in their parents (child-to-parent, C2P, operations). The effect of boxes on each other are computed by translation (XLT) operations, for all boxes that do not contribute to the near field at that level. When the effects of all the boxes are aggregated in the root box of the tree, the root box distributes its effect into its children (parent-to-child, P2C, operations) each of which distributes their new computed fields into their children down to the leaf nodes of the tree where the effects of the parent boxes are summarized into the actual spatial points (SUM operations).

The computations at the finest level of the tree (P2P and SUM operations) are all independent and can thus run in parallel. The child-to-parent operations are independent for different parents but updating a particular parent can only be done one at a time. The translation operations should be done in order for the same target box, otherwise they are all independent. Therefore, there are many potentially parallel operations for different boxes at different levels that necessitate a careful implementation to be realized. Using the parallel `for` clause from OpenMP, for example, cannot exploit the available parallelism across the levels. The main advantage of task-based implementation of the FMM method is that using nested loops in the program to formulate the operations does not introduce implicit synchronization points at any part of the program. As soon as the data dependencies of tasks are fulfilled, they can run regardless of their position in the tree or the relationships between their data arguments. This vital characteristics of task-based parallel programming is used with the SuperGlue framework and the OpenMP tasks in Paper IV to mix the near-field and far-field computations and achieve the best possible perfor-

mance from the multicore processors in a shared memory architecture. One of the advantages of using the SuperGlue framework compared with OpenMP tasks, is the possibility of defining commutative tasks. These are tasks that update the same data, and can therefore not run concurrently, but can be reordered. Using these type of tasks, e.g., for the children-to-parent operations (C2P) in the FMM algorithm, enables them to be reordered.

6. Summary of papers

6.1 Paper I

A. Zafari, M. Tillenius, and E. Larsson. Programming models based on data versioning for dependency-aware task-based parallelisation. In: *IEEE 15th International Conference on Computational Science and Engineering (CSE)*, Nicosia, 2012, pp 275–280.

This paper explains the benefits of task-based parallel programming in relieving the application developer from thinking about the technical details of different parallelization techniques while providing access to the high performance of multicore computer systems. A family of task parallel programming models is derived, where data dependencies are manged through data versioning for both shared and distributed memory architectures. It is described that using this type of model is beneficial since it is easy to represent different types of dependencies and that scheduling decisions can be made locally. Experiments show that a thread parallel shared memory implementation as well as a hybrid thread/MPI distributed memory implementation scale well on a system with 64 cores. A comparison of a pure MPI implementation with a hybrid thread/MPI implementation that shows better performance/scaling for the hybrid version for systems with a large number of cores, is also presented.

6.2 Paper II

A. Zafari, E. Larsson, and M. Tillenius. DuctTeip: An efficient programming model for distributed task parallel computing. Available as arXiv:1801.03578, 2018.

This paper explains the design aspects of the DuctTeip framework and demonstrates its techniques for efficient extraction of available parallelism from an application, and efficient execution of the application on distributed memory systems. More specifically, the data *versioning* technique, that simplifies tracking the dependencies between tasks and data is described. Hierarchical data partitioning and hybrid MPI-Pthreads parallel execution of the program are the techniques which result in improved performance and scalability. It is also discussed how the *listeners* can help to not only avoid putting high loads on the underlying communication library, but also avoid redundant data transfers among distributed processors. The fruitfulness of these techniques is examined by implementing and running relevant experiments.

6.3 Paper III

A. Zafari. TaskUniVerse: A Task-Based Unified Interface for Versatile Parallel Execution. In: R. Wyrzykowski, E. Deelman, et al. (Eds.), Parallel Processing and Applied Mathematics, 12th International Conference, PPAM 2017, Krakow, Poland, September 10–13, 2017, Lecture Notes in Computer Science, Springer, 2018, 14 pp., to appear.

In this paper, a unified interface is proposed for task-based parallel programming. This interface decouples the application programs from the task-based parallel programming frameworks by providing interfaces for the programming constructs of task-based parallel programming that are common to all such frameworks. Multiple frameworks can be combined together to run a single application. The proposed interface allows the frameworks to cooperate without knowing each other. Confirmed by the experiments in the paper, this allows the application to always achieve the best performance out of the existing frameworks and parallel computing resources without refactoring the application program. This can assure the application owners that their legacy software is easily adaptable to the changes in the underlying hardware and software in the future while still preserving the high performance.

6.4 Paper IV

Afshin Zafari, Elisabeth Larsson, Marco Righero, M. Alessandro Francavilla, Giorgio Giordanengo, Francesca Vipiana, Giuseppe Vecchi. Task Parallel Implementation of a Solver for Electromagnetic Scattering Problems. Available as arXiv:1801.03589, 2018.

In this paper, we used a task parallel approach for implementing a solver based on the multilevel fast multipole method. This method performs hierarchical decompositions through bisections in each dimension of the domain and uses a quad-tree data structure for representing the resulting regions (boxes). The root of the tree encloses the whole domain, and the leaves at the finest level of the tree contain the individual spatial points of the domain. There are different computations for every pair of boxes at each level of the tree or across the levels both downward and upward. This leads to complex dependencies between the computations all over the tree levels while providing a large amount of potential parallelism at the same time. We compared OpenMP tasks and the SuperGlue task-based framework in terms of productivity and performance with a set of experiments and presented the results.

6.5 Paper V

A. Zafari, E. Larsson, Distributed Dynamic Load Balancing for Task Parallel Programming. Available as arXiv:1801.04582, 2018.

In this paper, we extended the DuctTeip task-based parallel programming framework to enable it to balance the work load of processes dynamically at run-time, which is useful for cases where a static even distribution of work among the participating processes is not possible or easy. The approach used here is independent of the application, the number of the processes and the network topology. In order to balance the load (the number of ready-to-run tasks), tasks from the busy processes are migrated to the idle (less busy) processes. To achieve scalability to large numbers of processes, we used a random selection strategy for finding pairs of busy and idle processes. We analytically showed that this random selection with high probability succeeds within a few tries, hence has a fixed overhead regardless of the network topology and the application. We introduce and investigate different parameters that can affect the success of the work load redistribution in shortening the total execution time. The corresponding experiments and their results are presented and discussed.

7. Summary in Swedish

Parallell programmering är nödvändig för långa vetenskapliga simuleringar av storskaliga problem. Skillnaderna jämfört med sekventiell programmering gör parallellprogrammeringen utmanande. Programmeraren behöver skapa flera trådar som exekverar parallelt och samtidigt styra tillgången till delade data och bestämma var och hur trådarna får anslutas till datat. Dessutom, när datorsystemet är distribuerat eller heterogent, bör programmeraren också bestämma hur och när data ska kommuniceras.

Olika sätt har införts för att förenkla dessa överväganden så mycket som möjligt. I denna avhandling väljs uppgiftsbaserad parallellprogrammering för implementering av DuctTeip-ramverket med nya avancerade tekniker. DuctTeip ramverket är avsett för distribuerade arkitekturer och är efterträdaren till SuperGlue ramverket, som är utvecklade för delat minne. I dessa ramverk skrivs ett program i termer av uppgifter som arbetar på vissa data. I stället för att utföra uppgifterna omedelbart, skickas uppgifterna till ramverket för schemaläggning. Ramverket avgör vilka uppgifter som är redo att köras och schemalägger dem parallellt på tillgängliga processorer.

Liksom SuperGlue är DuctTeip ett datadrivet uppgiftsbaserat ramverk där beroendet mellan uppgifter och data driver exekveringen. Båda ramverken använder en innovativ versioneringsteknik för att effektivt representera uppgiftsrelaterade beroenden. I den här tekniken har varje data ett versionsnummer som visar hur många gånger det använts i olika läs- och skrivoperationer. Genom att övervaka versionsnumren av data kan ramverket hitta vilken uppsättning uppgifter som kan köras med aktuella versioner av data.

I DuctTeip ägs varje data av en specifik process, uppgifter som ändrar ett visst data körs av den processen. Om andra processer behöver datat skickas en sändningsförfrågan till ägaren. För att skapa effektiv kommunikation använder DuctTeip lyssnare för att representera denna typ av icke-lokala förfrågningar. Med hjälp av lyssnare kan DuctTeip skjuta upp kommunikationen till den tidpunkt då datat är redo och upptäcker också multipla förfrågningar och kan därigenom undvika onödig kommunikation.

Uppgifterna och datat i DuctTeip kan delas hierarkiskt i mindre bitar och användas i uppgifter på olika nivåer. DuctTeip använder detta för att köra program i en MPI/PThreads-hybridexekveringsmodell för olika nivåer av hierarkin. Olika datastorlekar på olika nivåer, gör det möjligt att anpassa storlekarna samtidigt med avseende på kommunikationsbandbredden och cachestorlekarna för bästa möjliga programprestanda.

Den abstrakta synen på uppgifter och data i dessa ramverk, gör det möjligt att skapa generiska abstrakta definitioner oberoende av vilket specifikt ramverk som avses. Ett gränssnitt för att arbeta med dessa generiska uppgifter och data föreslås, som gör den uppgiftsbaserade programmeringen enhetlig och frikopplar ramverksprogrammeringen från applikationsprogrammeringen. Alla ramverk som kan implementera det enhetliga gränssnittet, kan ta emot, schemalägga och köra generiska uppgifter som skickats in från en applikation eller från ett annat ramverk. För att samarbeta med varandra behöver ramverken inte känna till varandra så länge de följer gränssnittsprotokollen.

Alla dessa tekniker utvärderas genom att implementera olika tillämpningsproblem i DuctTeip och jämföra med implementationer i andra moderna uppgiftsbaserade ramverk. De experimentella resultaten visar att DuctTeip är konkurrenskraftigt när det gäller prestanda och skalbarhet.

Acknowledgments

Firstly, I thank my advisor, Elisabeth Larsson. Thank you for all your support and precious advice that prohibited me from choosing a thousand wrong ways that I could have taken in the research.

Secondly, I thank my wife Azadeh. Thank you, for all your support and your tolerance for all the tough times you have had due to my absence in family.

Thank you, Arman! I could not make it without the happiness you brought to our home.

Then I thank Martin Tillenius, for his contribution in the papers and for all I learned from him and his masterpiece the SuperGlue framework.

I would like to thank my dear friends, Halimeh Vahid, Maliheh Gholami Sheeri, Saleh Rezaeiravesh and Hamidreza Rezazadeh for their help and warm encouragements.

And thank you all TDB! Your respect for me will be remembered forever.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, *QR factorization on a multicore node enhanced with multiple GPU accelerators*, in Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International, IEEE, 2011, pp. 932–943.

[2] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, *Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model*, Research Report RR-8927, Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA, June 2016.

[3] E. Agullo, B. Bramas, O. Coulaud, M. Khannouz, and L. Stanisic, *Task-based fast multipole method for clusters of multicore processors*, Research Report RR-8970, Inria Bordeaux Sud-Ouest, Oct. 2016.

[4] E. Agullo, L. Giraud, A. Guermouche, S. Nakov, and J. Roman, *Task-based Conjugate Gradient: from multi-GPU towards heterogeneous architectures*, Research Report RR-8912, Inria, May 2016.

[5] A. M. Alakeel, *A guide to dynamic load balancing in distributed computer systems*, IJCSNS Int. J. Comput. Sci. Netw. Secur., 10 (2010), pp. 153–160.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: high-level and efficient streaming on multi-core*, Programming multi-core and many-core computing systems, parallel and distributed computing, (2014).

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures*, in Proceedings of the 15th International Euro-Par Conference, vol. 5704 of Lecture Notes in Computer Science, Delft, The Netherlands, Aug. 2009, Springer, pp. 863–874.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, *StarPU: a unified platform for task scheduling on heterogeneous multicore architectures*, Concurrency and Computation: Practice and Experience, 23 (2011), pp. 187–198.

[9] M. Balasubramaniam, K. Barker, I. Banicescu, N. Chrisochoides, J. P. Pabico, and R. L. Carino, *A novel dynamic load balancing library for cluster computing*, in Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004, pp. 346–353.

[10] P. Bauer, S. Engblom, and S. Widgren, *Fast event-based epidemiological simulations on national scales*, The International Journal of High Performance Computing Applications, 30 (2016), pp. 438–453.

[11] P. Bauer, S. Engblom, and S. Widgren, *Fast event-based epidemiological simulations on national scales*, The International Journal of High Performance Computing Applications, 30 (2016), pp. 438–453.
[12] L. Boillot, G. Bosilca, E. Agullo, and H. Calandra, *Task-based programming for seismic imaging: Preliminary results*, in 2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014, 2014, pp. 1259–1266.
[13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, *PaRSEC: Exploiting heterogeneity to enhance scalability*, Computing in Science & Engineering, 15 (2013), pp. 36–45.
[14] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, *Productive cluster programming with OmpSs*, in European Conference on Parallel Processing, Springer, 2011, pp. 555–566.
[15] B. L. Chamberlain, D. Callahan, and H. P. Zima, *Parallel Programmability and the Chapel Language*, The International Journal of High Performance Computing Applications, 21 (2007), pp. 291–312.
[16] Charm++, *Charm++: Parallel Programming with Migratable Objects*. `http://charm.cs.illinois.edu/research/charm`. Accessed: 2017-10-22.
[17] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, *PTG: an abstraction for unhindered parallelism*, in Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on, IEEE, 2014, pp. 21–30.
[18] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra, *PaRSEC in Practice: Optimizing a legacy Chemistry application through distributed task-based execution*, in 2015 IEEE International Conference on Cluster Computing, IEEE, 2015, pp. 304–313.
[19] D. del Rio Astorga, M. F. Dolz, L. M. Sanchez, J. G. Blas, and J. D. García, *A C++ generic parallel pattern interface for stream processing*, in Algorithms and Architectures for Parallel Processing, Springer, 2016, pp. 74–87.
[20] A. Ernstsson, L. Li, and C. Kessler, *SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems*, International Journal of Parallel Programming, (2017), pp. 1–19.
[21] M. Garcia, J. Corbalan, R. Badia, and J. Labarta, *A dynamic load balancing approach with smpsuperscalar and mpi*, in Facing the Multicore - Challenge II, R. Keller, D. Kramer, and J.-P. Weiss, eds., vol. 7174 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 10–23.
[22] M. Garcia-Gasulla, *Dynamic Load Balancing for Hybrid Applications*, ph. d thesis, Universitat Politècnica de Catalunya. Departament d'Arquitectura de Computadors, Barcelona, Spain, 2017.
[23] A. Goude and S. Engblom, *Adaptive fast multipole methods on the GPU*,

The Journal of Supercomputing, 63 (2013), pp. 897–918.
[24] *TensorFlow: An open source software library for Machine Intelligence*. `https://www.tensorflow.org/`. Accessed: 2017-10-22.
[25] H. KAISER, T. HELLER, B. ADELSTEIN-LELBACH, A. SERIO, AND D. FEY, *Hpx: A task based programming model in a global address space*, in Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, ACM, 2014, p. 6.
[26] Z. KHAN, R. SINGH, J. ALAM, AND R.KUMAR, *Performance analysis of dynamic load balancing techniques for parallel and distributed systems*, IJCNS Int. J. Comput. Netw. Secur., 2 (2010), pp. 123–127.
[27] X. LACOSTE, M. FAVERGE, P. RAMET, S. THIBAULT, AND G. BOSILCA, *Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes*, Research Report RR-8446, INRIA, Jan. 2014.
[28] G. MARTÍN, M.-C. MARINESCU, D. E. SINGH, AND J. CARRETERO, *FLEX-MPI: An MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems*, in Proceedings of Euro-Par 2013, F. Wolf, B. Mohr, and D. an Mey, eds., Springer, Berlin, Heidelberg, 2013, pp. 138–149.
[29] V. MARTÍNEZ, D. MICHÉA, F. DUPROS, O. AUMAGE, S. THIBAULT, H. AOCHI, AND P. O. NAVAUX, *Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system*, in Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on, IEEE, 2015, pp. 1–8.
[30] *StarPU Modularized Schedulers*. `http://starpu.gforge.inria.fr/doc/html/ModularizedScheduler.html`. Accessed: 2017-10-22.
[31] MPI, *MPI Forum*. `http://mpi-forum.org/`. Accessed: 2017-10-22.
[32] *The Microsoft Task Parallel Library*. `https://software.intel.com/en-us/node/608557`. Accessed: 2017-10-22.
[33] S. OHSHIMA, S. KATAGIRI, K. NAKAJIMA, S. THIBAULT, AND R. NAMYST, *Implementation of FEM Application on GPU with StarPU*, in SIAM CSE13-SIAM Conference on Computational Science and Engineering 2013, 2013.
[34] OPENMP, *OpenMP Specifications*. `http://www.openmp.org/specifications/`. Accessed: 2017-10-22.
[35] J. REINDERS, *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*, " O'Reilly Media, Inc.", 2007.
[36] E. H. RUBENSSON AND E. RUDBERG, *Chunks and Tasks: A programming model for parallelization of dynamic algorithms*, Parallel Computing, 40 (2014), pp. 328–343. 7th Workshop on Parallel Matrix Algorithms and Applications.
[37] ——, *Locality-aware parallel block-sparse matrix-matrix multiplication using the chunks and tasks programming model*, Parallel Computing, 57 (2016), pp. 87–106.
[38] M. SCHREIBER, C. RIESINGER, T. NECKEL, H.-J. BUNGARTZ, AND A. BREUER, *Invasive compute balancing for applications with shared and hybrid parallelization*, International Journal of Parallel Programming, 43

(2015), pp. 1004–1027.
[39] A. SHTERENLIKHT, *Fortran coarray library for 3D cellular automata microstructure simulation*, in Proc. 7th PGAS Conf, 2014, pp. 16–24.
[40] A. SPIEGEL, D. AN MEY, AND C. H. BISCHOF, *Hybrid parallelization of CFD applications with dynamic thread balancing*, in Applied Parallel Computing. State of the Art in Scientific Computing. PARA 2004, J. Dongarra, K. Madsen, and J. Waśniewski, eds., vol. 2732 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2006, pp. 433–441.
[41] M. TILLENIUS, *SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization*, SIAM Journal on Scientific Computing, 37 (2015), pp. C617–C642.
[42] M. TILLENIUS AND E. LARSSON, *An efficient task-based approach for solving the n-body problem on multicore architectures*, in PARA 2010: State of the Art in Scientific and Parallel Computing, Reykjavik, Iceland, 2010, University of Iceland, pp. 74–1.
[43] M. TILLENIUS, E. LARSSON, R. M. BADIA, AND X. MARTORELL, *Resource-aware task scheduling*, ACM Transactions on Embedded Computing Systems (TECS), 14 (2015), p. 5.
[44] M. TILLENIUS, E. LARSSON, E. LEHTO, AND N. FLYER, *A scalable RBF–FD method for atmospheric flow*, Journal of Computational Physics, 298 (2015), pp. 406–422.
[45] UPC, *Berkley Unified Parallel C*. `http://upc.lbl.gov/`. Accessed: 2017-10-22.
[46] W. WU, A. BOUTEILLER, G. BOSILCA, M. FAVERGE, AND J. DONGARRA, *Hierarchical DAG scheduling for hybrid distributed systems*, in Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE, 2015, pp. 156–165.
[47] X10, *The X10 Parallel Programming Language*. `http://x10-lang.org/`. Accessed: 2017-10-22.
[48] A. ZAFARI, E. LARSSON, M. RIGHERO, M. A. FRANCAVILLA, G. GIORDANENGO, F. VIPIANA, AND G. VECCHI, *Task parallel implementation of a solver for electromagnetic scattering problems*, Tech. Report 2016-015, Uppsala University, Division of Scientific Computing, 2016.
[49] A. ZAFARI, E. LARSSON, AND M. TILLENIUS, *DuctTeip : An efficient programming model for distributed task based parallel computing*, Tech. Report diva2:1173778, Uppsala University, Division of Scientific Computing, 2018 (submitted).

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1621

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-338838

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018