UNIVERSITÄT PASSAU
Fakultät für Informatik

Extending a Task Farming Framework with Dependences to P2P Communications

Diplomarbeit

Autor:
Philipp Claßen

Betreuer:
Priv. Doz. Dr. Martin Griebl
Lehrstuhl für Programmierung
Universität Passau

Passau, February 5, 2009

Abstract

Grid computing is becoming more and more popular, but contrary to earlier expectations the development of applications for Grid environments is still not possible without expert knowledge in the field of parallelization. Therefore, recent work [14, 4] has combined automatic loop parallelization (in the polytope model) with component-based Grid programming: First, automatic parallelization techniques are used to convert the sequential input program into a task farming application (allowing dependences between tasks). Second, this parallelized application is used as application-specific code for a "master-worker with dependences" Higher-Order Component (HOC), which can be executed together with the required Grid middleware.

In this thesis, the master-worker HOC is replaced by a new P2P-HOC that adds peer-to-peer communication between workers, thus increasing the scalability and performance of the target application. The focus of this work will be less on code generation, but more on the extension of the Grid runtime environment, especially on task scheduling and its consequences on the communication between workers.

Acknowledgment

First of all, I would like to thank Martin Griebl for helpful discussions, and also for his comments on an earlier draft of this work.

Next, I would like to thank all member of the LooPo team, especially Armin Größlinger. In various discussions, he contributed valuable ideas, and also helped to lay the theoretical foundations of this work.

My final thanks goes to my whole family, especially to my brother Micki.

Contents

1 Introduction

Writing efficient programs for distributed memory environments is challenging, especially for large-scale heterogeneous clusters. In the last years, much research has been focused on the organization of these large systems, most notably on peer-to-peer and Grid computing [15, 16]. The goal is to provide a high-level programming model, which helps the programmer to concentrate on the application logic rather than on the various low-level aspects like efficient communication protocols, platform independence or security services.

This approach overlaps with the field of component-based software engineering, where the application is created by combining reusable components. Each component encapsulates a certain functionality. The idea goes back to the sixties [35], but it has become increasingly popular since the nineties [41]. By now, there exist several component frameworks for Grid computing, for example, the Globus Toolkit [18].

However, the programmer is still responsible to expose the parallelism of the application. Since the available machines will generally differ in computing power, load balancing must also be considered. Therefore, the ultimate goal was to relieve the programmer from this burden, and let the compiler find the parallelism automatically. Since that is extremely difficult for general programs, it is advisable to concentrate on the parts of the program where it spends most of its execution time. For many computationally intensive programs, these are loop nests. This is also the reason why most of the research on automatic parallelization addresses loop nest optimization. One sophisticated technique is known as the polytope model [32, 22], which is used by the LooPo loop parallelizer [34].

It has been recently demonstrated that LooPo can also be used to generate code for Grid environments [4]. The generated parallel application uses the task farming paradigm but also allows dependences between tasks. Thus, the execution of a task can be delayed until all tasks it depends on have finished.

The transformation of the user application consists of two steps: First, automatic parallelization techniques are used to convert the sequential input program into a parallel program for distributed memory [10]. Second, this parallel program is used as application-specific code for a so-called ”master-worker with dependences” Higher-Order Component [10, 13, 14], which can be executed together with the required Grid middleware.

One advantage of task farming is that it addresses load balancing, which is especially important in Grid environments. However, the client-server architecture with one dedicated master (server) that assigns tasks to its workers (clients), can have a negative impact on scalability. Whenever a worker asks for a task, the master has to write the data that is required into a buffer, and send it to the worker. After the task is finished, the master receives all computed values, and updates its local data.

The goal of this diploma thesis is to eliminate this overhead: To achieve this, the data will no longer be stored exclusively on the master but can also be distributed on the workers. So, after a task has terminated, it is not necessary to immediately send its results back to the master. Consequently, it becomes necessary to extend the protocol to support peer-to-peer communication between workers, because the data that is required to execute a task will be generally located on multiple workers. For this purpose, the "master-worker with dependences" Higher-Order Component (master-worker HOC) will be replaced by a new Peer-to-Peer Higher-Order Component (P2P-HOC). Note that the additional complexity behind these enhancements is completely hidden to the end user.

The organization of this work is as follows: The P2P-HOC and the suggested system architecture are described in section 2. As a consequence of the P2P extensions, task scheduling[1] and task prefetching[2] will become more important than in the master-worker HOC, therefore, section 3 is dedicated to this subject. Section 4 is about of the automatic generation of the application-specific code of the P2P-HOC. Finally, section 5 presents experimental results and section 6 contains conclusions and ideas for future work.

1.1 Motivation: One typical example

Let us suppose the user wants to execute the following program, which we will refer to as SOR1d (one-dimensional successive over-relaxation), on a Grid environment:

```
for (int k = 1; k <= m; k++) {
  for (int i = 2; i <= n-1; i++) {
    A[i] = (A[i-1] + A[i+1]) / 2.0;
  }
}
```

It is a typical example of a numerical calculation on arrays using nested for loops. Note that it is not obvious, how this program can be rewritten to take advantage of multiple computers (e.g., it is not possible to simply execute one of the two loops in parallel, because there are data dependences between each iteration). Instead of that, the creation and execution of the automatically parallelized program consists of two steps:

1. LooPo parallelizes the input program to generate target code, which is used to parameterize the master-worker Higher-Order Component (master-worker HOC).

2. The master-worker HOC can be executed with a special component-based framework acting as the Grid middleware.

In summary, we obtain a task farming application with dependences between tasks, as illustrated in the task dependence graph shown in Figure 1. Each task has an unique task ID, but the actual values are not relevant for the framework[3]. In fact, the tasks could also be renamed as ①, . . . , ⓝ.

It should be noted that this graph only contains 24 nodes, but real-world task dependence graphs are significant larger. To create the small "toy" graph, we reduced the problem size (i.e., m and n) and also limited the parallelism. This was necessary for

[1] The role of the task scheduler is to decide how available tasks are assigned to requesting clients.

[2] Task prefetching is a technique to improve the resource utilization of the clients (computation power and network bandwidth) by requesting multiple tasks at the same time.

[3] The code generation internally uses tiling vectors as task IDs, because, as we will later see, tasks are created by a technique called tiling. In this example, two-dimensional tiling is used.

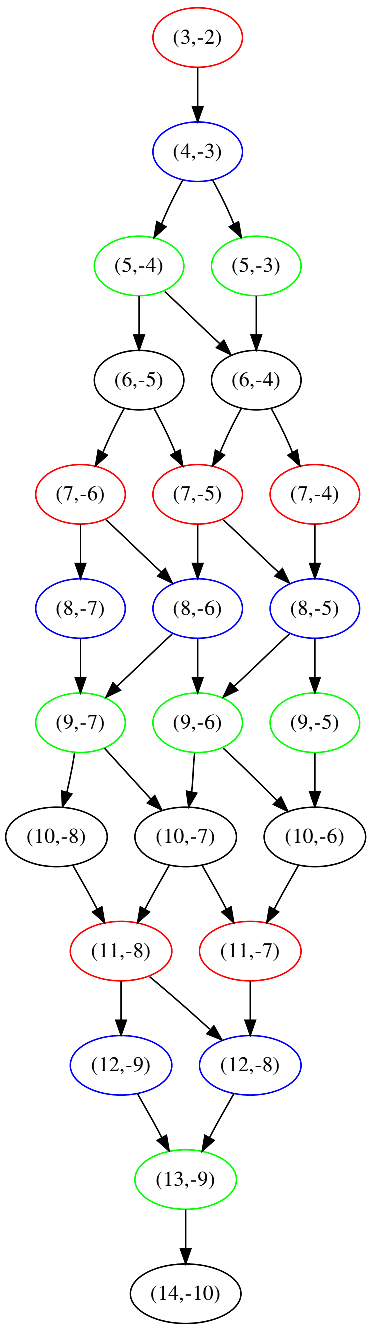


Figure 1: Task dependence graph for the SOR1d example.

practical reasons, because realistic graphs will consist of thousands of tasks (e.g., the graph that performed best in our experiments consisted of about 10,000 tasks).

However, all task dependence graphs for the SOR1d example share the same characteristics: It begins with a phase of increasing parallelism and ends with decreasing parallelism. Therefore, our example graph is reasonable, apart from the fact that it is too small.

The execution of the parallel application is controlled by the master's task scheduler, which assigns the tasks to workers (without violating any dependences from the task dependence graph). The master sends the task assignments along with the task's input data. After the worker has finished the task's execution, it sends the modified data back to the master.

One drawback is that the data flow always has to go through the master, including all intermediate results, because all data is exclusively stored on the master. Therefore, the master can become the bottleneck of the computation, which will harm the scalability. In this thesis, we propose to optimize the data flow by replacing the master-worker architecture with a centralized P2P network, where the data can also be distributed among the workers. To take advantage of the P2P architecture, the communication scheme needs to be modified:

- Intermediate results will be propagated by P2P communications between workers.
- All communication from and to the master will be moved to the begin and end of the overall computation.

Thus, the indirection through the master is eliminated. Instead, the data flow of intermediate results takes place along the edges of the task dependence graph.

Our experiments prove that the scalability improves as a result of the P2P approach. In the case of the SOR1d example, the master-worker version scaled to 16 computers, but the P2P version scaled to 48 computers. Additionally, the overhead of the parallel computation is reduced, as there is no need for the repeated communication of the data between master and workers.

2 Extending the framework

2.1 Terms and definitions

This section contains definition of terms that are important in the context of this work. First, we define two general terms, namely *component* and *framework*. The following definitions are taken from *The Common Component Architecture Forum* [43].

Definition 1 (Component)**.** A component is a software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities. A component has a clearly defined interface and conforms to a prescribed behavior common to all components within an architecture. Multiple components may be composed to build other components.

Definition 2 (Framework)**.** A framework is a specific implementation of a component architecture.

The framework provides the glue that binds components together. It is used to combine separate components from a component pool into a running application. It allows components to be linked together and to make calls on specific component interfaces. Additionally, the framework can provide information about the run-time environment.

The next term *Higher-Order Component* has been developed at the university of Münster [27]:

Definition 3 (Higher-Order Component (HOC))**.** Higher-Order Components (HOCs) are reusable patterns of parallelism, available to Grid application programmers as a collection of Grid Services. HOCs are used as generic components, parameterized with application-specific units of code that exploit a novel mechanism for handling code mobility in grid environments. The parameters of a HOC are application-specific codes provided by users[4] who are specialists in a specific scientific domain, for example, biochemists, physicists or seismologists.

Throughout this work, we will refer to the application that is being parallelized by using the following terms:

Definition 4 (User application, input program, and input file)**.** The application that the user wants to parallelize is called *user application.* It is the execution of the *input program*, which is a synonym for the user application's source code. If the input program is annotated with LooPo specific tags (e.g., `// loopo begin loop (...)`), it is also called *input file.*

Finally, we would like to address the somewhat controversial term *peer-to-peer* (or *P2P*). Alas, there is no simple definition, which is generally accepted, but there are several (often contradictory) opinions on this topic.

One central issue is whether the peer-to-peer structure has to be used for all purposes (sometimes referred to as *true peer-to-peer networks*) or if it is allowed to use a client-server structure for specific tasks (e.g., searching). Note that the former condition excludes famous "P2P" file sharing applications like Napster, or earlier versions of eDonkey2000.

In this work, we will use the latter more relaxed condition. This is also according to a definition from Foster and Iamnitchi [16]. However, as our proposed architecture is centralized, it is not a "true peer-to-peer" architecture.

2.2 Related work

The framework that is extended in this thesis is based on the work *Loop parallelization for a GRID master-worker framework* [4], which combines model-based parallelization and task farming.

[4] In the context of this diploma thesis, the application-specific code is generated automatically from a sequential input program provided by the user.

2.2.1 Model-based parallelization

The idea of automatic loop parallelization is to reorder the loop's iterations to increase the number of independent iterations that can be computed in parallel, while still preserving all data dependences. But how can this transformation be found?

In model-based parallelization, a mathematical model is used to describe the loop. For example, the polytope model [32, 22, 10], which is used by the model-based parallelization framework LooPo [34], uses a polytope representation: Each loop translates to one dimension of the polytope, and each integral point inside the polytope represents an unique loop iteration. This *source polytope* is then transformed to a *target polytope*, which is finally used to generate the target code.

The transformation is called *space-time mapping*. It consists of the *schedule* and the *placement* (or *allocation*). The schedule maps each iteration to a certain time, and the placement determines by which computer it should be computed. In other words, the schedule decides *when*, and the placement *where* the computation takes place.

The advantage of model-based parallelization is that, because of the underlying mathematical theory, it is possible to prove the correctness of the transformation. It also allows to quantify the optimality of the transformation regarding certain criteria (e.g., latency).

While the original polytope model is very restrictive, many parts has been extended and generalized over the years. We will refer to the literature for an overview of these extensions [22].

2.2.2 Task farming

Task farming is a general parallelization technique, also called Master-Worker paradigm. The idea is to break a problem into subproblems called *tasks*, which can be executed by multiple independent *workers*. These workers are coordinated by a dedicated *master*. The *task scheduler*, which is part of the master, is responsible for assigning tasks to workers.

Task farming is a natural fit for distributed heterogeneous clusters and Grid environments, especially for "embarrassingly parallel problems" like image rendering, but it can also be applied to a wider range of applications [1, 20].

A common extension is the generalized Master-Worker paradigm [26]: In contrast to the simple Master-Worker paradigm, it allows not only one single set of independent tasks, but several batches of tasks. These batches are executed sequentially, but all tasks that are part of the batch can be executed in parallel:

```
while (!finished()) {
  executeNextBatchOfTasks(); // distributed among workers
}
```

Tasks from newer batches can use results from the older batches, but not from the same batch. This generalized model is well suited for many application, for example, n-body simulations [21], genetic algorithms [7], and Monte Carlo simulations [5].

A similar approach is also becoming increasingly popular for shared memory architectures: Let the user expose the parallelism (in form of tasks), but leave the decision how the work is scheduled on the processor's cores to the runtime system. This abstracts from the low-level thread synchronization and from the number of threads being used, while still preserving sequential semantics, which is useful for debugging and testing.

It is often combined with work-stealing schedulers, like in the C language extension Cilk [6], allowing idle processors to "steal" tasks from busy processors. This further improves load balancing and scalability compared to non-stealing scheduling policies.

Most implementations allow dependences between tasks in the following form:

- Each task can have multiple children tasks.
- A parent task can only be executed if all children tasks are finished.
- Each task can have at most one parent. This restricts the dependences between tasks to tree-like graphs.

Although the idea is not new, for example, Cilk has been developed since the early nineties, but it is only recently that multithreaded parallel programming has become mainstream. Meanwhile, the industry has begun to use libraries (or language extensions) for task parallelism to simplify the development of efficient applications for multi-core processors, or to extend legacy code:

- OpenMP directly supports task parallelism since version 3.0, which added the constructs `#pragma omp task` and `#pragma omp taskwait` [37, 38]. However, it lacks support for speculative tasks, which for instance makes it difficult to parallelize nondeterministic search algorithms[5].
- Intel Threading Building Blocks [42] (also known as TBB) is a C++ template library, mainly providing parallel algorithms (e.g., `parallel_for`, `parallel_while`, `parallel_sort`) and data structures (e.g., `concurrent_queue`). The library internally uses tasks to implement the parallel algorithms, but the task interface is also available to the user.

  Recently, TBB has been released as open source.
- Cilk++ is a commercial enhancement of MIT Cilk, which among others will add support for C++ [8].

2.2.3 Master-worker framework

As mentioned earlier, the master-worker framework combined model-based parallelization and task farming [4]. The goal is to simplify the execution of sequential input program on Grid environments. It also provides load balancing, thus solving one of the major problems with statically scheduled model-based parallelization [10].

[5] Cilk's `inlet` and `abort` keywords allow the user to cancel speculative tasks.

First, the input program is parallelized by LooPo. To abstract from the Grid environment, LooPo does not create a stand-alone executable, but describes the parallelized application using the master-worker HOC, which is a specially designed Higher-Order Component for task farming with dependences. The master-worker HOC is then executed on the Grid using the master-worker framework, which acts as the Grid middleware.

To partition the parallel application into tasks, LooPo uses tiling [45, 31], a technique that has been originally used for cache optimizations. It can be used to automatically create blocked algorithms (e.g., for matrix multiplication), thus improving cache efficiency by breaking the problem down into subproblems, which fit into cache.

In parallel computing, tiling is often used to reduce communication costs by combining multiple computations into so called *tiles*. Thus, communication takes place between tiles (instead of computations), which has two advantages:

1. It reduces the number of messages that are sent, because there are an order of magnitude less tiles than computations.

2. Communication within a tile is not necessary, because all the tile's computations will be executed by the same processor.

Tiling can be used in exactly the same way to generate tiles that represent tasks. In fact, in the context of the automatic HOC generation, the terms task and tile can be used interchangeably. As the tasks have to be executed with respect to the dependences in the input program, the master-worker HOC includes a description of all dependences between these tasks. This acyclic directed graph (DAG) is called *task dependence graph*, and it is required by the task scheduler to assign tasks to workers, without violating any data dependences.

Because of the component-based design, the master-worker HOC is reusable in other component frameworks for Grid computing, which include important Grid related features that are missing in the original prototype framework (e.g., security layers, authentication). Recent work has integrated the master-worker HOC in the Globus Toolkit, under the name ”LooPo HOC” [44]. It is available for download as a part of the HOC-SA Globus incubator project [19].

2.3 Basic architecture and protocol

2.3.1 Initial situation: Master-worker HOC

The master-worker HOC provides a classical task farming architecture, where one dedicated master controls multiple independent workers. There is no communication between workers, but only between master and worker.

The role of the framework is to run the master-worker HOC, which has been generated by LooPo from the input file. The framework is a Java application that is started on each computer, but only one will act as the master. This master process will produce the same output as the sequential execution of the input program. However, depending on the success of the parallelization, the master should produce the results faster.

The overall execution can be divided into three phases:

Startup phase All workers register to the master, which assigns each of them a fixed worker ID.

Additionally, the master executes the input program until it reaches the parallelized loop (*pre-loop code*). At this point, all runtime parameters are known, which are needed by the master to create the task dependence graph.

Computation phase This is the most interesting phase from the point of view of distributed computing, because finally all computers work together to compute the selected loop, which is "divided" into single tasks.

In principle, each worker performs the following loop:

1. Request the next task from the master.
   (If the master signals that all tasks are finished, leave the loop.)
2. Execute the task.
3. Send the results back to the master.
4. Goto step 1.

(The efficiency can be improved, if the workers will request multiple tasks simultaneously. This important optimization is discussed later in section 3.1.1.)

Meanwhile, the master has two functions:

- Respond to task requests from workers by assigning tasks to them. If there are no tasks available, block until new tasks become available.
- Join tasks that have been finished by workers, and update the task dependence graph because new tasks may have become available.

(Task assigning and joining have to done in an asynchronous way. Otherwise, there can be deadlocks, because if the assignment blocks, it can only be unblocked if old tasks are joined.)

Finalization phase After all tasks have terminated, the master executes the remaining part of the input program, which follows the parallelized loop (*post-loop code*).

Finally, all connections to the workers are closed, and the execution of the framework is terminated on each computers.

Let us illustrate the computation phase with the previous SOR1d example (see section 1.1). We will also adopt its simplified task dependence graph (Figure 1) for practical reasons, because realistic graphs will consist of thousands of tasks.

Note that the intention of this example is to explain the communication scheme between master and workers. Because of the example's simplicity, there is only limited parallelism.

To avoid unnecessary repetition, we will only describe the communication for the first five tasks. The relevant part of the task dependence graph is shown in Figure 2.

Example 1. Figure 3 shows the communication at the begin of the SOR1d example. There are three computers in this example: One master and two workers **A** and **B**.

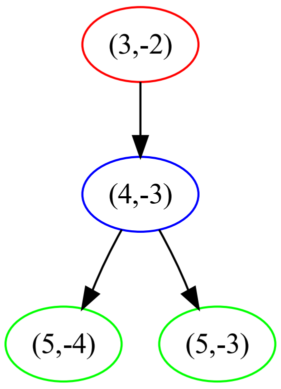


Figure 2: Relevant part of the task dependence graph for the SOR1d example. The complete dependence graph is shown in Figure 1.

The diagram also contains an actor for the scheduler, which is part of the master and is responsible for assigning tasks to workers.

At the begin, both workers are idle and request tasks from the master, which passes the requests to the scheduler. Considering the task dependence graph (Figure 1), only task $(3, -2)$ is available. So, the scheduler decides to assign $(3, -2)$ to worker **A**, and temporarily blocks the request of worker **B**. To complete the task assignment, all the data that is required to execute the task must be provided to the worker, too. In case of the master-worker HOC, the master attaches all data that will be read when the task is executed. Finally, the task assignment, consisting of the task ID and the input data, is sent to worker **A**.

When worker **A** receives the assignment, it starts to execute the task, which involves three steps:

1. Unpack the input data (i.e., all data that will be read)
2. Perform the computation
3. Pack the output data (i.e., all data that has been written)

After the task has been executed, the worker sends the results back to the master, and simultaneously requests the next task. Note that after the results have been sent, the data are no longer used on the worker. In order words, the worker ”forgets” the results of the computation.

When the message has arrived, the master joins the tasks, that means it unpacks its results and informs the scheduler to update the task dependence graph. Note that at this point, the application relevant memory on the master is exactly the same as if task $(3, -2)$ had been executed on the master (instead of client **A**).

Since task $(3, -2)$ has been finished, task $(4, -3)$ becomes available. This awakens the task assignment thread of the scheduler, which decides to assign the new task to worker **B**. As there is no other available task left, the second task request of worker **A** is blocked.

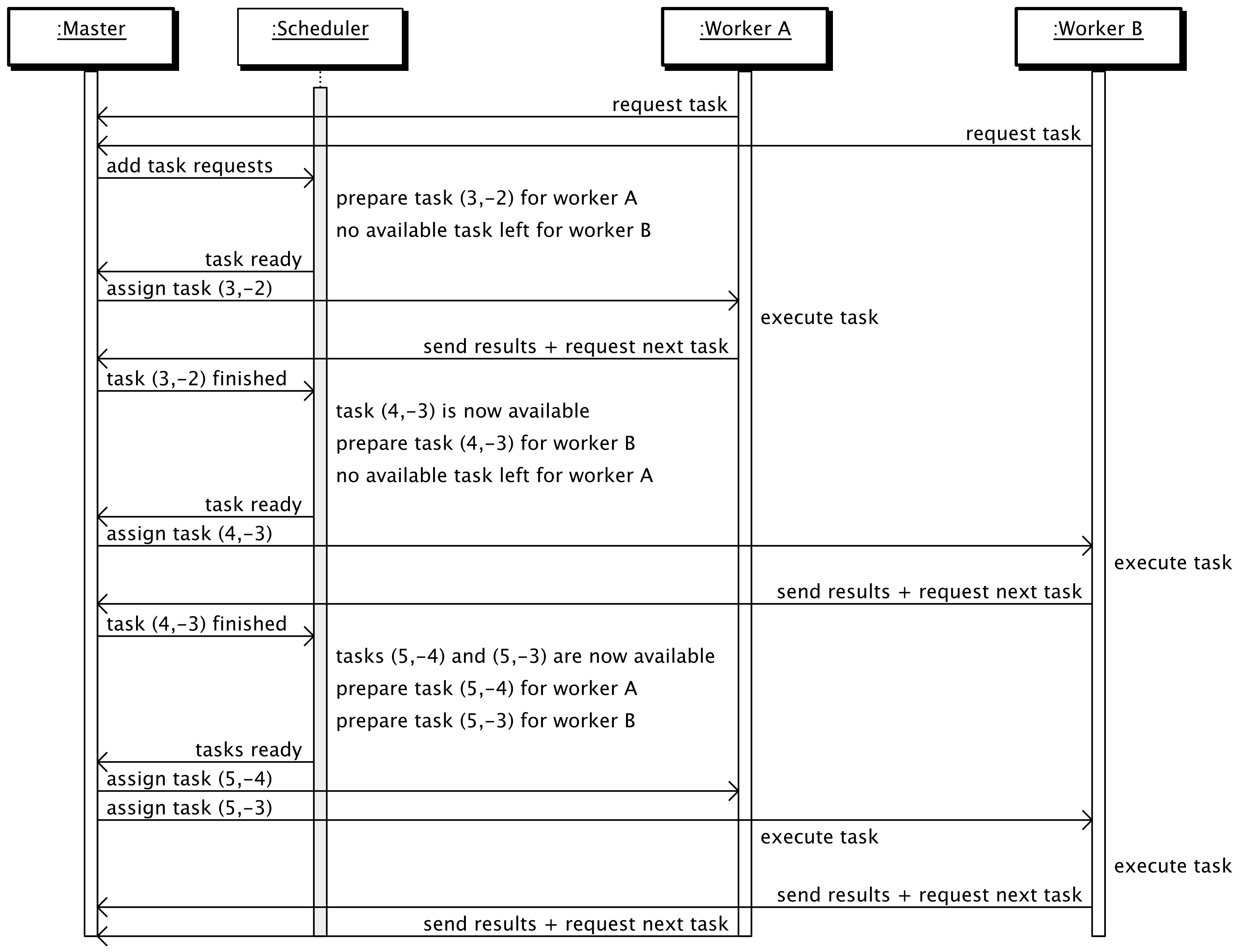

:Master
:Scheduler
:Worker A
:Worker B
request task
request task
add task requests
prepare task (3,−2) for worker A
no available task left for worker B
task ready
assign task (3,−2)
execute task
send results + request next task
task (3,−2) finished
task (4,−3) is now available
prepare task (4,−3) for worker B
no available task left for worker A
task ready
assign task (4,−3)
execute task
send results + request next task
task (4,−3) finished
tasks (5,−4) and (5,−3) are now available
prepare task (5,−4) for worker A
prepare task (5,−3) for worker B
tasks ready
assign task (5,−4)
assign task (5,−3)
execute task
execute task
send results + request next task
send results + request next task

Figure 3: Sequence diagram showing the communication scheme for the original master-worker HOC. The task dependence graph for this example (SOR1d) is shown in Figure 1.

The assignment and execution of task $(4,-3)$ on worker **B** is analog to the execution of task $(3,-2)$, but after task $(4,-3)$ is finished, there are two available task: $(5,-4)$ and $(5,-3)$. Thus, it is possible to assign tasks to both workers, which will execute them independently, and send the results back to the master.

Let us summarize some observations concerning this example:

- The task assignment is blocked if there is no task available.
- There is no communication between workers.
- Every task execution forces communication between master and worker, even if both tasks had been executed by the same workers (e.g., if the scheduler had assigned both tasks $(3,-2)$ and $(4,-3)$ to worker **A**).
- After an executed task has been joined by the master, the application relevant memory on the master is exactly the same as if the task had been executed on the master. At the same time, the task's results become invalid on the worker.

2.3.2 How to extend the task farming approach to P2P communication

In our proposed P2P-HOC, the clients, which will replace the workers, should play a major role, while the load of the task controller, which will replace the master, should be reduced as much as possible.

In the master-worker HOC, the possibilities of cooperation between the workers is limited, because all data is exclusively stored on the master. Thus, the basic idea of the P2P-HOC is to distribute the arrays among the controller and clients in the course of the computation: Only at the begin and at the end of the computation, all data is located on the controller, but during the computation, the intermediate results are distributed among the clients.

After a client has executed a task, it does not send the results back to the controller (as it would be the case in the master-worker HOC), but it only informs the controller that the task is finished, and requests a new task. Whenever the controller assigns a task to a client **A**, which requires data from tasks ①, …, ⓝ that have been finished on clients $\mathbf{B_1}, \ldots, \mathbf{B_n}$, the controller sends each client $\mathbf{B_1}, \ldots, \mathbf{B_n}$ a message, so they will send the required data to client **A**.

As already mentioned, at the begin of the computation all data is located on the controller. Therefore, it must also be possible to communicate data from the master to the clients as far as it is necessary to execute a specific task. This is called *soaking.* Additionally, communication from clients to the master have to be supported as well, because after the computation all data has to be available on the controller. This is called *draining.*

Figure 4 illustrates the three types of data packets that are communicated during the computation:

Soak data packet Sent from the controller to a client (mostly at the begin of the computation).

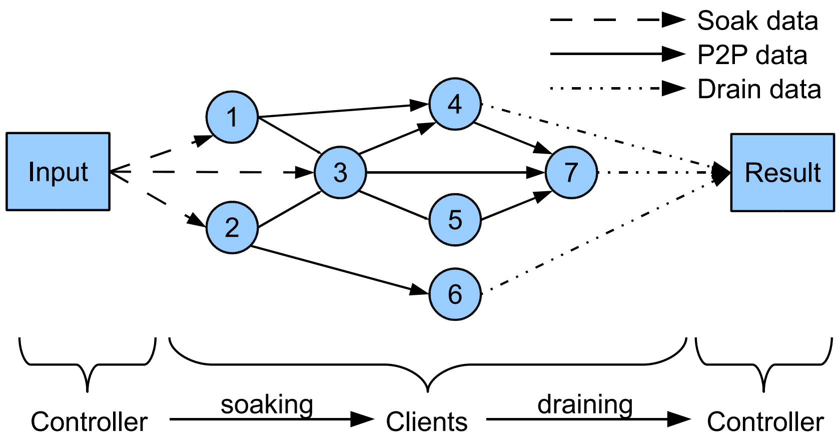


Figure 4: Overview of the different types of data packets that are communicated during the computation.

P2P data packet The default data packet that is sent from one client to another client.

Drain data packet Sent from a client to the controller (mostly at the end of the computation).

Each data packet belongs to a specific task. The input data of a task is provided by soak and P2P data packets and the data that is stored on the client. The output data of a task is stored on the client that executed it. The output can be further divided into intermediate results that are needed to execute other tasks, and final results that are sent to the controller in the form of drain data packets.

To reduce the communication overhead, soak data packets can be attached to the task assignment message that is sent from the controller to the client. Additionally, drain data packets can be attached to the ”task-finished” message that is sent from the client to the controller.

The differences between master-worker and P2P-HOC (regarding data distribution), also affect the efficiency of task scheduling strategies. The task scheduler is part of the master (or controller), and it reacts on task requests by assigning tasks to workers (or clients). To decide which tasks are available, the scheduler uses the information from the task dependence graph. Whenever a task is finished, the task dependence graph must be updated as well, because new tasks may become available.

In the master-worker HOC, one simple and efficient task scheduling strategy is to select one available task arbitrarily, and assign it to the client that has waited for the longest time. If no task is available, the scheduler blocks until new tasks become available. Note that because of the service policy (first-come, first-served), load balancing is guaranteed.

However, once peer-to-peer communication between clients is allowed, the role of the task scheduler becomes crucial. But why is task scheduling more important for the P2P-HOC, although it has little impact in the master-worker HOC?

One reason is that an enhanced task scheduler can avoid communication by keeping P2P communication local (i.e., the same client is both sender and receiver). Note that in

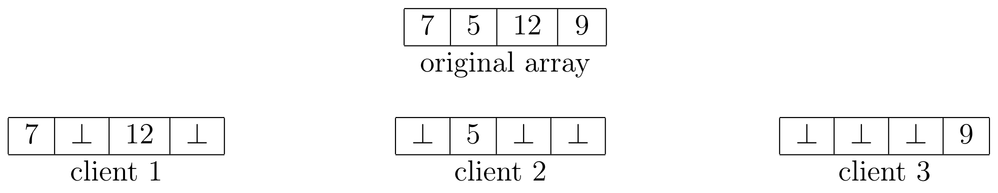


Figure 5: Distributed arrays without memory reduction: Each client has allocated the complete array, even if most cells are undefined.

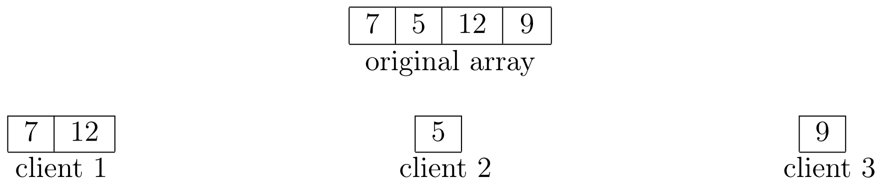


Figure 6: Distributed arrays with memory reduction: Each client has only allocated memory for the array cells, which are located on this client.

the master-worker HOC, there cannot be local communications, because there are only two types of communications: Master to worker and worker to master. In both cases, the communication is non-local.

Another reason concerns task prefetching, which is discussed in section 3.1. This is an optimization technique, which is both applicable to the master-worker HOC and the P2P-HOC. It is similar to pipelining, and works by overlapping computation with communication to utilize the maximum computation power and network capacity.

2.3.3 Outlook: Client side memory reduction

Our current implementation is not able to reduce the memory footprint of the user application, even though the arrays are distributed among the clients. That is because the complete arrays have to be allocated on each computer. However, at any time, each modified array cell is valid on exactly one computer and is undefined on all other computers, as shown in Figure 5.

An improvement would be to support real distributed arrays, as illustrated in Figure 6. This would make it possible to efficiently execute user applications, which require more memory than is available on the target machines (e.g., out-of-core problems).

The problem is that it cannot be predicted, on which client one given array cell will be located for a certain time, as it changes dynamically in the course of the computation, depending on how the tasks are assigned to the clients.

A possible solution would be to replace the arrays on the clients by more sophisticated data structure for sparse arrays (e.g., hash tables, Judy arrays).

Due to the nature of the sequential input program, it cannot be avoided that, at the begin and at the end of the computation, all data has to be stored on the controller. This requires efficient data storage techniques, otherwise the controller will become the bottleneck of the computation.

A straightforward solution would be to replace the arrays on the controller by memory-mapped files (or indirectly by databases), possibly in combination with tiling to increase the locality of the disk accesses [45, 31].

Future work might examine these ideas to answer the open questions:

Is it efficient? If the overhead caused by the more complex data structure is significant, the parallelized program may become slower than the original sequential program. However, if the original program will run out of memory (or at least requires paging), even a high overhead may be acceptable.

When to shrink? In the course of the computation, the intermediate results stored on the clients may become outdated. In our current implementation, outdated data is no longer accessed, but it is still stored on the client and may eventually be overwritten by newer results. To save memory, however, these data should be freed as soon as possible.

Will disk accesses become the bottleneck? As long as there is only P2P communication between the clients, no disk accesses on the controller are necessary. If there is much soaking and draining, however, the controller's disk accesses are likely to become the bottleneck, especially if the locality is low.

Remark. There is ongoing research outside of this thesis to use Ehrhart polynomials [12, 36, 33] to implement an efficient sparse array representation. The details of this promising idea would go beyond the scope of this thesis, but it is expected that the results will be superior to dictionary-based sparse arrays in terms of memory usage and access speed.

However, in order to use Ehrhart polynomials, the architecture of the clients has to be modified. In contrast to the current approach, where the data is persistently stored on the clients, the new approach will use a data flow architecture.

Instead of executing a task and storing the results persistently on the client, the computed data will be exclusively propagated in form of P2P data packets. As a consequence, there is no need to shrink arrays during the computation, because all the data on the clients is stored within the P2P data packets, which can be immediately freed after sending.

2.4 Soaking and draining

Soaking is the transfer of input data from the controller to a client, which the client requires to execute a task. These data can be further distinguished into *runtime parameters* and *input arrays*. Currently, LooPo cannot detect them without help, but requires that the user surrounds the loop that should be parallelized with a special annotation (for details see section 4.1).

The annotation contains the names of all parameters and arrays that are used inside the loop. Additionally, the array sizes have to be specified. Optionally, arrays can be declared readonly or writeonly, which can be used for optimizations.

```
// loopo begin loop (constants: m, n; arrays: A{n+1})
for (int k = 1; k <= m; k++) {
  for (int i = 2; i <= n-1; i++) {
    A[i] = (A[i-1] + A[i+1]) / 2.0;
  }
}
// loopo end loop
```

Figure 7: Annotated loop for the SOR1d example.

Example 2. In the SOR1d example, there are two parameters m and n, and one array A of size $n + 1$. The annotated loop is shown in Figure 7.

The array A is by default declared read and writeable. Thus, potential soaking data may include two runtime parameters m and n, and one input array A (or at least parts of it).

Note that from performance perspective, only the soaking of the input array is important. In fact, in the master-worker HOC, the runtime parameters are part of the task description and are transmitted for each task. But as most examples contain only few runtime parameters, the overhead is insignificant. Nevertheless, the P2P-HOC communicates runtime parameters only once at the beginning. This is not done for performance reasons, but it is a design decision, which shows a fundamental difference between the two HOCs:

The master-worker HOC is stateless, that means after a task is finished, no information is stored, because with each new task assignment, the master provides all information that is required to execute the task. This is possible, because data is exclusively stored on the master.

In contrast, the idea of the P2P-HOC is to store as much as possible on the client side. In order to allocate storage for these data, the client uses the information from the runtime parameters to determine the size of the arrays that are used in the computation. Therefore, before the actual computation is started, there is an additional initialization step, in which the controller creates a half-initialized `UserClient` object containing only the runtime parameters. This object is broadcasted to all clients, which will finish the initialization.

The following two sections will discuss the soaking and draining of the input arrays. Draining is the opposite of soaking. It is the transfer of output data from the client back to the controller. As stated earlier, the soaking of the runtime parameters is more or less an implementation detail. Draining of the runtime parameters is also not necessary, because runtime parameters cannot be modified by the loop. So from now on, we will refer to the data stored in the arrays, when talking about soaking and draining.

2.4.1 Distributing the soak data

Whenever a task that requires soaking is assigned to a client **A**, the controller will also send a soak data packet along with the task assignment message. Then the communication of the soak data is as follows:

1. The controller calls the P2P-HOC method `writeSoakData`, which generates a soak data packet for the task.
2. The soak data is serialized and sent to client **A**.
3. Client **A** unpacks the soak data by calling `loadSoakData`.

It is important to note that not every task requires soaking. In fact, the benefits of the P2P approach can only show to advantage if the bulk of the communication comes from P2P data packets. Therefore, the framework will skip soak communication if `writeSoakData` signalizes that the task does not need soaking (by returning `null` instead of a soak data packet).

Remark. Theoretically, the P2P-HOC could be (ab-)used in a way that is equivalent to the master-worker-HOC by using solely soaking and draining (but without P2P communication). In other words, the P2P-HOC is backward compatible.

Although this is of no practical importance, it allows one theoretical conclusion: If the parameters for the P2P-HOC — in our case, the automatically generated code — are optimal, the P2P-HOC should not be slower than the master-worker HOC (assuming that the extended framework causes no significant overhead).

2.4.2 Collecting the drain data

From the P2P-HOC's point of view, the transfer of drain data is almost identical to the transfer of soaking data:

1. The client calls the P2P-HOC method `writeDrainData`, which generates a drain data packet for the task.
2. The drain data is serialized and sent to the controller.
3. The controller unpacks the drain data by calling `loadDrainData`.

But in contrast to soaking, the framework has more options concerning the point at which the drain data communication is initiated. Basically, there are two policies: The client can send the task's drain data immediately, or wait for a request from the controller. We will see that both approaches have their advantages and disadvantages, but let us first include a definition, which says that drain data should not overlap:

Definition 5 (Drain data consistency)**.** For every two different tasks ① and ②, the set of drain data of task ① must be disjoint to the set of drain data of task ②.

This condition implies that drain data is the final output of the computation, and it must not be overwritten by subsequent computations.

Remark. Generally, the P2P-HOC's `writeDrainData` method should be implemented in such a way that it produces consistent drain data. Otherwise, draining will be less efficient, because draining communication will not only consist of the final output of the computation, but it will also include intermediate results.

Policy 1: Send when finished *(eager draining)* The first draining strategy is simple: Whenever a client has finished the execution of one task, it should send the task's drain data immediately.

This helps to reduce communication overhead in two ways:

- The controller does not have to send a special drain data request message to the client.
- Drain data packets can be attached to the "task-finished" message that is sent from the client to the controller.

Another advantage is that it will also work if the drain data is not consistent, as long as the task dependence graph is correct. Suppose there are two different tasks ① and ② with inconsistent drain data. Since both tasks produce overlapping drain data, there must be a data dependence between these two tasks. Let us assume that this dependence is ① $\longrightarrow$ ②.

Thus, task ② can only be executed if task ① has been finished. This also means that the controller will receive and load task ①'s drain data first. Then, when task ②'s drain data is loaded, it will overwrite the outdated results of task ①.

Policy 2: Send when requested *(lazy draining)* The alternative to eager draining is to send drain data only when it is requested by the controller. The obvious disadvantage is that it introduces communication overhead, because the controller has to send a special "request-drain-data" message. So what are the advantages?

Let us assume that one task is assigned to multiple clients simultaneously (as discussed in section 3.4). If eager draining is used, the controller would receive multiple drain packets for this task (because the task will be finished by multiple clients). To avoid that case, it is best that each client only informs the controller after it has have finished the task's execution, without sending the drain data. Then, the controller can request the drain data from one specific client (e.g., the first one that has finished the task or a client that has idle bandwidth resources).

Another advantage of this method is that the drain data communication can be processed asynchronously with a lower priority (both on the client and the controller side). Note that it will not slow down the overall computation if drain data packets are delayed.

Lazy draining will only work if the drain data is consistent. In the previous example about eager draining, tasks ① and ② had inconsistent drain data, but the controller still ended up with the correct result, because task ②'s drain data overwrote the results from task ①.

If lazy draining is used instead, this is no longer guaranteed. Although task ② will always be finished after task ①, their drain data could still arrive in reversed order if both tasks are finished on different clients.

In principal, this problem could be solved on the controller side by reordering the arriving drain data packets, so they can be loaded in a legal order (e.g., using a topological sorting of the task dependence graph). However, the buffering of the drain data packets requires additional memory. This may negatively affect both performance and predictability.

Summary:

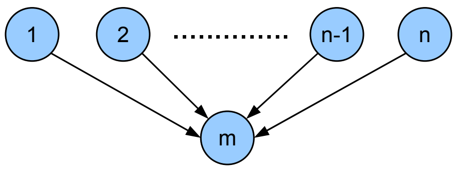
- Drain data should be consistent.
- If drain data is inconsistent, always use eager draining.
- If one tasks can be assigned to multiple clients, prefer lazy draining.

2.5 Peer-to-peer communication

The previous sections were about soaking and draining. In both cases, the communication took place between the controller and one client. In contrast, the peer-to-peer communication, which is discussed in this section, is between clients. However, the controller is still needed to determine the communication partners. That is why we introduced the P2P-HOC as a centralized peer-to-peer architecture.

A fundamental difference to soaking and draining is that soaking and draining involves only one task, but peer-to-peer communication involves two tasks. The reason is that soaking and draining describes the input and output data of a single task, while peer-to-peer communication describes the propagation of intermediate results between two task.

Peer-to-peer communication during the computation is triggered when a new task is assigned to a client. Let us assume the controller assigns task ⓜ to client **A**. In general, this task depends on other tasks ①, . . . , ⓝ:



To execute task ⓜ, data from all other tasks have to be transferred to client **A**. Note that only the controller knows which clients have finished these tasks. Therefore, the communication partners have to be chosen by the controller.

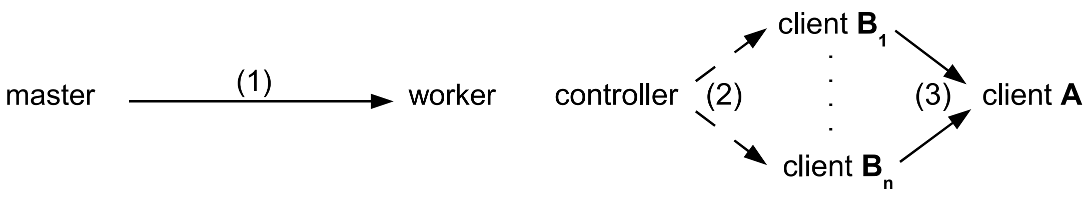


Figure 8: Shows the messages that are sent during task assignment (left: master-worker HOC, right: P2P-HOC). The dashed lines indicate P2P data request messages.

Let us assume that the controller has selected the clients $\mathbf{B_1}, \dots, \mathbf{B_n}$ to provide the data from task ①, ..., ⓝ. Then, the controller informs each of these clients by sending a P2P data request message. Note that one client can be selected for multiple tasks.

The client's P2P request handler can be implemented in a straightforward way:

1. Wait for a P2P data request, which consists of the following information:
   - Sender ID (client **A**)
   - Source task ID
   - Target task ID
2. Call the P2P-HOC method `writeDataForPeer`, which generates a P2P data packet for the communication between source and target task.
3. Serialize the P2P data packet and send it to client **A**.

2.5.1 Latency considerations

Let us compare the messages that are sent during task assignment in the master-worker HOC and the P2P-HOC, as illustrated in Figure 8. The bulk of the communication is along the arrows (1) and (3). Although the P2P-HOC still requires communication from the controller to the client (2), only the task assignment is transferred. Thus, this additional message should not significantly increase the overall communication volume, but it will increase the latency.

To counter this additional latency, the source clients $\mathbf{B_1}, \dots, \mathbf{B_n}$ should be notified as soon as possible. The ideal was that the receiver client can start to unpack the first P2P data packets immediately, while the remaining data packets are still generated. Therefore, our proposed task assignment sequence on the controller-side is as follows:

1. The scheduler responds to an open task request by selecting a task ⓜ and client **A**, which will execute this task.
2. The controller prepares the task for the client:
   For each predecessor task ⓘ ∈ {①, ..., ⓝ}
   a) Determine one client $\mathbf{B_i}$, which has finished task ⓘ.
      If client **A** itself has finished ⓘ, always select $\mathbf{B_i} = \mathbf{A}$.

   b) Send a P2P data request to client $\mathbf{B_i}$ (that $\mathbf{B_i}$ will generate P2P data between task ⓘ and ⓜ, and send it to client **A**).

   If $\mathbf{B_i} = \mathbf{A}$, it is possible to skip step b) because the communication is local (i.e., sender and receiver client are identical). Therefore, this optimization is called *local data optimization.*

3. The controller generates a soak data packet, and attaches it to the task assignment.

   (If no soak data is required, this step is skipped.)

4. The controller sends the task assignment to client **A**.

The effects of the additional latency can be further reduced if each client requests multiple tasks simultaneously. That allows the client to proceed with the next task immediately, while new tasks are fetched in the background.

This powerful optimization is discussed later in section 3.1. It is applicable to both the master-worker HOC and the P2P-HOC, but we will see that in the latter case it can be extended because of the local data optimization.

2.6 Example

In this section, we will take a closer look at the interaction between controller and clients during the computation phase. The initial situation remains the same as in in the previous example 1, which described the master-worker HOC. The only difference is that we will be using the P2P-HOC instead.

Let us briefly summarize the facts: We will use the SOR1d example from section 1.1 as the input program with the same simplified task dependence graph (its relevant part is shown in Figure 2). Again, there are three computers: One controller and two clients. The updated sequence diagram is shown in Figure 9.

Example 3. The first difference is that there is an additional initialization step before the actual computation, in which the controller creates a half-initialized `UserClient` object, and broadcasts it to all client. This object contains only the runtime parameters m and n, which were listed as constants in the loop annotation.

With this information, both clients can complete the initialization of their `UserClient` objects by allocating all arrays that are used inside the loop. In our example, there is only one array A of size $n + 1$. Once the initialization phase is completed, both clients send a task request to the controller.

The controller passes these task requests to the scheduler. Analog to the master-worker example, the scheduler assigns the only available task $(3, -2)$ to client **A**, and blocks the request of client **B**. The preparation of the task is also similar: As it is the first task, the soak data generated by the controller has to include the same data that would be sent by the master (which always sends all data that is required to compute the task).

As task $(3, -2)$ does not depend on other tasks, there is no need for peer-to-peer communication between clients. Therefore, the task preparation is finished, and the controller can send the task assignment to client **A**.

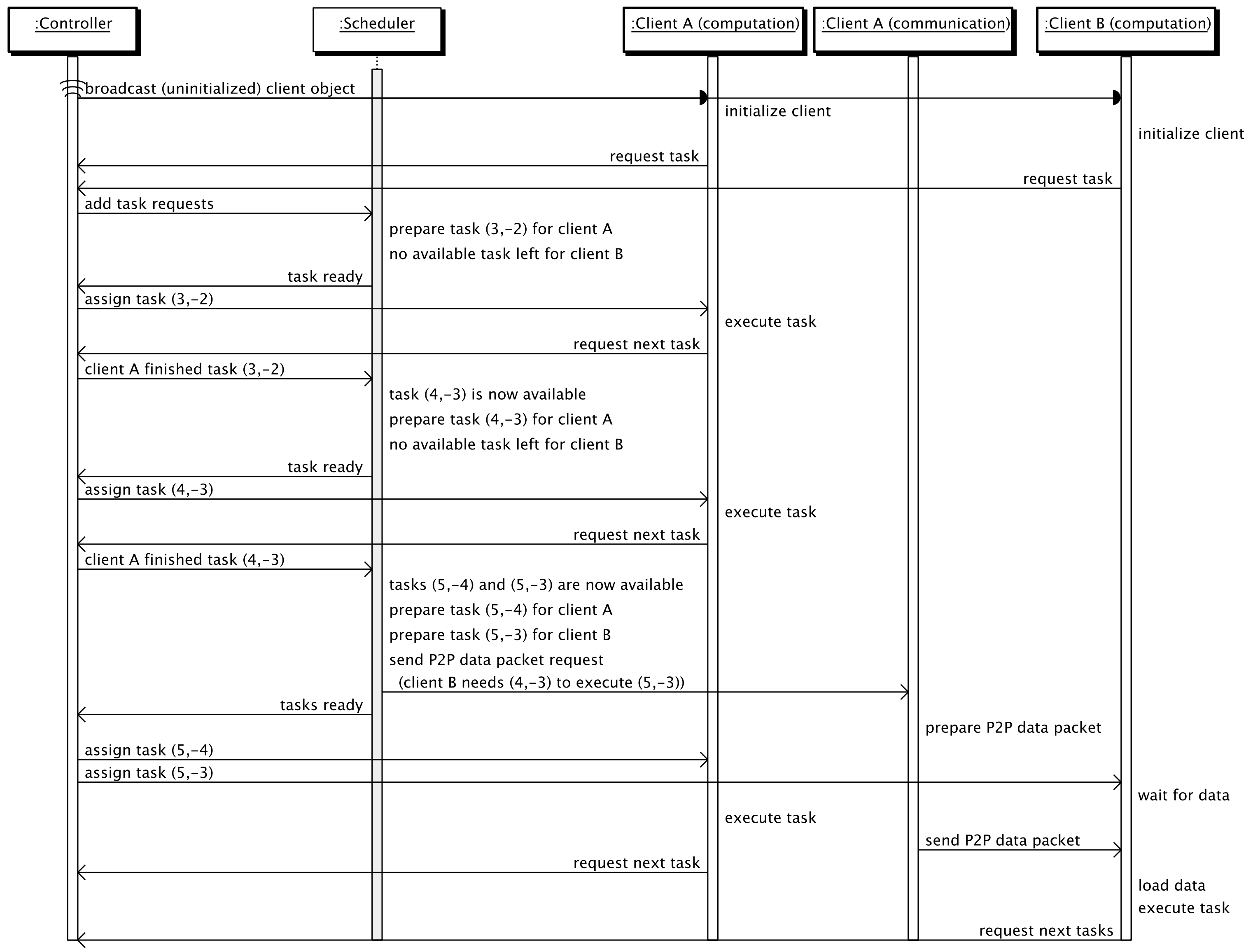

:Controller
:Scheduler
:Client A (computation)
:Client A (communication)
:Client B (computation)
broadcast (uninitialized) client object
initialize client
initialize client
request task
request task
add task requests
prepare task (3,−2) for client A
no available task left for client B
task ready
assign task (3,−2)
execute task
request next task
client A finished task (3,−2)
task (4,−3) is now available
prepare task (4,−3) for client A
no available task left for client B
task ready
assign task (4,−3)
execute task
request next task
client A finished task (4,−3)
tasks (5,−4) and (5,−3) are now available
prepare task (5,−4) for client A
prepare task (5,−3) for client B
send P2P data packet request
(client B needs (4,−3) to execute (5,−3))
tasks ready
prepare P2P data packet
assign task (5,−4)
assign task (5,−3)
wait for data
execute task
send P2P data packet
request next task
load data
execute task
request next tasks

Figure 9: Sequence diagram showing the computation phase using the P2P-HOC. The task dependence graph for this example (SOR1d) is shown in Figure 1.

After loading the soak data, client **A** can immediately execute the task, because no P2P data packets are required. After the task is terminated, the client sends a notification to the controller with a request for the next task, but beside that it does not send any data[6]. When the message has arrived, the controller informs the scheduler, which updates the task dependence graph, thus unlocking task $(4, -3)$. Note that, up to this point, there was no substantial difference compared to the master-worker HOC.

But now, we have to depart from the last example: There are two requesting clients, but only one available task. In the master-worker example, the scheduler decided to assign the task to worker **B**, but it could as well have assigned it to worker **A**. Here, in the P2P-HOC example, the scheduler is almost forced to assign the task to client **A**, because the required data is already available on this client.

The assignment and execution of task $(4, -3)$ on client **A** is analog to the execution of task $(3, -2)$.

The completion of task $(4, -3)$ unlocks two further tasks: $(5, -4)$ for client **A** and $(5, -3)$ for client **B**. Again, the task assignment for client **A** does not require communication, but client **B** needs data from task $(4, -3)$, which has been executed by client **A**. Therefore, the scheduler also sends a P2P data request to client **A**.

When the message is received on client **A**, its P2P data request handler starts to prepare the data packet, and then sends it client **B**. Meanwhile, client **A** has received the task assignment and starts to execute task $(5, -4)$.

It is important to note that the simultaneous execution of the task and the generation of the P2P data packet does not lead to race conditions. Although both operations have access to the same memory, there is no need for synchronization, as we will see in the next section.

Let us return to the example, where client **B** has received its task assignment. However, it cannot immediately start to execute its task $(5, -3)$, because it has to wait for the P2P data packet. When the packet has arrived, the client unpacks it and can start to execute the task.

Finally, both clients finish the execution of their assigned tasks $(5, -3)$ and $(5, -4)$, and request new tasks from the controller.

This example illustrated the communication in the P2P-HOC, except drain communication, which becomes more relevant at the end of the computation. Since the example only covered the first five tasks of the computation, both clients only communicated once. In realistic applications, however, most of the communication consists of P2P communication between clients.

Table 1 shows a comparison between the previous master-worker example and this example. At the begin of the computation, both approaches are very similar, but the advantages of the P2P approach become more apparent in the course of the computation.

HOC type	P2P-HOC	master-worker HOC
Client/worker initialization	Initialization phase before the computation	Workers allocate resources on demand
Task assignment includes	Soak data (data that is required to compute the task, but which is not available on any client)	All data required to compute the task (all data that is read accessed)
Next task request includes	Drain data (final results of the computation)	All data generated by the task (all data that is written)
Location of the intermediate results	Distributed among the clients	Centralized on the master
Impact of the scheduler	Performance critical (can minimize the communication costs by maximizing local communication)	Only limited (might prefer idle workers with more computation power or faster network connection)

Table 1: Comparison between the master-worker HOC (example 1) and the P2P-HOC (example 3).

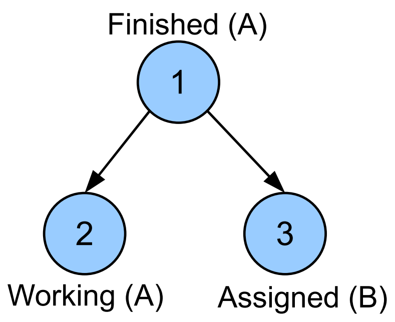


Figure 10: Initial situation.

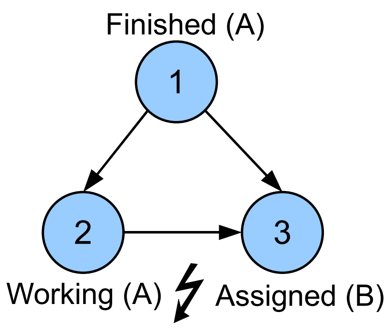


Figure 11: The updated situation, including the data dependence from task ② to task ③.

2.7 Asynchronous generation of P2P data packets

Consider the situation shown in Figure 10: There are two clients **A** and **B**, and three tasks. Task ① has been finished by client **A**, which is currently working on task ②. At the same time, task ③ has been assigned to client **B**. This task requires P2P data from task ①.

This it exactly the same situation as in the example from the last section. Is it possible that client **A** can continue working on task ②, and at the same time generate the P2P data packet (without any synchronization)?

Let us assume that there are race conditions, that means the execution of task ② overwrites data needed to generate the P2P data between task ① and task ③. In other words, task ② overwrites data that is read by task ③. This data dependence, however, is missing in the task dependence graph.

There are two possible directions for the data dependence, either ② $\longrightarrow$ ③ or ③ $\longrightarrow$ ②. In both cases, it contradicts with the fact that both task ② and ③ have been assigned simultaneously (to different clients[7]).

For example, Figure 11 shows the task dependence graph after adding ② $\longrightarrow$ ③. A correct task scheduler could not have assigned task ③ to client **B**, because this task depends on the still unfinished task ②. Consequently, the assumption that there are race conditions must be wrong.

In general, there are more tasks involved, but the argument remains the same: If there was a conflict between the execution of one task ⓘ and the generation of P2P data for another task ⓙ, it implies that there is a data dependence between these two tasks.

Although this data dependence may be missing in the task dependence graph, because it is part of the transitive hull of the existing data dependences, it still has to be respected by the task scheduler. This contradicts with the fact that a correct task scheduler must not assign both tasks simultaneously to different clients, as it has happened in the initial assumption.

2.8 Interface of the P2P-HOC

This section summarizes the most important methods of the P2P-HOC, and contains information about the code that is generated by LooPo. All methods that are not related to topics that are discussed in this work have been skipped.

It is possible — although not intended except for debugging purposes — that the user creates its own P2P-HOC application without LooPo, but by providing its own implementation of the following methods. The reason why we advise against using the P2P-HOC directly, is that it has been specifically designed for automatic code generation.

The difference between the presented P2P-HOC and a standard task farming component is that the P2P-HOC not only requires that the problem is divided into tasks,

[6] We will assume that there is no drain data for task $(3, -2)$, because it is the first task. If there was drain data, it would be sent along with the "request-next-task" message.

[7] We will see in section 3.1.2 that, under certain conditions, it is possible to assign both task simultaneously to the *same* client. The reason is that tasks are executed *sequentially* on each client, so the data dependence is again respected.

but these tasks must also be supplemented by communication between these tasks (P2P data packets), as well as soaking and draining communication. Experience shows that implementing efficient P2P communication can be error-prone for a human. However, a more user friendly interface could be build atop of the low-level implementation.

2.8.1 Controller interface: UserMaster

void *startUp* (String[] args) The startup function can be divided into three phases:

1. Execute the original code from the input program, which precedes the loop that has been selected for parallelization (*pre-loop code*).
2. Build the task dependence graph, and pass it to the task scheduler.
3. Remove redundant dependences. This is an optional optimization of the task dependence graph. It is discussed in section 3.5.

(The framework provides a suitable implementation for the task scheduler and the task dependence graph.)

void *end* () Execute the original code from the input program, which follows the loop that has been selected for parallelization (*post-loop code*).

long *joinTaskResult* (int clientID, UserTask taskResult) Informs the task scheduler that the specified client has finished the given task.

Serializable *writeSoakData* (int[] taskID) Generates a soak data packet that contains data located on the controller, which is required to execute the specified task.

If the task requires no soak data, this method should return `null`, so the framework can avoid sending an empty soak data packet.

void *loadDrainData* (int[] taskID, Serializable drainData) Updates the data stored on the controller by loading the drain data that has been received from a client.

2.8.2 Client interface: UserClient

***UserClient* ()** The constructor of the `UserClient`, which is executed on the controller process. Afterwards, the `UserClient` object is serialized and broadcasted to all clients, which will finish the initialization by calling the `init` method.

void *init* () This method is executed on the client to finish the initialization of a `UserClient` object received from the controller.

Its intended purpose is to allocate arrays that should not be sent from the controller to the client (which would be the default behavior if the arrays were allocated by the constructor, because the entire `UserClient` object gets serialized). If all initialization has been already done by the constructor, the functionality provided by this method is not needed, and it should be implemented with an empty body.

void *executeTask* (int[] taskID) Executes the specified task. This method contains the computation code, which is parameterized by the task's ID.

void *loadSoakData* (Serializable soakData, int[] taskID) Updates the data stored on the client by loading the soak data that has been received from the controller.

Serializable *writeDrainData* (int[] taskID) Generates a drain data packet for the specified task.

If the task generated no drain data, this method should return `null`, so the framework can avoid sending an empty drain data packet.

void *loadDataFromPeer* (UserDataPacket data, int[] sourceTaskID, int[] targetTaskID)

Unpacks the data packet that has been received from another client, which has finished the source task on which the currently executed target task depends.

UserDataPacket *writeDataForPeer* (int[] sourceTaskID, int[] targetTaskID)

Generates a P2P data packet, which consists of data produced by the source task that has been finished on the current client. The receiver of this data packet is the client to which the target task has been assigned.

(In contrast to the `writeSoakData` and `writeDrainData` methods, it is currently not possible to return `null` to avoid sending an empty P2P data packet. The reason is that the framework will only call `writeDataForPeer` if the task dependence graph contains a dependence between the source and target task. Therefore, it assumes that data has to be communicated because of the data dependence.)

3 Task scheduling

This sections discusses task scheduling, especially in reference to the P2P-HOC. Let us first summarize some traits of good task scheduling strategies.

Load balancing It is important to utilize the maximum computation power of all available computers. Otherwise, scalability would be severely hurt.

In general, load balancing is easy to achieve because of the task farming approach. It can become a problem, however, if the task dependences graph is too small to keep the available processors busy.

Locality (P2P-HOC only) P2P communication takes place between tasks. Thus, the scheduler should increase the locality of the communication by assigning connected tasks to the same client.

It is not relevant for the master-worker HOC, because there the communication between master and worker is fixed (two communications per task). In contrast, in the P2P-HOC, it is possible to eliminate local P2P communications.

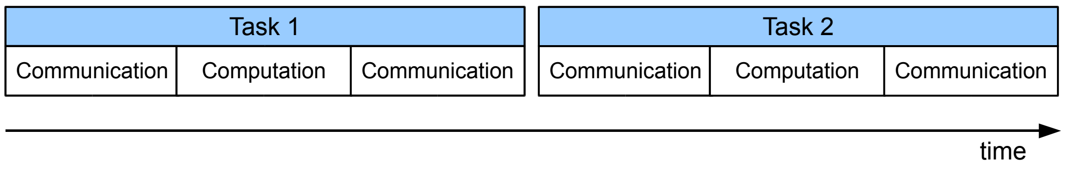


Figure 12: Sequential execution of two tasks in the master-worker framework.

Efficiency The efficiency of the task scheduler's implementation becomes important if it is operating on huge task dependence graphs (more than 100,000 nodes). It is less of a concern if the graphs are small (less than 1,000 nodes).

Another factor is the average task execution duration. In general, the more the tasks become fine-grained, the more important is efficiency.

3.1 Improved task prefetching

This section consists of two parts: First, we will briefly present task prefetching as it is realized in the original master-worker framework. After that, we will introduce an improved task prefetching approach for the P2P-HOC that relies on the possibility to eliminate client-local P2P communication.

3.1.1 Task prefetching in the master-worker framework

In the master-worker HOC, the execution of a task consists of three phases:

1. The master communicates the data to the worker
2. The worker computes the results
3. The worker communicates the results back to the master

The problem is that if all tasks are fetched and executed sequentially, as shown in Figure 12, the available resources are not used to full capacity:

- During the communication phases, the network is busy, but the processor is idle. Thus, available computation power is wasted.
- During the computation phase, the processor is busy, but the network is idle. Thus, available network capacity is wasted.

To use the resources more efficiently, task prefetching overlaps communication with computation by requesting multiple tasks at the same time. This improves performance because of the pipelining effect, as illustrated in Figure 13. However, if no tasks are available (e.g., all other tasks depend on the task that is currently executed), prefetching is not applicable because the task request will block. To counter this effect, the number of tasks can be increased by using smaller tile sizes (in the code generation).

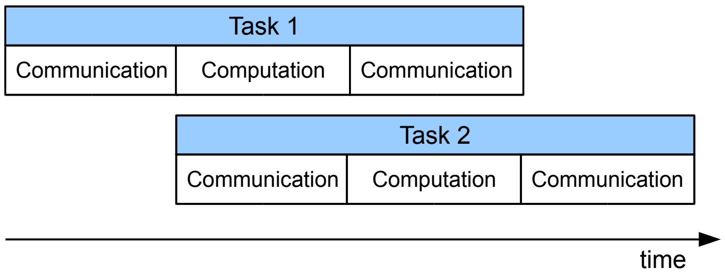


Figure 13: Task prefetching in the master-worker framework: Overlap communication with computation.

Another important reason for task prefetching is that it allows the parallel execution of tasks on the worker. For example, if a worker process is running on a dual-core computer, it may request two tasks at once, and execute them in parallel. This is better than starting two separate worker processes on the same machine, because it halves the memory consumption.

3.1.2 Improved task prefetching in the P2P framework

There is an important differences between task scheduling in the master-worker and in the P2P-HOC: In the master-worker HOC, an unfinished task can either be available or blocked. This is independent of the workers. In contrast, in the P2P-HOC, a task can be available only to certain clients, but not to all of them. But let us define these terms step by step:

Definition 6 (Available task)**.** A task is *available* if it has not been finished, but all its source tasks have been finished.

Note that this definition implies that tasks can be assigned to multiple clients. Although, in principle, this is feasible and could even prove useful in some situations (e.g., to improve system stability in case of clients failures), it can also create data races if no special care is taken. Therefore, task duplication is not supported by our current framework implementation, but section 3.4 contains a short discussion about possible extensions to the protocol.
However, in this current section and also in the rest of this thesis, we will assume that each task can only be assigned and finished by one client.

Definition 7 (Blocked task)**.** A task is *blocked* if at least one of its source tasks is not finished. (In other words, an unfinished task is blocked if it is not available.)

The idea of improved prefetching is as follows: All clients together are working simultaneously, but each client on its own executes its assigned tasks sequentially. That means a new task can be assigned to a client if the task is available assuming that the client has finished all its assigned tasks.

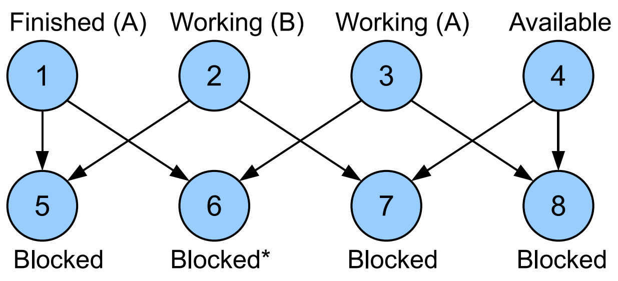


Figure 14: Initial scenario: Client **A** requests to prefetch the next task. Tasks that are blocked but are available to client **A** are labelled as **blocked***.

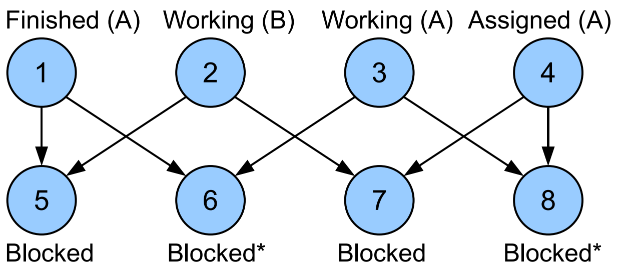


Figure 15: Updated scenario: Task ④ has been assigned to client **A**.

Definition 8 (Client specific available task)**.** A task is *available to client* **A** if the task has not been finished, but all its source tasks have been either finished or have been assigned to client **A**.

Obviously, one task that is available (according to definition 6), is also available to every client (according to definition 8). However, a task can be blocked, but at the same time be available to one specific client. This can be used by the scheduler to improve prefetching efficiency, as illustrated in the following example.

Example 4. Let us consider the scenario shown in Figure 14. There are two clients called **A** and **B**. **A** has just finished ① and is now executing ③. The second client **B** is working on ②. Which task should the scheduler now assign to client **A**? In other words, which task should be prefetched by client **A**?

The only available task is ④. In the classical task farming paradigm, there is no other choice. However, after the scheduler assigns ④ to client **A** (Figure 15), there are no available tasks left, so client **B** has to wait until further tasks are finished before it can get its next task. There are two possibilities:

1. **A** finishes task ③, which unlocks task ⑥
2. **B** finishes task ②, which unlocks task ⑤

However, until one of these conditions is met, no prefetching is possible.

Lets go back to the initial situation (Figure 14): In the P2P approach, the scheduler has more options. Not only is task ④ available to client **A**, but task ⑥ can be assigned

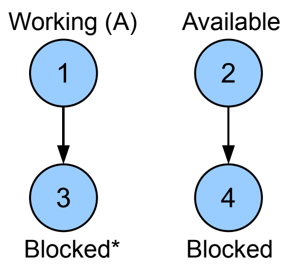


Figure 16: Scenario: Two clients **A** and **B**. Client **A** requests to prefetch the next task. In order to minimize communication, it is best to assign task ③ to **A** and leave ② and ④ to client **B**. Thus, all communications are local and can be eliminated.

as well. The reason is that all its dependences are available at the time when task ⑥ will be executed. As the dependences are local, they can be optimized away. Thus, no data packets have to be requested.

Therefore, the better strategy for the scheduler is to assign ⑥ to client **A**, and leave ④ to client **B**.

A slightly worse alternative that also relies on improved prefetching would be to assign ④ to client **A** (as shown in Figure 15), and leave task ⑤ to **B**. This is possible, because after client **A** has finished task ①, task ⑤ becomes available to client **B**.

There are two reasons why the former scheduling strategy is preferable:

Less communication The idea behind improved prefetching is to maximize local communication, which can be completely eliminated. After assigning task ④ to client **A**, the assignment of ⑤ to client **B** forces the communication of data from client **A** to client **B**[8].

Flexibility If we introduce a third client **C** that has neither been assigned nor finished any tasks yet, the only available task for client **C** is ④. In other words, an independent task like ④ without sources is too valuable, so it should be assigned only if there is no other possibility.

3.1.3 Asynchronous P2P data packet generation with prefetching

This section addresses the following question: Can task prefetching, especially improved task prefetching, interfere with the asynchronous generation of P2P data packets (see section 2.7), or introduce other race conditions?

The short answer is that standard task prefetching cannot interfere, and improved task prefetching can also not interfere *assuming* that all tasks are executed sequentially by the client.

[8] Actually, in this example, the communication cannot be completely avoided, but it is only delayed. A better example to illustrate the idea is shown in Figure 16.

1. Standard (non-improved) task prefetching cannot interfere, because only *available* tasks will be assigned:
    a) There are no data dependences between these tasks, thus the parallel execution of the assigned tasks cannot interfere.
    b) The execution of the assigned tasks cannot interfere with the generation of P2P data for another tasks, because otherwise there would be a data dependence between these two tasks, which would contradict the assumption that the assigned task was available.

2. Improved task prefetching complicates the situation, because the assigned tasks can depend on other tasks that have not been finished. Therefore, there can be race conditions between these tasks if they are executed in parallel by the client.

    However, if the clients executes the tasks sequentially (in the same order in which they were assigned by the scheduler) the race condition is no longer possible. Since all tasks that have been assigned were available to the specific client **A**, they are available to all clients (i.e., they do not longer depend on tasks that are not finished) after the client **A** has finished the execution of all its previous tasks.

It turns out the sequential execution of the assigned tasks assures correctness, but it is more restrictive than necessary. It is sufficient to execute the assigned tasks in such a way that the dependences between the tasks are not violated. Thus, tasks could be executed in parallel within the clients (see section 3.3).

3.2 Scalability

Remark. This section discusses scalability aspects of the task scheduler that should be able to efficiently operate on task dependence graphs, even if the number of tasks increases.

Throughout this thesis, we are concerned that the overall computation scales with the number of clients and the problem size. As the task dependence graph grows with the problem size, an inefficient implementation of the scheduler can negatively affect overall scalability. Therefore, we will shortly list the requirements of the scheduler, which is performance critical, both time- and space-critical:

Time-critical: There are two implementation challenges:

- The number of task requests increases with the number of tasks and clients. If the controller cannot handle all requests, the clients have to wait for tasks, even if there were enough task available.
- The scheduler must be able to join finished tasks and assign new tasks simultaneously, because task requests can block until there are new tasks available (in other words, until old task have been joined). Therefore, access to the dependence graph has to be serialized, which limits the benefits of multicore processors[9].

[9] In our current implementation, the controller utilizes dual processors, but it might be possible to

Space-critical: The task scheduler's memory consumption consists mainly of two parts:

1. The task dependence graph has to be stored. The graph grows with the number of tasks.
2. For each client, the scheduler has to store the following information:
   a) The set of tasks that has been *assigned* to the client
   b) The set of tasks that has been *finished* by the client

   The memory consumption grows with the number of tasks and clients.

Our experiments contained task dependences graphs with more than 300,000 tasks (computed by 80 clients). This is about the limit that can be efficiently processed by our current implementation.

In order to process huge graphs efficiently, the scheduler has to rely on heuristics. However, a sophisticated precomputed task assignment strategy is not reasonable, anyway, because of the lack of information:

- No information about individual task durations is available.
- No information about the duration of an individual communication is available.
- The client's computing power may vary unpredictably, in the course of the computation, if multiple users have access to the computer.

As a result, our task scheduler uses a greedy algorithm for task assignment:

1. Create a list of all tasks that are available to the requesting client.
2. Use a heuristic to evaluate each task with regard to the following optimization criterions:
   - Minimize remote communication (between different clients)
   - Maximize local communication (between the same client)
   - Always assign task chains to the same client[10]
3. Assign the best evaluated task to the requesting client.

We will skip the details of the optimization heuristic at this point, but the algorithm is sketched in appendix B.

A disadvantage of the presented task assignment algorithm is that, for each task request, all available tasks have to be determined and evaluated. It is difficult to avoid this recomputation though, because the optimization criterion depends on the client that has requested the task, and also on the tasks that have been finished by other clients.

improve the support of multiprocessors. For example, multiple tasks can be joined simultaneously. Another idea is to processes multiple task requests in parallel.

[10] A task chain is a sequence of tasks ①, …, ⓝ, where each task (except the first and the last) has a single incoming dependence from the previous task, and a single outgoing edge to the next task: ① → … → ⓝ. Consequently, task chains should be executed sequentially by the same client.

Therefore, the evaluation score cannot be reused for requests from other clients. In case of improved task prefetching, the list of available tasks also depends on the requesting client.

However, if the same client asks for multiple tasks, for example, because of task prefetching, it is not necessary to compute and evaluate the list of available tasks from scratch. Instead, the previous results could be updated to avoid the expensive reevaluation.

3.3 Outlook: Client internal parallelism

Remark. The parallel execution of tasks, as discussed in this section, is not implemented in our prototype framework. Currently tasks are always executed sequentially within the clients (P2P-HOC) or workers (master-worker HOC).

3.3.1 Shared memory parallelism

Although the P2P-HOC mainly targets distributed heterogeneous clusters or Grid environments, there is also a trend towards computer-architectures with more and more processors, which should not be neglected. There are two basic possibilities to take advantage of these multiple processors:

1. Start multiple clients on the same machine.

2. Start one client per machine, but let the clients execute multiple tasks in parallel.

The first approach is simple, but the second one is preferable for two reasons: It is more memory efficient, and it can use shared memory to avoid explicit communication.

In case of the master-worker HOC, the parallel version is easy to realize by adjusting the number of prefetched tasks to the number of the machine's processors (or slightly more to assure that communication and computation are still overlapping). Unfortunately, the client internal sequential execution of the P2P-HOC tasks, as suggested in section 3.1.3, eliminates all parallelism within the client:

Example 5. Consider the task dependence graph shown in Figure 17 (left). There is only one client (or worker) **A**, and we will assume, in this example, that the client will prefetch as much tasks as possible.

In the master-worker HOC, there are two available tasks ② and ③. These two tasks will be assigned to worker **A** and could be executed in parallel. After both tasks are finished, tasks ④ becomes available and can be also be assigned and executed.

In the P2P HOC, the controller will first assign the tasks ② and ③. After these two tasks have been assigned, task ④ becomes available to client **A** as well, and will also be assigned. The dependences between these three assigned tasks is shown in Figure 17 (middle). If the tasks are executed sequentially, the tasks are executed in the same order, in which they were assigned: ② $\rightarrow$ ③ $\rightarrow$ ④.

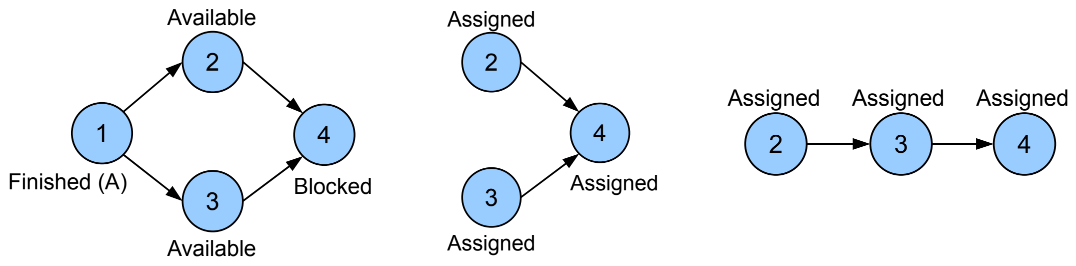


Figure 17: Left: The original task dependence graph: There is only one client **A** and two available tasks (② and ③). (After both tasks have been assigned, task ④ also becomes available to client **A**.)
Middle: The part of the dependence graph containing only the assigned tasks ②, ③ and ④.
Right: Shows the execution order of the assigned tasks after sequentialization (note that this is not a task dependence graph).

This example shows that it is generally not possible to execute all tasks in parallel. Although the dependences between the tasks force that task ④ is executed after ② and ③, the sequentialization is too restrictive, because it prevents the parallel execution of ② and ③.

The solution is simple: The client also needs a task scheduler. This scheduler operates on a simplified task dependence graph that only contains the client's assigned tasks with their dependences. Its purpose is to assign the tasks that it requests from the controller to the processors, without violating any of their dependences. Figure 18 illustrates this idea assuming that both clients run on quad-core machines. Note that no explicit communication between the tasks is required because of the shared memory. The synchronization is done by the client's task controller.

3.3.2 Maximize parallel execution within clients

The option to execute tasks in parallel yields new challenges for the task scheduler. In this section, we will describe how the task assignment strategy, discussed in section 3.2, can *prevent* parallel execution within clients. The reason is that its sole optimization criterion is to avoid communication by keeping P2P communications local whenever possible. To solve this issue, a new optimization criterion will be introduced at the end of this section.

Example 6. The task dependence graph, shown in Figure 20, consists of four independent task chains. We will assume that there are two clients **A** and **B**, running on identical machines. We further assume that each task is identical in terms of computation time and communication volume. So, is there an optimal task assignment strategy to distribute the tasks among the clients? Obviously, each client should compute two (of the four) task chains, but is it relevant in which order the task are executed?

Without client internal task execution, each client can only execute one task at the same time. To improve the cache performance, it is best to execute all tasks from the

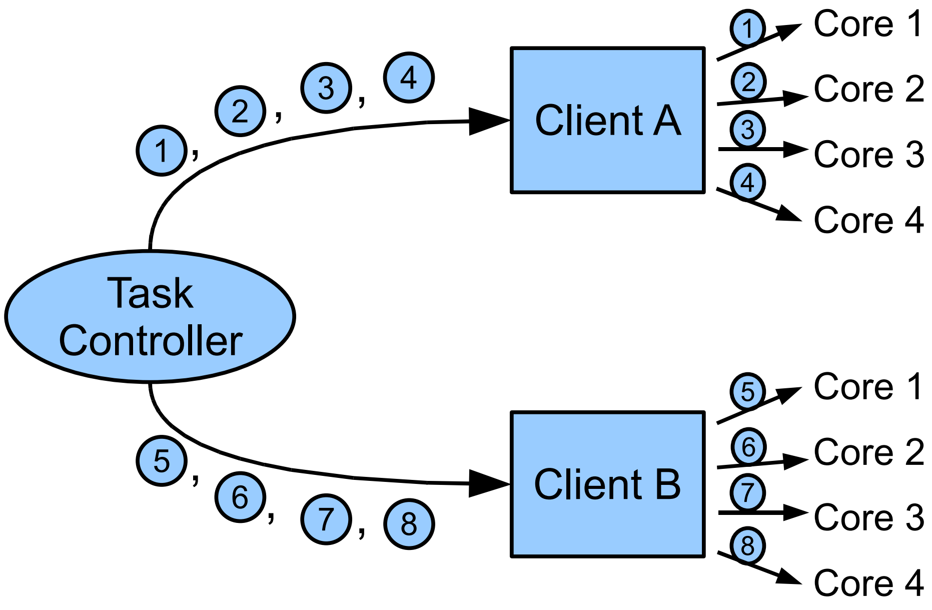


Figure 18: Illustration of two-stage task scheduling: First, the controller's task scheduler assigns tasks to clients. These tasks will then be assigned to the available processors by the client's task scheduler.

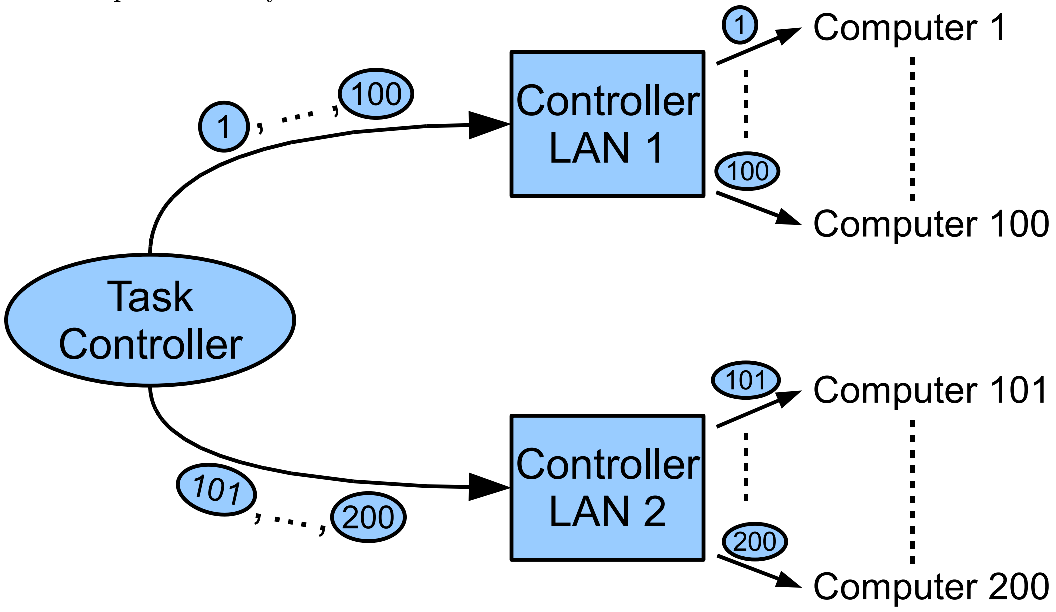


Figure 19: Task scheduling with support for subnets: Tasks are not directly assigned from the controller's task scheduler to the clients. Instead, a task assignment passes through the network hierarchy, where each subnet has its own task scheduler.

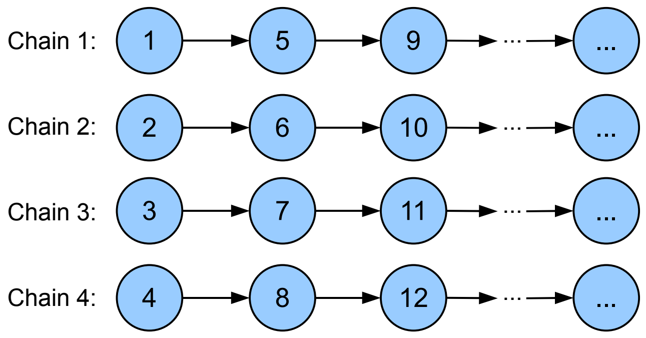


Figure 20: Four independent task chains.

first chain before continuing with tasks from the second chain. This is also according to our statement that the task scheduler should *always* prefer task chains (see section 3.2).

However, with client internal task execution, it depends on the number of processor that are available on each client. If both clients are running on dual processor machines, it is best to assign each client tasks from both chains simultaneously. Otherwise, each client could only utilize one processor, while the other processor remains idle. It is important to note that increasing the number of prefetched tasks will not help to avoid that problem.

Let us summarize the observations from the example:

- Task chains should always be preferred *if* clients will not execute multiple task in parallel, because task chains are optimal for sequential execution.
- Preferring task chains can limit the parallelism within a client, because task chains are unsuited for parallel execution.

Since the task scheduler should performance well in both cases — with and without client internal parallelism — a solution has to be found.

One idea is to ”cut” long task chains by adding a new optimization criterion to the task scheduler, which penalizes tasks that depend on other tasks that are not finished. This penalty should increase with the resulting delay. Tasks that can be executed immediately are not concerned. Therefore, these tasks will be preferred, thus increasing the number of tasks that can be executed in parallel.

Let us take a closer look at the idea:

What are the preconditions? Only a task, which cannot be immediately executed by the requesting client, will be penalized. Thus, at least one of its predecessor tasks has to be assigned but not finished by the requesting client. Additionally, it is only sensible to penalize tasks if the requesting client has more than one processor. Otherwise, the optimization will be counterproductive.

In theory, the optimization should be turned off if the requesting client still has enough assigned but not finished task to utilize all of its processors. In example 6, we concluded that a client with two processors should not be assigned exclusively tasks from a single chain. Instead, it is optimal to take tasks from two chains, one for each processor, but it is slightly worse to take tasks from three chains.

However, at the time when the task request is processed, the task scheduler — running on the controller — cannot determine exactly how many of the assigned tasks have been already finished by client. Therefore, it is difficult to decide whether there are enough assigned task left to utilize all processors of the client.

What does it mean to "cut" a task chain? If the delay of a task (from a task chain) exceeds a certain limit, the penalty will effectively prevent that the next task from the chain will be chosen, since all other available tasks will be preferred. As a result, the task chain is temporarily cut at this point until enough of its predecessor tasks will be finished, thus reducing the penalty caused by the delay.

What length is best? The optimal length of the chains depends on the number of available processors on the requesting client and on its already assigned tasks. In general, having more processors will favor shorter chains. However, if the client has only one available processor, chains should be infinitely long.

Can it be generalized beyond chains? Yes, the idea is not limited to task chains, although task chains are the most urgent targets. However, determining the delay becomes more expensive. In the case of task chains, the delay is equal to the number of unfinished predecessor tasks, because task chains have to be executed sequentially. In general, however, the parallel execution of tasks within the client has to be considered, too.

3.3.3 Support for hierarchical networks

The idea of parallelism within clients can be extended to improve performance in hierarchical networks where the communication costs between clients from different subnets are not uniform. But first, let us motivate the need for subnet support with an example:

Example 7. The user has access to two separate LAN networks, each consisting of 100 computers. The communication between computers from the same network is fast, but the communication between computers from different networks is slow.

To increase the performance, the tasks should be distributed among the two local networks, so that most of the P2P communication takes place between computers from the same network. In other words, the communication should be as local as possible if we use a wider definition of locality (i.e., not on the same computer, but on the same network).

In our current framework, it is not possible for the task scheduler to support subnets, because no information about them is available. Thus, the first step is to describe the network hierarchy to the task scheduler. Therefore, we will use the following model:

- The network consists of multiple subnets.

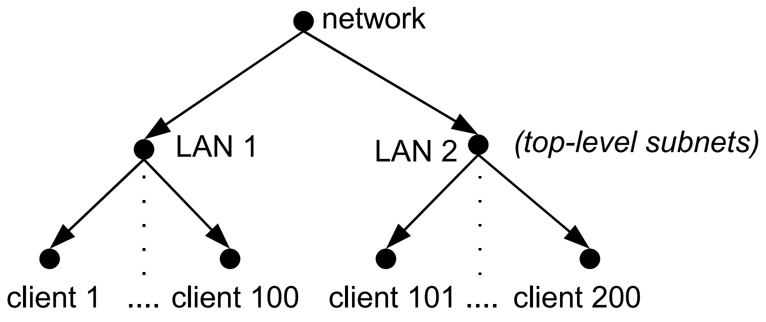


Figure 21: Shows the network in example 7, which consists of 200 clients that are organized in two independent subnets.

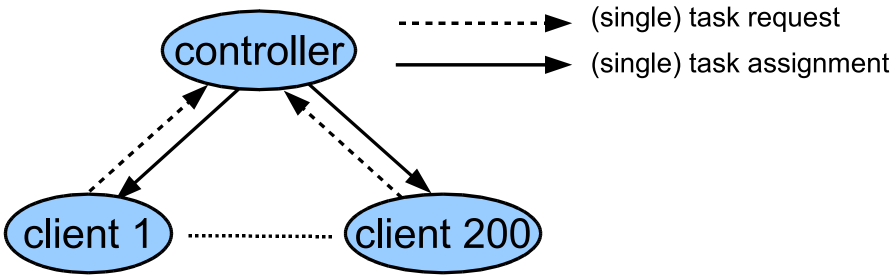


Figure 22: Original task assignment scheme: All clients directly request and receive tasks from the controller.

- Each client is a subnet, and a subnet can again consist of multiple subnets.

As a consequence, the network is described as a tree of subnets, where the clients represent the leaves. Figure 21 shows the resulting tree for example 7.

In principle, the task scheduler could use the information from the subnet tree to optimize the task assignment. The realization is difficult, however, because the scheduler would have to solve a complex optimization problem under strict time constraints. Therefore, the complexity has to be reduced, even at the cost of precision.

Our proposal is to add an intermediate layer, representing the subnets, between the task controller and the clients. Instead of directly assigning tasks to clients, we will use *multi-stage task scheduling*:

1. The task controller assigns tasks to the top-level subnets.
2. The subnets assign tasks to their associated subnets until the task is finally assigned to a client.

It is called multi-stage because the second step can be repeated multiple times, depending on the height of the tree. Figure 23 illustrates this modified task assignment scheme for example 7.

The advantage to the original task assignment scheme (shown in Figure 22) is as follows: Since there are multiple schedulers — as each subnet has its own scheduler —

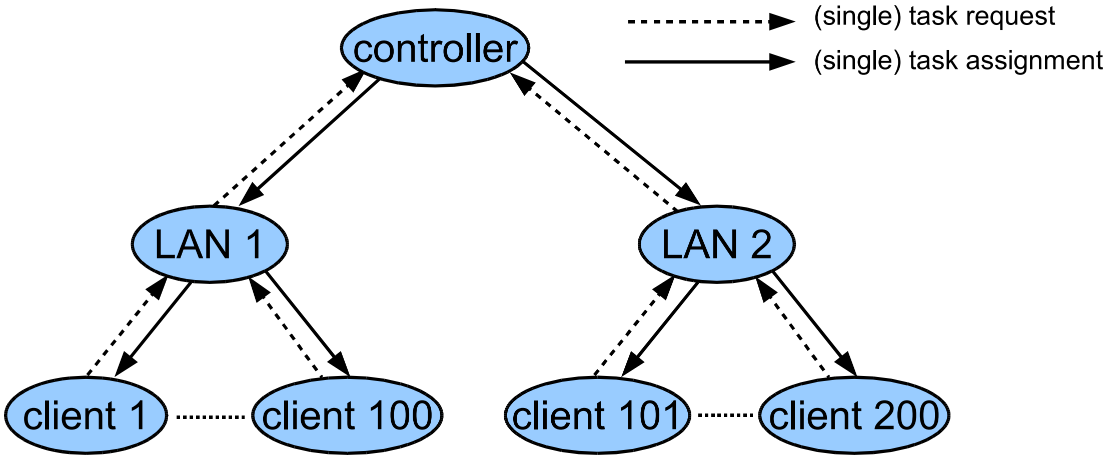


Figure 23: Simplified task assignment scheme with support for hierarchical networks: Both subnets LAN 1 and LAN 2 only forward task requests from its children subnets to the parent subnet.

a single task scheduler does not have to consider all clients that are part of its network. Instead, only its associated subnets have to be considered, thus significantly simplifying the complexity of the optimization problem. Let us illustrate the differences with an example:

Example 8. We will again use the scenario from the last example, as it is shown in Figure 21. Let us assume that client 1 (which is part of LAN 1) requests a task from the controller.

In the original approach (Figure 22), the task request is directly sent to the controller's task scheduler, which will respond by assigning the best suited task to client 1. To determine the optimal task, information about *all* clients have to be considered, especially the following information:

- The set of tasks that are finished by the client
- The set of tasks that are assigned to the client (but not finished)
- The set of clients that belong to the same subnet

In the new scheme (Figure 23), client 1 requests a task not directly from the controller's scheduler but from the task scheduler of LAN 1. To respond to a task request, the controller's scheduler only has to consider its children subnets (LAN 1 and LAN 2). For example, as LAN 1 requested the task, the controller's scheduler has to determine a task that is optimal for LAN 1 (but less suited for LAN 2). It is important to note that *no clients* are considered.

However, the scheduler of LAN 1 can only assign tasks to its children subnets (in this case, client 1) if it has requested these tasks from the task scheduler of its parent network (in this case, the controller's scheduler). As we will see after this example, it is therefore essential that task prefetching is used by all subnets (LAN 1, LAN 2, client 1, . . . , client 200). Otherwise, the quality of the task assignment will be significantly reduced.

To respond to the task request from client 1, the scheduler of LAN 1 selects the best-suited task from its set of prefetched tasks. Note that only tasks can be assigned, which are *available to client 1.* If all tasks are blocked, the client's task request must be blocked as well.

Let us summarize the important differences:

- In the original task assignment scheme, the task request has been processed by the controller. Therefore, *one* optimization problem (*controller* → *client 1*) has to be solved.
- In the new task task assignment scheme, the task request has been split into two phases. Therefore, *two* optimization problems have to be solved:
    1. Coarse-grained distribution of tasks to subnets (*controller* → *LAN 1*)
    2. Fine-grained distribution of tasks to clients (*LAN 1* → *client 1*)

Why is task prefetching essential? In the middle of the example, we noted that each subnet has to use task prefetching to improve the quality of task assignment. Indeed, otherwise, the whole idea of reducing the controller's load by first delegating the task assignment to the subnets, and then solving a less complex optimization problem, would fail.

Let us first examine the situation when subnet schedulers do not use task prefetching but only request tasks on demand. In that case, whenever a client requests a task from its subnet scheduler, the request will be relayed from subnet scheduler to parent subnet scheduler until the request finally arrives at the topmost controller. In other words, the original task request from the client will be processed by the topmost controller, which is responsible for the whole network.

Effectively, the assignment process is equivalent to the original task assignment scheme without support for subnets (Figure 22). However, the quality of task assignment will even be reduced, as the topmost controller has to respond to a task request from an unknown client (the controller only knows which subnet has requested the task).

How can prefetching fix these shortcomings? First, the initial situation is different, because each subnet scheduler has already requested several tasks from its parent subnet, even before the client sends its task request. This updated task assignment scheme is illustrated in Figure 24. As a result, there are two immediate benefits:

1. The task assignments made by the scheduler of the topmost controller are less binding. The responsibility to distribute tasks among the clients shifts to the subnet schedulers.

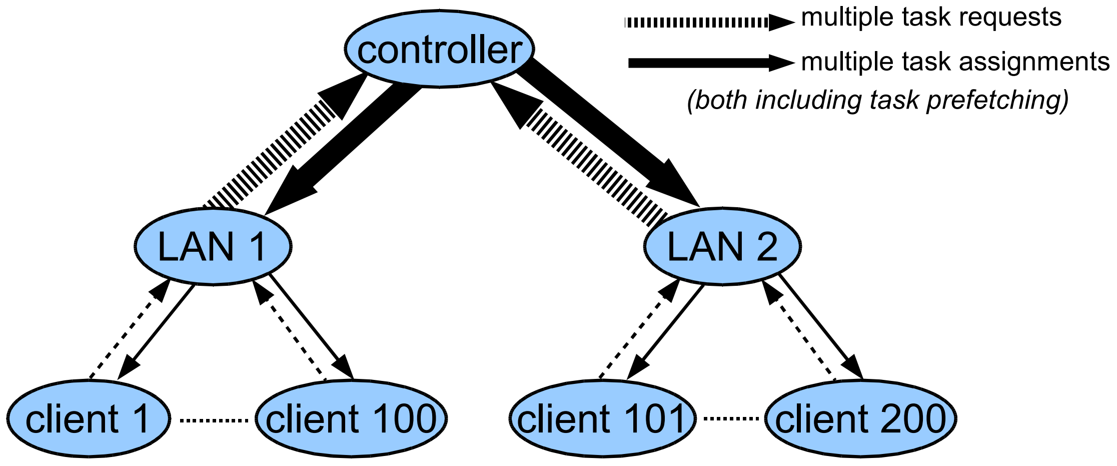


Figure 24: Updated task assignment scheme: In contrast to Figure 23, where all subnets only request tasks on demand, both LAN 1 and LAN 2 will prefetch multiple tasks to improve the quality of the task assignment of their children subnets.

2. When the first task request from the client arrives, the subnet scheduler has a pool of several tasks from which it can select the best one for the requesting client.

Comparison with two-stage task scheduling Note that there is a great similarity to two-stage task scheduling, discussed in section 3.3.1, where the controller assigned tasks to clients that again distributed the tasks among their available processors. For a visual comparison, consider Figure 18 and Figure 19 (both on page 41).

In fact, the idea of this section is to create an abstraction of the client internal shared memory parallelism, while the subnets are the abstraction of multiple processors within the clients. Thus, the requirements for the task scheduler remain the same. It still has to optimize task assignments with regard to locality and parallelism (as discussed in the previous section). In particular, it is not necessary to distinguish between task schedulers from different layers (i.e., the same scheduling algorithm can be used).

Nevertheless, there is one difference between client internal parallelism and support for hierarchical networks that has to be considered when using both concepts: Instead of dealing with clients that internally use several CPUs (of the same machine), the generalized concept deals with subnets consisting of several clients running on different machines. As a consequence, communication cannot be implicitly done via shared memory, but messages have to be sent. To efficiently support client internal parallelism, the final layer of task schedulers (connecting the clients and their CPUs) should therefore be specialized.

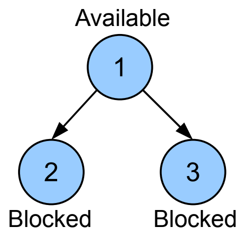


Figure 25: Scenario: Two idle clients **A** and **B**, but only one available task. If task duplication is allowed, task ① can be assigned to both clients.

3.4 Outlook: Task duplication

The idea of task duplication (or redundant task assignment) is to allow that one single task can be assigned not only to one client but to multiple clients. If there are enough idle clients available, task duplication might improve the performance by reducing the number of P2P communications.

Example 9. Let us consider the scenario shown in Figure 25. There are two clients **A** and **B**, which are both idle, but there is only one task available. With task duplication, both clients can execute task ① simultaneously. Let us assume that client **A** finishes task ① and continues with task ②. Meanwhile, client **B** has also finished task ①, and starts to execute task ③. Note that all three tasks could be executed without P2P communication between both clients.

In contrast, if task duplication is not allowed, P2P communication is required. For example, if task ① is assigned to client **A**, client **B** has to wait. After task ① is finished, both clients can execute the remaining two tasks in parallel, but client **A** has to send P2P data to client **B** (containing the results of the execution of task ①).

However, there is a tradeoff between P2P and soaking communication that should not be neglected. In the last example, we only considered P2P communication, but ignored soaking communication. Although task duplication saved one P2P communication, it also increased soaking communication, because the controller had to send the soaking data of task ① to both clients.

3.4.1 Avoiding race conditions

Although task duplication is a promising extension, the realization is not trivial. If a task is executed more than once, it can create race conditions between the task execution and the generation of P2P data, as illustrated in the following example:

Example 10. The scenario, shown in Figure 26, consists of two tasks and two clients **A** and **B**. There is only one variable X, which is undefined at first. Task ① initializes X to 0. The value of X is required by task ②, which increases X to 1.

We will stop the execution, after client **A** has finished task ①. To demonstrate the race condition, we will assume that each of the following actions will be executed atomically:

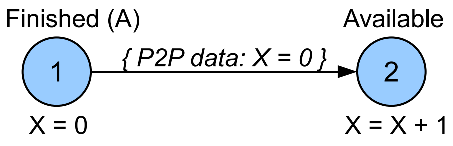


Figure 26: Scenario: Two clients **A** and **B**. Client **A** has finished task ①, which initialized the variable X with 0.

1. The controller assigns task ② to client **A**.
2. The controller assigns task ② to client **B** (using task duplication). During the assignment, the controller sends client **A** a P2P data request.
3. Client **A** executes task ② (thus, overwriting X with 1).
4. Client **A** receives the P2P data request to send client **B** a P2P data packet (between task ① and ②). This data packet contains the current value of X, which is 1.
5. Client **B** receives the P2P data packet, unpacks it, and executes task ②, therefore increasing X to 2.

   (In order words, the computation within task ② has been executed twice.)

To solve this problem, the protocol has to be changed. Unfortunately, it is difficult to support task duplication efficiently without completely redesigning the protocol. However, as a redesign goes beyond the scope of this discussion, we will only point out two solutions to avoid the race condition in a straightforward way:

Solution 1: Delay the execution of all tasks that could overwrite important data until it is safe because all concerned P2P data packets have been created:

- Add a synchronization phase to assure that a specific task has been finished: If a client has finished the execution of a task, it informs the task scheduler and blocks the execution of all tasks that could overwrite the results. These are all tasks that are directly reachable by the finished task's outgoing edges.
- Meanwhile, the task scheduler will wait until all clients to whom the task has been assigned, have also send the task termination message. Only then, it can send the unblock message.

However, solution 1 has major disadvantages, which renders it impractical:

- The extended protocol adds significant overhead. Especially, all clients have to wait for the slowest.
- The synchronization phase severely limits scalability.

Solution 2: Create all P2P packets immediately after the task's execution, before another task can overwrite the data:

- After a client has finished the execution of a task, it has to generate all P2P data packets for all outgoing edges (i.e., all tasks that directly depend on the finished task).
- When the task scheduler decides that a task will not be assigned any more, the P2P data packets for all of the task's incoming edges can be freed[11]. That means each of these P2P data packets has to be stored on the client until the task scheduler informs the client by sending a message.

Evaluation of solution 2:

- No explicit synchronization is necessary.
- Although this solution is more efficient than solution 1, there is still a time- and space overhead:
  - P2P packets have to be generated, even if they are never sent (time overhead).
  - P2P packets have to be buffered for unspecified time (space overhead and reduced predictability).

  (In contrast, without task duplication, P2P packets are generated on demand. After generation, the data packets are immediately sent. After sending, the memory can be immediately freed.)

Remark. Our current prototype does not support task duplication, only the task scheduler provides experimental support. More precisely, it is possible to assign a task to multiple clients, but the protocol modifications, as discussed in this section, are not implemented. Unfortunately, the expected overhead of both solutions seems to exceed the expected performance gain, but further experiments are required to substantiate this claim.

3.4.2 Conclusions and outlook

Depending on the test example and the test environment, the task duplication based version (without synchronization) performed slightly better or slightly worse, both in the range of about 20 percent. The significance of these first results is low, however, as the overhead of the synchronization techniques discussed in the last section is not considered. Nevertheless, there are two main reasons why task duplication might be of interest for future developments (both topics will be discussed in the course of this section):

- The scheduling heuristic that we used to decide whether a task should be duplicated can be further optimized.
- The overhead of the synchronization techniques can be significantly reduced if we allow greater modifications of the protocol.

[11] As finished tasks cannot be assigned, eventually, all P2P data packets will be freed.

When to duplicate? Task duplication is a dangerous optimization, as it can hurt the performance if used carelessly. It also adds complexity to the task scheduler, because there are several interacting factors, which have to be considered while deciding to duplicate a task, for example:

- How many idle tasks are available (to the requesting client)?
- How long will it take to execute a task on a specific client[12]?
- How long will it take to communicate the required data?
- How many other clients are idle (at the time of the task assignment)?

As not all of these information are available, even at runtime, new heuristics are essential. Most related work focuses on static task scheduling for multiprocessors [30, 28], where most of the relevant information are available at compile time. However, recent work also addressed dynamic task scheduling for Grid computation [17, 29, 39, 3], but without considering dependences between tasks nor task duplication.

In our experiments, we used a simple task duplication strategy: Whenever an idle client requests a tasks, task duplication is used if there are no idle tasks lefts (i.e., all available tasks are assigned to other clients). In other words, task requests will never block, but instead an already assigned task will be duplicated. The same scheduling algorithm that is used to select the next task among the set of available tasks (as described in appendix B) can be used to select the task that will be duplicated.

The advantage of this simple strategy is that it minimizes the client's CPU idle time. However, it is not optimal, as the execution of future tasks can be delayed because of the resulting lack of idle clients, as shown in the next example:

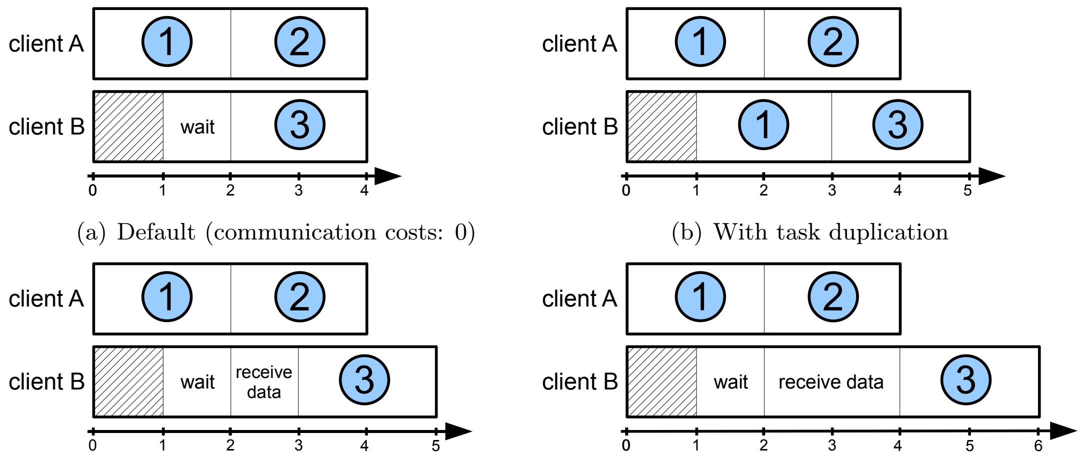
Example 11. Let us again consider the task graph shown in Figure 25 (on page 48). In this example, we will assume that there are two clients **A** and **B**. Let us further assume that it takes two periods to compute one task, independent of the specific task and the client. At the start of the example, only client **A** is idle, while client **B** will become idle after one period.

Without task duplication, client **B** will wait for one period until task ① has terminated. Then, both clients can compute tasks ② and ③ simultaneously. Thus, the overall execution will take four periods (Figure 27(a)).

The alternative is to use task duplication (Figure 27(b)). Instead of waiting for one period, client **B** can immediately start computing task ①. The drawback is that after task ① terminated on client **A**, client **B** is still busy, so it is not possible to start the computation of both available tasks ② and ③ immediately. As a result, the overall computation time increases from four to five periods.

In this simplified example, communication costs were neglected. However, under the assumption that the communication between client **A** and **B** takes one period,

[12] Note that in desktop Grids environments the task processing times on a specific client may increase unpredictably because of other user's processes running on the client's machine.



(a) Default (communication costs: 0) (b) With task duplication

(c) Default (communication costs: 1) (d) Default (communication costs: 2)

Figure 27: Task scheduling in example 11.

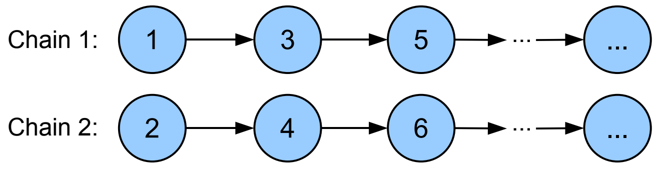


Figure 28: Two long, independent task chains.

the overall computation will take five periods in both cases (Figure 27(c) shows the version without task duplication). If the communication costs are increased to two periods, the task duplication version will outperform the classical version (five periods instead of six periods, as shown in Figure 27(d)). Note that using task duplication eliminates all communication in this example. Therefore, the task duplication based version does not suffer from an increasing penalty for communications.

Limited task stealing Another advantage of task duplication is that it allows techniques similar to task stealing, which we will call *limited task stealing*, as the task is not really stolen but only duplicated[13]. However, even in its limited form, task duplication can help to improve dynamic load balancing of already assigned tasks. A simplified application is illustrated in the next example:

Example 12. Figure 28 shows a task graph with two long, independent task chains. From computational perspective, both chains are equivalent. If there are two clients **A** and **B**, the task scheduler will assign each client one task chain to

[13] Supporting true task stealing would require to cancel "stolen" tasks, which is not possible without extending the protocol.

maximize local communications. Let us assume that client **A** will be assigned tasks with odd task IDs, and **B** will compute tasks with an even task ID.

Let us further assume that client **A** is running on a fast machine, but client **B** is running on a extremely slow machine (possibly overloaded by processes from other users). The faster client **A** will finish the computation of all its tasks while **B** is still busy. In this situation, the overall computation is delayed as the faster client has to wait for the slower client. By using task duplication, this worst case scenario can be avoided. Note that the overall computation is finished as soon as the controller has received all outstanding drain data. In particular, it is not required to wait for the termination of all duplicated tasks.

Although the last example might seem unrealistic, comparable situations are quite common at the begin and end of the computation. The reason is that many parallelized applications start with a phase of increasing parallelism and end with decreasing parallelism, similar to the graph task for the SOR1d example (see Figure 1 on page 8).

Compatibility with future developments In the last section, we stated that because of possible race conditions it is difficult to support task duplication without redesigning the protocol. So, how could a successful redesign look like?

Section 2.3.3 briefly discussed an alternative data flow architecture that is being developed outside this thesis to support client side memory reduction using Ehrhart polynomials. Based on this client architecture, solution 2 (page 50) can be implemented without significant overhead, because the data flow architecture already demands that all P2P data packets are generated immediately. Therefore, task duplication could be supported efficiently (at least in the data flow architecture).

3.5 Eliminating redundant dependences without communication

The task dependence graph consists of all relevant data dependences between the tasks. If there is additional information available about these dependences, it can be used to optimize the graph. In this section, we will distinguish between data dependences that are relevant for communication and others that are only needed to avoid data corruption. Finally, we will present a safe optimization to eliminate redundant dependences without communication.

Exact definitions of data dependences between array accesses can be found in the literature [22], but in the context of this thesis, we are more interested in dependences between tasks[14]. We will use the following informal definitions:

Definition 9 (Data dependence between two tasks)**.** There is a data dependence between two tasks if data written by one task is accessed by the other task.

[14] As tasks are executed sequentially, the framework can ignore all data dependences inside one task. However, the code generation still has to consider data dependences on the level of array accesses.

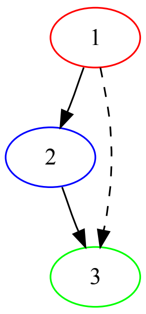


Figure 29: Dependences without communication are displayed by dashed arrows, where solid arrows denote dependences with communication. In this example, the dependence from task ① to ③ is redundant.

Data dependences in the original input program determine the order in which two conflicting tasks have to be executed. The first task be called *source task*, and it has to be executed before the *target task*.

Definition 10 (True dependence)**.** A data dependence is called *true dependence* if values written by the source task (but possibly overwritten by other tasks) are read by the target task.

If possible, we are more interested in a certain class of true dependence:

Definition 11 (Flow dependence)**.** A data dependence is called *flow dependence* if values written by the source task are directly needed by the target task.

Flow dependences are the most important type of dependence, in this context, because they describe how the intermediate results of the computation are propagated between tasks.

In contrast, the other types of data dependences only have to be respected for task scheduling, but they do not lead to communication:

Definition 12 (Output dependence)**.** A data dependence is called *output dependence* if values written by the source task are overwritten by the target task.

Definition 13 (Anti dependence)**.** A data dependence is called *anti dependence* if values needed by the source task are overwritten by the target task.

The only purpose of anti and output dependences is to delay the assignment of conflicting tasks. Thus, we can safely remove anti and output dependences if the remaining dependences already force the correct task execution order (as illustrated in Figure 29).

Remark. Removing redundant dependences is very similar to finding the *transitive reduction* of a graph [2], but with the addition that not all edges are equal, since edges representing true or flow dependences cannot be removed. Consequently, the problem of removing redundant dependences is at least as hard as computing the transitive reduction of an acyclic graph[15].

Example 13. Figure 30 shows the original task dependence graph for the SOR1d example. Most of its tasks have three incoming dependences.

The elimination of all redundant dependences yields the optimized task dependence graph shown in Figure 31. The maximum number of incoming dependences dropped from three to two, thus reducing the number of sent messages per task.

To understand why these dependences are redundant, we have to consider the detailed task dependence graph with all dependences (Figure 32).

Note that the detailed graph can contain multiple edges between the same two tasks (e.g., there is a flow and an anti dependence from task $(3, -2)$ to $(4, -3)$). This is no problem, because these edges can be unified into a single "merged" dependence, which causes communication if at least one of the unmerged dependences causes communication (e.g., the merged dependence from $(3, -2)$ to $(4, -3)$ causes communication because of the flow dependence).

Let us now take a closer look at the dependences in this example:

flow The flow dependences are drawn as solid lines. They cannot be removed from the graph, because they represent the communication between the tasks.

anti The set of anti dependences is identical to the set of flow dependences. Thus, they are virtually superfluous, because they will be merged with the flow dependences.

output These are all redundant dependences, because they do not force communication, and the execution order will not be changed if they are removed. For example, the output dependence from task $(4, -3)$ to $(6, -4)$ is redundant because of two flow dependences: The first from task $(4, -3)$ to $(5, -4)$, and the second from $(5, -4)$ to $(6, -4)$, thus $(4, -3)$ has to be executed before $(6, -4)$.

Although the saved messages are small, in fact, they are all empty, they introduce additional latency that cannot be avoided, even if task prefetching is used. According to our experiments, the overall execution time of the SOR1d decreased about 20 percent when all redundant dependences were eliminated.

Although removing the redundant dependences can increase the performance, there is a tradeoff: Computing the reduction takes time, thus if this time is not saved by a faster overall computation, the optimization is not effective.

Space efficiency is also important, especially for huge graphs with more than 100,000 nodes. Therefore, we used a heuristic to compute the reduced graph. We will skip the details, but the algorithm is described in appendix C.

Remark. The optimization discussed in this section is not enabled by default, but must be enabled with the HOC runtime option `--remove-transitive-dependences`. There are two reasons for this decision:

[15] The reduction is simple: Convert the acyclic graph to a task dependence graph where all edges are output dependences. Removing all redundant dependences yields the transitive reduction of the original graph.

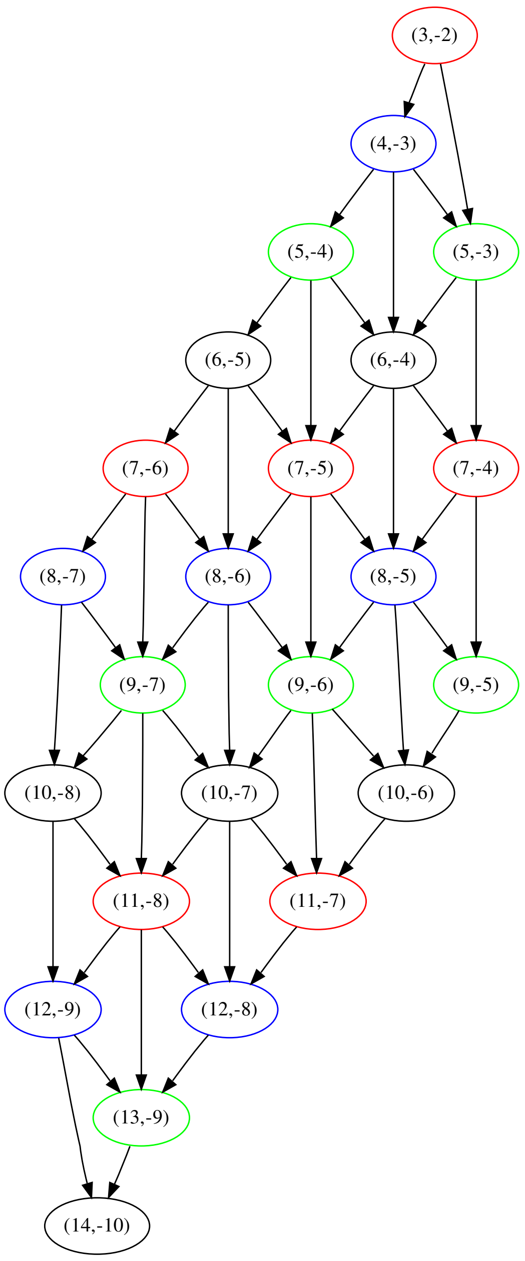


Figure 30: Original task dependence graph for the SOR1d example.

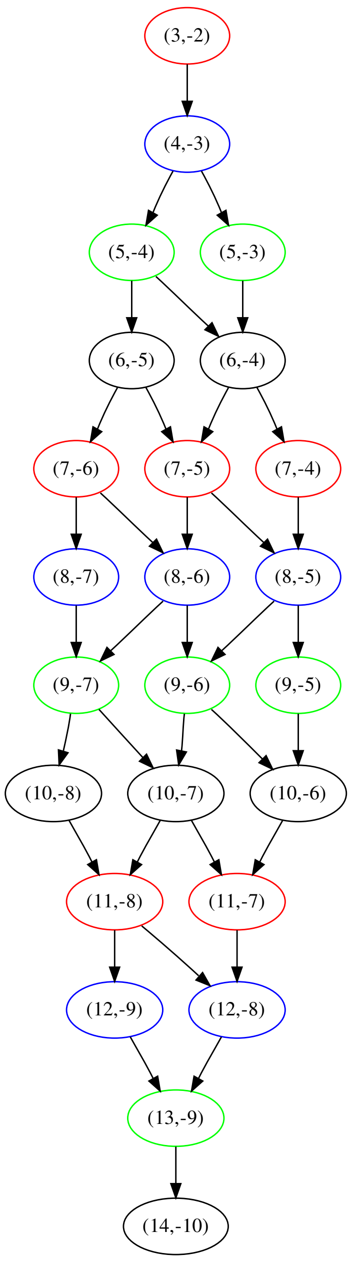


Figure 31: Optimized version of the task dependence graph shown in Figure 30.

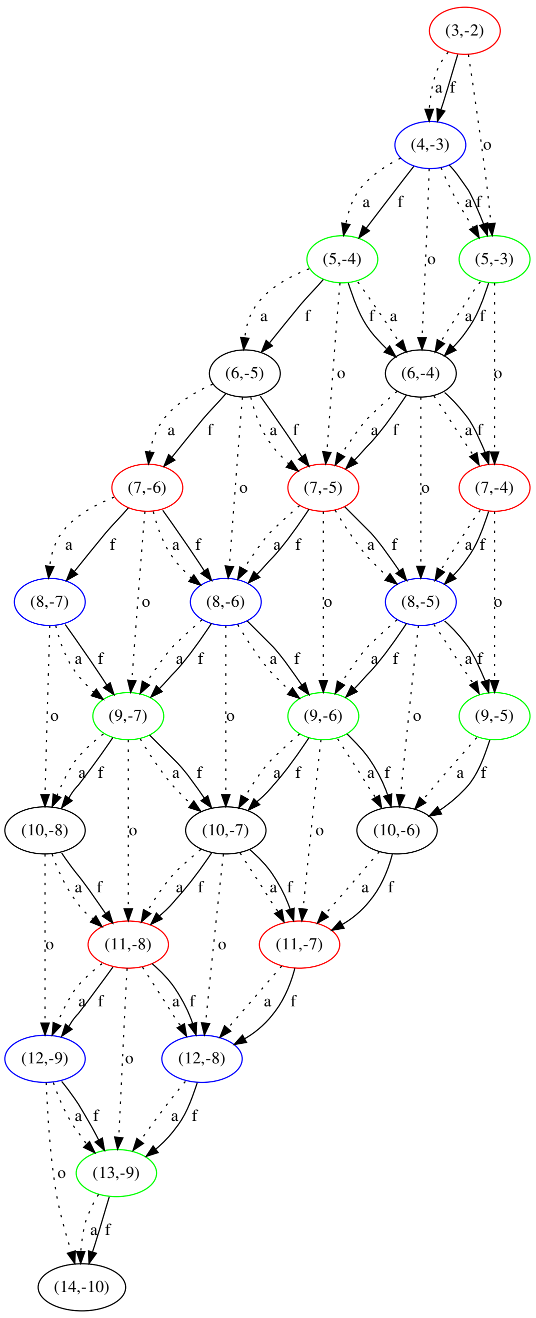


Figure 32: Detailed task dependence graph for the SOR1d example. Each dependence is labelled with its type (**f**=**f**low, **a**=**a**nti, **o**=**o**utput). Dependences without communication are displayed by dashed arrows.

- Depending on the user application and the quality of the dependence analysis, it is possible that there are no redundant dependences.
- The heuristic is potentially expensive, for example, its space complexity is $O(n^2)$, where n is the number of tasks.

Note that there is a simple alternative: The framework may skip the communication between two tasks if there are only anti or output dependences between these tasks. This effectively avoids the problem that empty messages are sent, but the task scheduler has to operate on a larger graph because the redundant dependences will not be removed.

4 Automatic generation of the HOC code

Although the subject of this thesis is not code generation, but rather the development of the P2P-HOC and the extension of the existing runtime framework, the topic is important, as the success of the P2P-HOC depends significantly on the quality of the parallel code. Thus, the P2P-HOC has been designed to support the automatic code generation as far as possible.

However, we will not discuss parts of the code generation that are neither related to the P2P-HOC nor to the extended framework. Instead we will refer to the literature, which describes both P2P-HOC generation [9] and master-worker HOC generation [4], because some parts of the master-worker HOC's code generator have been reused for the P2P-HOC. Internally, the code generator is based on automatic code generation for distributed memory architectures in the polytope model [10, 32, 22].

4.1 Format of the input file

Figure 33 shows the input file for the SOR1d example. It is standard Java code, which has to be runnable on its own when embedded in a simple wrapper class with a main function, as shown in Figure 34. Each input file consists of four parts, which are identified by special annotations (`// loopo ...`):

1. Variable and array declarations	(lines 1–4)
2. Initialization code	(lines 6–12)
3. Parallelizable loop	(lines 14–20)
4. Finalization code	(lines 22–23)

4.1.1 Loop annotation

The loop annotation contains declarations of all constants and array variables that are used inside the loop nest. Multiple declarations have to be separated by semicolons.

```
int m;
int n;
double[] A;
// loopo end declarations

// Allocate memory for the arrays
M = 500000;
N = 1000000;
A = new double[N+1];

// Initialize input array A
// ...

// loopo begin loop (constants: m, n; arrays: A{n+1})
for (int k = 1; k <= m; k++) {
  for (int i = 2; i <= n-1; i++) {
    A[i] = (A[i-1] + A[i+1]) / 2.0;
  }
}
// loopo end loop

// Print results
// ...
```

Figure 33: Input file for the SOR1d example. The commentaries beginning with *loopo begin* and *loopo end* are parsed as special annotations.

```
public class SeqWrapperClass {
  public static void main(String[] args) {

    // insert input file here
    // ...

  }
}
```

Figure 34: Simple wrapper class to execute input files sequentially.

```
// loopo begin loop (...)
for (...) {
  for (...) { S1; S2; }
  for (...) { S3; }
}
S4;
for (...) { S5; }
// loopo end loop
```

Figure 35: An example of a non-perfect loop nest with five statements.

- The **constants** declaration declares the loop's parameters. It is assumed that all parameters are of type `int`.

  Why are runtime parameters listed as constants? Because they are required to be constant during the execution of the loop, but they are not compile-time constants (e.g., the user application can read them from a file, or ask the user to enter them).

- The **arrays** declaration declares the loop's array variables, along with their sizes: `arrays [IN/OUT/IN-OUT]: name {dim1, ..., dimN}`

  The array sizes can be arbitrary Java expressions, as long as only variables and arrays are used that are defined in the loop annotation. It is assumed that all arrays are rectangular.

  Optionally, arrays can be declared as readonly (IN) or writeonly (OUT). Note that this attribute refers only to the usage of the array inside the loop. If no attribute is given, it is assumed that the array is both readable and writeable (IN–OUT).

In principle, it is possible to select more than one loop nest, but it would complicate the implementation considerably without much gain for experiments. Therefore, our prototype supports only one parallelizable loop nest at the moment. However, the loop nest is not restricted to a perfect loop as in the SOR1d example, for instance, Figure 35 shows a more complicated but legal example with multiple loops and statements.

4.1.2 Implementation dependent limitations

During the code generation, the initialization and finalization code is inserted directly into the intended position of the output file. More precisely, the initialization code is inserted at the begin of the controller's `startUp` function, whereas the finalization code fills the `end` function (see section 2.8.1). Consequently, the initialization and finalization code can be any legal Java code.

The declarations and the loop, however, will be further processed during the parallelization. Because of that, there are some implementation specific restrictions:

	controller side	client side
soaking	`writeSoakData`	`loadDrainData`
draining	`loadDrainData`	`writeDrainData`

Table 2: Overview of all P2P-HOC specific soaking and draining functions (for a reference of the P2P-HOC interface, see section 2.8).

- Declarations have to be in the following format:
  - Exactly one declaration per line
  - No declarations with initialization (e.g., not "`int n = 1;`")
  - All arrays that are used inside the loop have to be of type `double`[16]
- The loop is parsed by LooPo, which introduces some minor restrictions because of LooPo's parser (e.g., no support for some operators like `+=` or `++`)

4.2 Soak/drain statements

In the original master-worker HOC, all data is stored centralized on the master. In contrast, the P2P-HOC requires soaking and draining. In this section, we will discuss some aspects of the generation of soaking and draining code, but only as far as it concerns the framework or the annotations of the input file.

To generate the soaking and draining code, special statements, called soak and drain statements, are added to the input file before the data dependence analysis is started. Their only purpose is to force additional data dependences, which will be used to generate the soaking and draining code. As the additional statements are self-assignments, they can be safely removed from the target code at the end of the code generation. Finally, the generated code is used to implement the P2P-HOC functions shown in Table 2.

There are two simple optimizations, which will improve the performance of the generated code that is responsible for soaking and draining:

- Soaking can safely ignore all arrays that are marked as output-only (OUT), because the initial values of these arrays are not needed for the computation.
- Draining can safely ignore all arrays that are marked as input-only (IN), because these arrays will not be modified during the computation.

Note that all arrays marked as IN–OUT, which is the default, must be considered for both soaking and draining. That is why it will improve performance if the user provides this information, because our prototype implementation cannot detect it automatically. Future work might examine possibilities to deduce the information from the results of the dependence analysis.

[16] The limitation to multidimensional `double` arrays has nothing to do with the parser of the input file, but with the communication code's internal buffer management. It should be unproblematic to extend the implementation to support other primitive types as well. Note that the P2P-HOC's interface does not contain assumptions about the array type (see section 2.8).

```
// loopo begin loop (constants: n;
arrays [IN]: A{n,n}, B{n,n}; arrays [OUT]: C{n,n})
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
// loopo end loop
```

Figure 36: Annotated loop for the matrix multiplication example. The complete input file is shown in Figure 44 in appendix A.

Example 14. Figure 36 shows the annotated loop for a matrix multiplication algorithm[17]. There are two input arrays A and B, and one output array C. The input arrays are initialized on the controller during the execution of the user's initialization code, but are not available on the clients at the begin of the computation. Thus, the soaking data consists of the arrays A and B, and the draining data consists of array C, which is the result of the computation.

Figure 37 shows the modified input file with soak and drain statements. The array sizes given in the loop annotation define the surrounding loops of the inserted statements. In this example, all arrays are two-dimensional and of size $n \times n$.

In the target code, the self assignments will be removed, but the additional dependences define the soaking and draining communication:

- Soak communication is defined by the dependences from both soak statements (for the arrays A and B) to the loop statement.
- Drain communication is defined by the dependences from the loop statement to the drain statement (for the array C).

4.2.1 Problem: Repeated soaking communication

It is possible that the same soak data will be communicated repeatedly, as illustrated in the following example.

Example 15. Consider the code from Figure 38. It is a simplified example, as it shows only the soaking statement for array A, which contains only one constant value. This value $A[0]$ is used to scale the second array B, but it is not modified.

[17] The input program relies on the Java property that all allocated arrays, which are of type `double`, are automatically initialized with 0.

```
// soak statement for input array A
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; i++) {
        A[i][j] = A[i][j];
    }
}
// soak statement for input array B
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; i++) {
        B[i][j] = B[i][j];
    }
}
// original loop
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
// drain statement for output array C
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; i++) {
        C[i][j] = C[i][j];
    }
}
```

Figure 37: Modified input file with soak and drain statements.

```
// (simplified) soak statement S (for array A)
A[0] = A[0];

// scale array B with A[0]
for (int i = 0; i < n; i++) {
  B[i] = B[i] * A[0]; // computation statement T
}
```

Figure 38: Example with a soak statement S and a computation statement T.

There is a flow dependence from the soak statement S to the computation statement T for each instance of the loop[18]:

$$\langle (), S\rangle \longrightarrow \langle (i), T\rangle \text{ for } 1 \leq i \leq n$$

Even though $A[0]$ is never modified, it will be communicated multiple times, because these flow dependences will be mapped directly to soak communication.

Although this problem does not harm correctness, it may create significant communication overhead. Note that if tasks are assigned to different clients, it is necessary to send $A[0]$ once per client, but subsequent communications to the same client are superfluous.

One drastic workaround is to simply broadcast all soaking data before the computation. This approach is discussed in the next section, which deals with a similar problem that is caused by an imprecise dependence analysis.

4.3 Dealing with an imprecise dependence analysis

The task of the dependence analysis is to find all data dependences that are in the input program. It is allowed that the analysis returns additional dependences, that means it can be pessimistic, but it must never miss existing dependences. In the context of this section, we call a dependence analysis *imprecise* if it returns indirect true dependences, which are true dependences but not flow dependences.

It is important to note at this point that imprecision does not imply that the analysis is incorrect. In fact, if the input program contains conditional constructs (e.g., `if-then-else`), it is generally unavoidable that the analysis will be imprecise. In other words, it is only possible to offer a analysis with full precision for static control programs.

In contrast to flow dependences, indirect true dependences are problematic, because there is an in-between write operation, which overwrites the data that has been written by the original source. Therefore, the communication, which is represented by this non-optimized dependence, will transfer old data. Beside the communication overhead, this can even lead to an incorrect execution if the receiver client overwrites new data located on its computer, as illustrated in the following example:

Example 16. In our simplified scenario, shown in Figure 39, there is only one client, and only one array A of size one, which is initialized (on the controller) with 0. There are three simple tasks: ① → ② → ③. Each task increments $A[0]$ by one. Thus, the final value of $A[0]$ should be 3.

The intended execution of this example is shown in Table 3. Each line describes the execution of one task:

1. At the begin, $A[0]$ is undefined on the client, therefore the controller sends a soak packet, which sets $A[0]$ to 0.

[18] Statement S is not surrounded by a loop, therefore its iteration vector is empty.

```
// input: A[0] = 0
// loopo begin loop (arrays: A{1})
A[0] = A[0] + 1; // task 1
A[0] = A[0] + 1; // task 2
A[0] = A[0] + 1; // task 3
// loopo end loop
// output: A[0] = 3
```

Figure 39: Simplified input file to illustrate the problems that are caused by an imprecise dependence analysis.

Task ID	A[0] (initial)	soak data	A[0] (soak data loaded)	P2P data	A[0] (P2P data loaded)	A[0] (after execution)
①	$\perp$	0	0	–	0	1
②	1	–	1	optimized	1	2
③	2	–	2	optimized	2	3

Table 3: Optimum situation: Only soak data at the begin.

As task ① does not depend on other tasks, no P2P data packets are required, so $A[0]$ remains 0.

After the execution of the computation code, $A[0]$ is incremented to 1.

2. No soak data is required, so $A[0]$ remains 1.

   The P2P data packet that communicates data from task ① to ② can be eliminated, because the communication is local (both tasks are executed on the same client).

   After the execution of the computation code, $A[0]$ is incremented to 2.

3. Analog to task ②: No soak or P2P data. $A[0]$ is incremented to 3.

Note that the only communication is the soak communication at the begin.

Now let us consider the problematic situation, where — as a result of the imprecise dependence analysis — soak communication is attached to all tasks. Figure 40 shows the loop code after soak and drain statements were added. There is one flow dependence d_1 from the soak statement S to the first task T_1. This dependence is used to generate the soaking code for the first task.

In contrast, the other two dependences d_2 and d_3 are not flow dependences but indirect true dependences. Therefore, d_2 and d_3 should be optimized away by a precise dependence analysis. Otherwise, in case of an imprecise analysis, soaking code will be generated for all three tasks. This results in the execution shown in Table 4:

The execution of task ① is unchanged, but when task ② is executed, the controller sends an obsolete soak data packet (containing the old value of $A[0]$ on the controller).

$$\begin{array}{rl} S: & \mathrm{A}[0] = \mathrm{A}[0] \\ \hline T_1: & \mathrm{A}[0] = \mathrm{A}[0] + 1 \\ T_2: & \mathrm{A}[0] = \mathrm{A}[0] + 1 \\ T_3: & \mathrm{A}[0] = \mathrm{A}[0] + 1 \\ \hline D: & \mathrm{A}[0] = \mathrm{A}[0] \end{array}$$

$$\begin{array}{l} d_1 : \langle (), S \rangle \longrightarrow \langle (), T_1 \rangle \\ d_2 : \langle (), S \rangle \longrightarrow \langle (), T_2 \rangle \\ d_3 : \langle (), S \rangle \longrightarrow \langle (), T_3 \rangle \end{array}$$

Figure 40: Left: Loop code with soak statement S and drain statement D.
Right: Dependences from the soak statement to the computation statements T_1, T_2 and T_3. Only d_1 is of type *flow*, but d_2 and d_3 are indirect *true* dependences.

Task ID	A[0] (initial)	soak data	A[0] (soak data loaded)	P2P data	A[0] (P2P data loaded)	A[0] (after execution)
①	$\perp$	0	0	–	0	1
②	1	0	0	optimized	0	1
③	1	0	0	optimized	0	1

Table 4: Problematic situation: Soak data overwrites local data. P2P local data optimization is no longer possible.

When this soak packets is unpacked, it overwrites the current value of $A[0]$ on the client with 0. After the task ②'s computation code, $A[0]$ is 1, but it should be 2 instead. The same problem occurs when task ③ is executed. Consequently, $A[0]$ remains 1.

Though the ideal solution is to use a precise dependence analysis, but if that is not possible, alternatives have to be found. The first solution repairs the damage at the framework level:

Solution 1. Disable local data optimization, but always send P2P packets, even if sender and receiver are identical. Thus, soaking will still overwrite existing data, but after unpacking the P2P packets, the lost data will be restored.

The modified execution is shown in Table 5. The soaking communication in task ② and ③ still overwrites $A[0]$, but after unpacking the P2P data, the value of $A[0]$ is restored.

Task ID	A[0] (initial)	soak data	A[0] (soak data loaded)	P2P data	A[0] (P2P data loaded)	A[0] (after execution)
①	$\perp$	0	0	–	0	1
②	1	0	0	1	1	2
③	2	0	0	2	2	3

Table 5: Solution 1: Disable P2P local data optimization.

Task ID	A[0] (initial)	soak data	A[0] (soak data loaded)	P2P data	A[0] (P2P data loaded)	A[0] (after execution)
①	0	–	0	–	0	1
②	1	–	1	optimized	1	2
③	2	–	2	optimized	2	3

Table 6: Solution 2: The broadcasted `UserClient` contains all arrays, which are not marked as writeonly (OUT). Therefore, no soak data packets are needed.

Note that the P2P data would not have been sent if the local data optimization had been enabled.

In contrast, the second solution radically changes the soaking strategy of the generated code by moving all potential soaking communication before the computation phase, thus eliminating all soaking communication during the computation.

Solution 2. Attach all arrays that are relevant for soaking (i.e., which are used inside the loop nest, but are not marked as writeonly) to the `UserClient` object that is broadcasted by the controller to all clients[19]. Therefore, no soaking communication during the computation phase is required.

The modified execution is shown in Table 6. As all soaking communication has been moved to the begin, the array A is fully initialized on the client. Before the first task is executed, $A[0]$ is 0, whereas in the other examples it is still undefined. Note that no data is overwritten, because of the lack of soak communication during the computation.

Although the second solution does not require changes to the framework, it assumes that the broadcasting of the client is implemented in an efficient, scalable way. In our implementation, we used a binary tree algorithm, but more sophisticated algorithms can be found in the literature [40].

Let us compare the advantages and disadvantages of both solutions:

- Solution 1: Turn off local data optimization
  - + Faster startup times.
  - ± Potentially less communication than in solution 2, but only if the overhead created by the additional dependences does not exceed the communication costs of the initial broadcast.
  - − Local communication is inexpensive, as it does not require that data is sent over the network, but there is still a significant overhead because of the generation and unpacking of the P2P data packet. Note that this overhead would have been eliminated by the local data optimization.
  - − It is incompatible with improved task prefetching, as described in section 3.1.2.

[19] Normally, a half-initialized `UserClient` object is broadcasted, which only contains the values of the runtime parameters. The arrays are later allocated by a call to the `init` method (see section 2.8.2), but soaking communication is required to send the content of the arrays.

Test example	Input file	Problem sizes
SOR1d	Figure 33 (see section 4.1)	$M = 500,000, N = 1,000,000$
Matrix multiplication	Figure 44 (see appendix A)	$n = 4,000$
Polynomial product	Figure 45 (see appendix A)	$n = 500,000$

Table 7: Overview of the test examples.

- Solution 2: Broadcast soak data before the computation
  - + During the computation, no soak communication is necessary, which speeds up the computation phase. It also reduces the controller's workload, thus improving scalability (at least during the computation phase).
  - + Guarantees that soak data is only send once, thus avoiding the problem with repeated soak communication, which is discussed in example 15 in section 4.2.1.
  - − Expensive startup phase. Especially, it is not suited for low-bandwidth networks.
  - − It is incompatible with client side memory reduction (see section 2.3.3), which is an important future goal. The reason is that solution 2 forces each client to allocate enough memory to hold all broadcasted arrays.

5 Experiments

Our test environment consists of about 100 desktop PCs (ranging from AMD Athlon XP 3000+ to Intel Core 2 Duo 6400), connected by a 100 MBit Ethernet network. However, we could not use more than 80 of them, at the same time, because not all computers are online all the time. Since the computers that we used are part of the CIP-Pool of the university of Passau, they were not exclusively reserved for our tests but were available to all students and members of staff. Therefore, dynamic load balancing could be tested under realistic conditions.

Table 7 contains the test examples with references to their input files. Our experiments show that the P2P version (Figure 41) is faster than the master-worker version (Figure 42) in all examples we have tested.

Remark. The diagrams contain super linear speedups, for example, in the P2P-HOC, the parallel SOR1d running on 16 computers is more than 20 times faster than the sequential version running on a single computer. How is this possible?

Although it may seem surprising at first, it has to do with cache effects: Tiling is applied on the input program during the parallelization, thus the cache efficiency of the target code is also improved. We compared the speedup against the original sequential input programs, but not against a hand-optimized cache-optimal version[20]. However, it is important to note that our conclusions about scalability are not concerned.

[20] For example, a blocked algorithm could be used to implement the matrix multiplication.

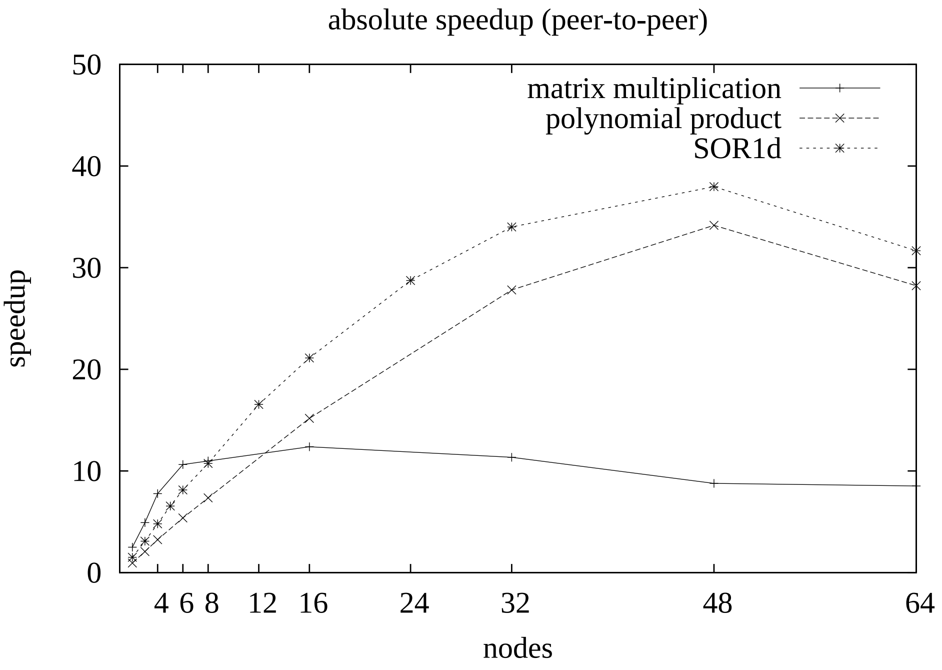


Figure 41: Absolute speedups in the P2P framework.

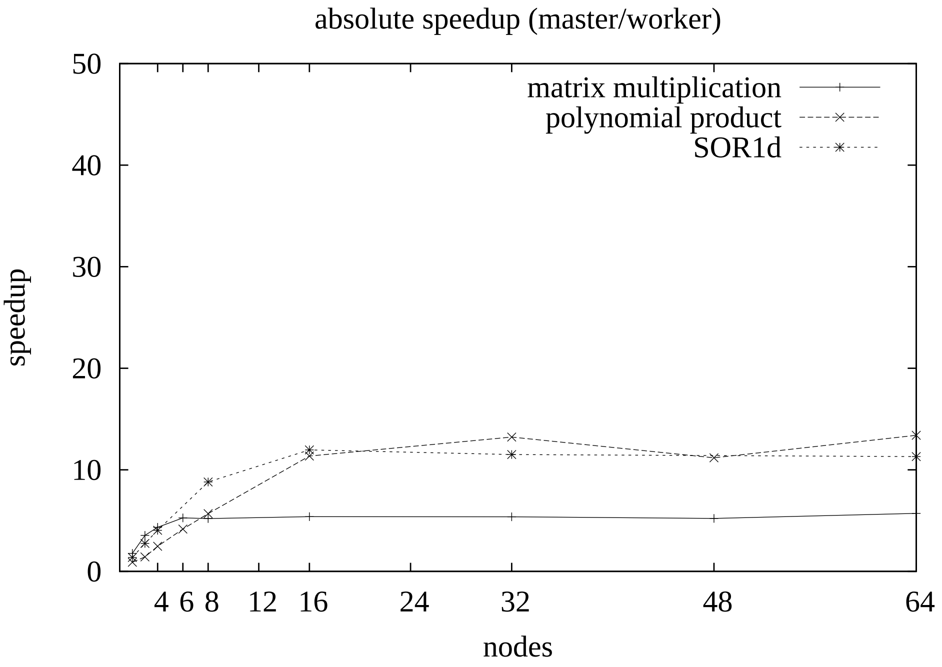


Figure 42: Absolute speedups in the master-worker framework.

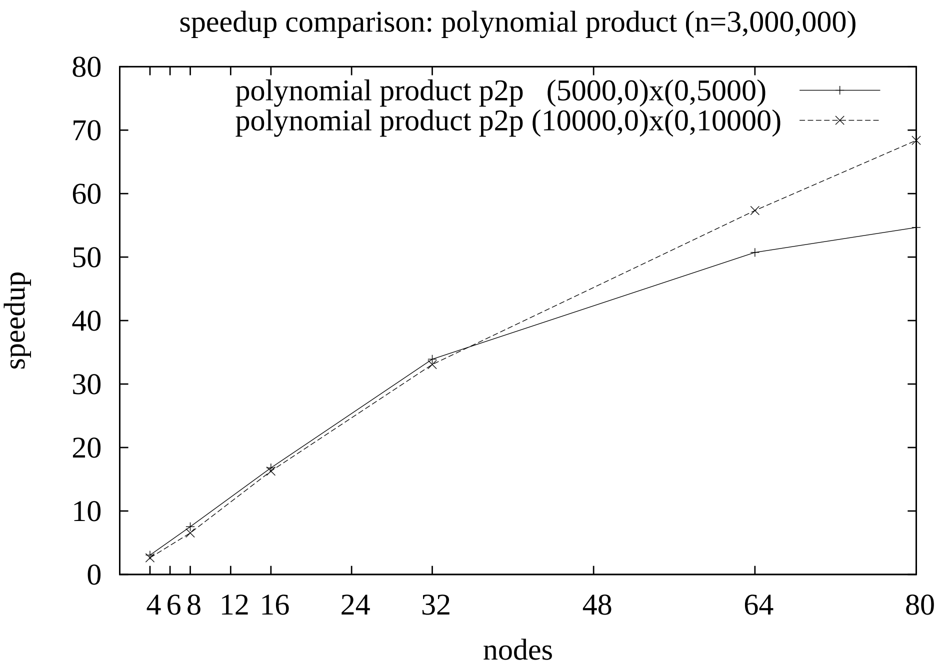


Figure 43: Speedup for the polynomial product example for two different tiling vectors (both using the P2P-HOC). The problem size n has been increased from $500,000$ to $3,000,000$.

The scalability also improves with increasing problem sizes. Unfortunately, as the memory requirements also scale with the problem size, we could not increase the problem sizes for the SOR1d and matrix multiplication example without running out of memory. However, Figure 43 shows the improved scalability for the polynomial product example, where the problem size n was increased from $500,000$ to $3,000,000$. Depending on the choices for the tiling vectors, the example scales almost linearly to 80 computers, which is also the maximum number of computers that we could use for our tests.

Our experiments also proved that the communication volume is reduced because of the the P2P extensions: The total network traffic on the controller (P2P-HOC) is about 3–10 times less than on the master (master-worker HOC). Therefore, we expect that the P2P version performs better than the master-worker version if the bandwidth is limited (i.e., if the available bandwidth is lower than the communication volume of the user application). As the communication volume increases with the problem size as well as with the number of processors, this also affects scalability in environments with limited bandwidth.

Our original framework suffered from latency problems, which disappeared when we replaced the code responsible for message passing with a more efficient implementation using non-blocking synchronization. Not surprisingly, the performance did improve, but more interesting is the observation that it helped the P2P-HOC significantly more than the master-worker HOC.

One plausible explanation is that in the master-worker HOC it is always possible to send data immediately from the source to the target, whereas in the P2P-HOC the P2P data communication requires one indirection, because the controller has to send one additional message to inform the client that should sent the data. Therefore, we conclude that the master-worker approach is more robust regarding high latency (assuming that the bandwidth is unlimited). Note that task prefetching helps to reduce latency problems, but it cannot completely eliminate them.

6 Conclusions and future work

In this thesis, we replaced the existing master-worker HOC by the new P2P-HOC, which allows communications between workers and lifts the performance bottleneck that all data has to be stored on the master, even during the computation. Thus, the central differences between the master-worker HOC and the P2P-HOC concern the communication scheme:

- In the master-worker HOC, the communication scheme is invariant:
  - Before the task execution, the master sends all data that will be read by the worker.
  - After the task execution, the worker sends back all data that has been written.
- In contrast, the P2P-HOC uses soak and drain communication between controller and clients, but most of the data is transferred via P2P communication (between clients).

As a consequence, the P2P-HOC's communication scheme has more potential for optimizations, because local P2P communications can be eliminated (*local data optimization*). Therefore, the task scheduler can reduce the amount of sent data by maximizing local communications. Additionally, the P2P-HOC turns out to be more flexible regarding task prefetching, which is an important optimization on its own, also used by the master-worker HOC. Based on local data optimization, we introduced the concept of *improved task prefetching*, which improves both prefetching and task scheduling by increasing the number of available tasks (for the requesting client).

The original claim that a centralized peer-to-peer architecture (as used by the P2P-HOC) is superior to the existing master-worker architecture with regard to performance and scalability was proved by the benchmarks. In all tested examples, the P2P-HOC outperformed the master-worker HOC.

Let us end with an outlook of future goals:

- In section 3, we briefly discussed some powerful task scheduling optimizations (client internal parallelism, support for hierarchical networks, and task duplication), but could not support the theory with experiments, so there is much space for further research.

- Section 4 deals with runtime workarounds for compile time problems closely related to dependence analysis. As both workarounds are painful, fixing the issues at compile time would be a major improvement.
- Currently, both the controller and the clients are single points of failure. Solving this problem for the controller seems difficult (because of the centralized architecture), but *checkpointing* should be sufficient (i.e., regularly storing the state of the application's memory and the task graph to recover from failures).

  On the client side, redundancy could be added by using task duplication to improve the system stability. Additionally, the restriction that all clients have to register at the controller before the computation could be lifted, thus supporting *CPU scavenging*.

Naturally, it is a matter of taste to decide which goals are most valuable, but let us point out two topics that are most urgent from the aspect of a typical end user.

Automatic tile size determination Currently, the tiling vectors have to be determined by the user, which is not trivial, because of the following reasons:

1. The *dimension* of the tiling vectors have to respect the loop dimensions of the input programs. Otherwise, the code generation will fail.
2. The *size* (scaling factor) of the tiling vectors: Smaller tiling vectors will increase the number of tasks, thus leading to more (fine grained) parallelism, but at the cost of more communication.
3. The *form* (direction) and the *shape* (ratio of the vector's lengths) of the tiling vectors affect the communication volume. Ideally, the tiling vectors should be chosen with respect to the direction of the dependences.

While the user might find a tiling vector that meets the first demand by trial and error, the other demands are highly problematic. Especially, determining the form and shape is difficult, because it requires insights into automatic parallelization techniques and knowledge about the dependences in the user application. This is contradictory to the original goal that the typical end user should be able to use the parallelization framework without expert knowledge. Unfortunately, at the current state, we cannot even provide simple guidelines to find good tiling vectors.

Note that dynamic techniques are not applicable yet, that means the tiling vectors have to be determined at compile time, before the computation. Recent work, however, has shown that, in principle, this restriction can be lifted using a technique called quantifier elimination [23, 24, 25, 11], which would allow us to postpone the determine of the tile size until runtime (using parametric tiling). The advantage would be that, at this stage, there is more information available (e.g., the number of processors, the values of the runtime parameters).

Meanwhile, a valuable interim goal might be to support the user with a good heuristic, for example, where the user has only to estimate the number of available processors and the expected problem sizes (i.e., the values of the runtime parameters).

Client side memory reduction It is difficult to achieve scalability with small problem sizes. For example, according to our experiments, the matrix multiplication example scaled only to about six computers. Does that imply that matrix multiplication is not parallelizable? Of course not, but we could only test $4,000 \times 4,000$ matrices, otherwise we would ran out of memory. These are relatively small matrices that can be multiplied on a single computer in reasonable time. However, to justify the increased complexity when switching from a single computer to a Grid environment, the application has to be extremely challenging. For example, it more likely that a typical Grid user will want to multiply matrices of size $1,000,000 \times 1,000,000$.

In these settings, efficient memory management is crucial, which additionally complicates the development of the parallelized application, but at the same time increases the need for automated code generation. Thus, future work might focus on the support of out-of-core algorithms by reducing the amount of memory that has to be allocated on the clients (as discussed in section 2.3.3).

```
int n;
double[][] A;
double[][] B;
double[][] C;
// loopo end declarations

// Allocate memory for the arrays
n = 4000;
A = new double[n][n];
B = new double[n][n];
C = new double[n][n];

// Initialize input arrays A and B
// ...

// loopo begin loop (constants: n; arrays [IN]: A{n,n},
    B{n,n}; arrays [OUT]: C{n,n})
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
// loopo end loop

// Print results
// ...
```

Figure 44: Input file for the matrix multiplication example.

A Input files for the examples

The two missing examples are shown in Figure 44 (matrix multiplication) and Figure 45 (polynomial product). The input file for the SOR1d example will not be replicated here, as it already occurred in Figure 33 in section 4.1.

B Algorithm: Task scheduling

Remark. Although our task scheduler supports task duplication, we will skip it in this section, because it is not supported by the rest of the framework.

```
int N;
double[] A;
double[] B;
double[] C;
// loopo end declarations

// Allocate memory for the arrays
n = 500000;
A = new double[n+1];
B = new double[n+1];
C = new double[2*n+1];

// Array initialization
// ...

// loopo begin loop (constants: n; arrays [IN]: A{n+1},
    B{n+1}; arrays [OUT]: C{2*n+1})
for (int i = 0; i <= n; i++) {
  for (int j = 0; j <= n; j++) {
     C[i+j] = C[i+j] + A[i] * B[j];
  }
}
// loopo end loop

// Print results
// ...
```

Figure 45: Input file for the polynomial product example.

Algorithm sketch: List all tasks that are available to the requesting client **A**. Assuming that redundant task assignment is not allowed, we can discard all assigned tasks. The remaining tasks are each evaluated with regard to the following criterions:

Primary goal: Minimize communication by preferring local communication:

- Slightly penalize tasks without incoming dependences.
  (Note: This sounds counterintuitive, but these tasks are too valuable because they are optimal for all clients, not only for client **A**.)
- If there are incoming dependences, however, minimize the number of remote communications:
  - Communication is local if the source task has been finished by client **A** or has been assigned to it. Otherwise, it is remote.
  - Local communication is slightly good, remote communication is bad.
    (If all communications are local, the task should be preferred to another task without incoming dependences. However, if at least one communication is not local, a task without incoming dependences is still preferable.)
  - Always prefer a *task chain*, which is a task that matches the following conditions:
    1. Exactly one incoming dependence (i.e., there is only a single source task).
    2. The source task has been finished by client **A**, or has been at least assigned to it. (We will call the latter a *future task chain*, as it can only be reached by improved task prefetching.)

Secondary goal: These criterions are only important to choose between tasks that scored equal on the primary goals:

- Prefer tasks with many outgoing dependences (to unlock as much future tasks as possible).
- Otherwise, choose the task randomly (among the best tasks).

The goal is not only to find a task that is communication optimal for the requesting client, but future task assignments have to be considered, too. Some tasks can be more expensive to some client than to another, for example, task chains should almost never be taken away from the client that has finished the source task. Thus, they are ideal targets for task prefetching.

In our implementation, we preferred standard task chains (over future task chains), because they may unlock further tasks, thus increasing the parallelism. Future task chains, however, can improve the cache performance, especially if they are executed sequentially.

C Algorithm: RemoveRedundantEdges

Remark. This section describes the heuristic mentioned in section 3.5. We do not claim that the presented algorithms are new, at least the basic algorithm is based on some existing transitive reduction algorithm, but unfortunately we could not find any traces of its origin. To the best of our knowledge, the modification resulting in the second algorithm is new.

Our heuristic assumes that the task dependence graph has the following properties:

- The graph is *acyclic* (this is guaranteed).
- In general, the graph will be *sparse* (this is very likely).
- In general, there will be *no long dependences*[21] (this is likely assuming that the example is well parallelizable and the quality of the dependence analysis is high).

The basic algorithm 1 traverses the graph's nodes in *reverse* topological order, and incrementally computes the set of reachable nodes R for each node, starting with empty sets. In line 5 the reachable nodes from node N is initialized as $\{N\}$, which is incrementally extended by the loop in line 6 where all outgoing edges from node N are traversed in *ascending* topological order.

All edges from N to some other node M, which have been already marked as reachable from node N, can be removed if the represented dependences do not require communication. Otherwise, the edge cannot be removed, and therefore all nodes that are reachable through M are also marked as reachable from N.

Algorithm 1 Basic algorithm: $O(n^2)$ space usage, where n is the number of nodes.

```
 1: function RemoveRedundantEdges
 2:     Compute topological order
 3:     $R \leftarrow \{\}$                          ▷ mapping: node to set of reachable nodes
 4:     for all nodes $N$ in descending topological order do
 5:         $R[N] \leftarrow \{N\}$                      ▷ init reachable nodes from node $N$
 6:         for all outgoing edges to node $M$ in ascending topological order do
 7:             if $M \in R[N]$ and edge $(N, M)$ causes no communication then
 8:                 Remove redundant edge $(N, M)$
 9:             else
10:                 $R[N] \leftarrow R[N] \cup R[M]$   ▷ add all nodes that are reachable through $M$
11:             end if
12:         end for
13:     end for
14: end function
```

Algorithm 1 cannot be used on huge graphs, because its memory consumption is $O(n^2)$, where n is the number of nodes (or tasks). This is because the set of reachable nodes for each nodes is stored, even if the nodes are no longer relevant.

[21] In this context, the length of a dependence ① $\longrightarrow$ ② is defined as the length of the longest path between ① and ②. For example, all task dependence graphs shown in this thesis contained only very short dependences (of length one or two).

Algorithm 2 tries to reduce this problem by detecting non-relevant nodes, where all incoming edges have already been examined. These nodes can be safely removed from all sets of reachable nodes. Although the space complexity remains $O(n^2)$ for arbitrary acyclic graphs, it significantly reduces time and memory consumption if the graph is sparse and contains no long dependences (or at least not many).

Algorithm 2 Improved version for graphs without long dependences.

```
1: function RemoveRedundantEdges
2:     Compute topological order
3:     Mark all edges as not finished
4:     $R \leftarrow \{\}$                    ▷ mapping: node to set of reachable nodes
5:     for all nodes $N$ in descending topological order do
6:         $R[N] \leftarrow \{N\}$                 ▷ init reachable nodes from node $N$
7:         for all outgoing edges to node $M$ in ascending topological order do
8:             if $M \in R[N]$ and edge $(N, M)$ causes no communication then
9:                 Remove redundant edge $(N, M)$
10:            else
11:                $R[N] \leftarrow R[N] \cup R[M]$   ▷ add all nodes that are reachable through $M$
12:            end if
13:            Mark edge $(N, M)$ as finished
14:            if all incoming edges of node $N$ are finished then
15:                Remove $N$ from all (non-empty) sets in $R$
16:            end if
17:        end for
18:    end for
19: end function
```

In line 15, the algorithm assumes that only non-empty sets are stored in R, while empty sets are represented by non-existing mappings. That allows the implementation to remove N only from non-empty sets. Without this optimization, all sets have to be tested whenever a node is detected as irrelevant, resulting in a quadratic number of tests.

References

[1] David Abramson, Jonathan Giddy, and Lew Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *IPDPS*, pages 520–528, 2000.

[2] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[3] Muhammad Waseem Akhtar and M-Tahar Kechadi. A comparative study of peer to peer dynamic load balancing on embedded topologies. In *PDCN'07: Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference*, pages 182–189, Anaheim, CA, USA, 2007. ACTA Press.

[4] Eduardo Argollo, Michael Claßen, Philipp Claßen, and Martin Griebl. Loop parallelization for a GRID master-worker framework. In *Proc. CoreGRID Workshop on Grid Programming Model*. CoreGRID Tech. Report TR-0080, June 2007.

[5] Jim Basney, Rajesh Raman, and Miron Livny. High Throughput Monte Carlo, In Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, March 1999.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[7] Erick Cantu-Paz. Designing Efficient Master-Slave Parallel Genetic Algorithms. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 455, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann.

[8] Cilk++. `http://www.cilk.com`.

[9] Michael Claßen, Philipp Claßen, and Christian Lengauer. Extending Loop Parallelization for the Grid to Largely Decentralized Communication. In *Integrated Research in Grid Computing, CoreGRID Integration Workshop*. Heraklion : Crete University Press, 2008.

[10] Michael Claßen and Martin Griebl. Automatic Code Generation for Distributed Memory Architectures in the Polytope Model. In *Eleventh International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'06)*, 2006.

[11] Philipp Claßen. Code Generation in the Polytope Model with Non-linear Parameters. Bachelor thesis, Fakultät für Mathematik und Informatik, Universität

Passau, April 2007. http://www.infosun.fim.uni-passau.de/cl/loopo/doc/classenph-b.pdf.

[12] Philippe Clauss and Benoît Meister. Automatic Memory Layout Transformations to Optimize Spatial Locality in Parameterized Loop Nests, January 2000. 4th Annual Workshop on Interaction between Compilers and Computer Architectures, INTERACT-4, Toulouse, France.

[13] Jan Dünnweber and Sergei Gorlatch. HOC-SA: A Grid Service Architecture for Higher-Order Components. In *Proc. IEEE Int. Conf. on Services Computing*, pages 288–294. IEEE Computer Society Press, 2004.

[14] Jan Dünnweber, Sergei Gorlatch, Martin Griebl, Eduardo Argollo, and Christian Lengauer. Making a task farm component parallelize loops for the Grid. In Thierry Priol Sergei Gorlatch, Marian Bubak, editor, *Proceedings of the CoreGRID Integration Workshop*, pages 93–104. CYFRONET Poland, 2006. ISBN 83-915141-6-1.

[15] Ian Foster. What is the Grid? A Three Point Checklist. *GRIDToday*, July 2002.

[16] Ian Foster and Adriana Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[17] Noriyuki Fujimoto and Kenichi Hagihara. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In *Proc. ICPP 2003*, pages 391–398, 2003.

[18] Globus Toolkit. www.globus.org.

[19] The HOC-SA Globus Incubator Project. http://dev.globus.org/wiki/Incubator/HOC-SA.

[20] Jean-Pierre Goux, Sanjeev Kulkarni, Jeff Linderoth, and Michael Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *HPDC*, pages 43–50, 2000.

[21] Vasudha Govindan and Mark A. Franklin. Application Load Imbalance on Parallel Processors. In *IPPS*, pages 836–842, 1996.

[22] Martin Griebl. Automatic Parallelization of Loop Programs for Distributed Memory Architectures. Habilitation thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2004. http://www.fmi.uni-passau.de/~griebl/habil.ps.gz.

[23] Armin Größlinger. Extending the Polyhedron Model to Inequality Systems with Non-linear Parameters using Quantifier Elimination. Diploma thesis, Fakultät für Mathematik und Informatik, Universität Passau, September 2003. http://www.infosun.fmi.uni-passau.de/cl/arbeiten/groesslinger.ps.gz.

[24] Armin Größlinger, Martin Griebl, and Christian Lengauer. Introducing Non-linear Parameters to the Polyhedron Model. In Michael Gerndt and Edmond Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, Research Report Series, pages 1–12. LRR-TUM, Technische Universität München, July 2004.

[25] Armin Größlinger, Martin Griebl, and Christian Lengauer. Quantifier Elimination in Automatic Loop Parallelization. *Journal of Symbolic Computation*, 41(11):1206–1221, November 2006.

[26] Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. In *GRID*, pages 214–227, 2000.

[27] Higher-Order Components for Grids (HOCs). `http://pvs.uni-muenster.de/pvs/forschung/hoc/index-en.html`.

[28] Shiyuan Jin, Guy Schiavone, and Damla Turgut. A performance study of multiprocessor task scheduling algorithms. *J. Supercomput.*, 43(1):77–97, 2008.

[29] M. Tahar Kechadi and Ilias K. Savvas. Dynamic task scheduling for irregular network topologies. *Parallel Computing*, 31(7):757–776, 2005.

[30] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

[31] Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[32] Christian Lengauer. Loop Parallelization in the Polytope Model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[33] Vincent Loechner, Benoît Meister, and Philippe Clauss. Precise Data Locality Optimization of Nested Loops. *J. Supercomput.*, 21(1):37–76, 2002.

[34] LooPo – Loop parallelization in the polytope model. `http://www.infosun.fim.uni-passau.de/cl/loopo`.

[35] Douglas Mcilroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[36] Benoît Meister, Vincent Loechner, and Philippe Clauss. The Polytope Model for Optimizing Cache Locality. ICPS RR 00-03, May 2000.

[37] OpenMP. `http://www.openmp.org`.

[38] OpenMP 3.0 API Specifications. `http://www.openmp.org/mp-documents/spec30.pdf`.

[39] Ilias K. Savvas and M. Tahar Kechadi. Efficient Load Balancing on Irregular Network Topologies Using B+tree Structures. *Sixth International Symposium on Parallel and Distributed Computing (ISPDC'07)*, 0:27, 2007.

[40] Christian Siebert. Efficient Broadcast for Multicast–Capable Interconnection Networks. Diploma thesis, Fakultät für Informatik, TU Chemnitz, September 2006. `http://archiv.tu-chemnitz.de/pub/2006/0182/index.html`.

[41] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional, December 1997.

[42] Intel Thread Building Blocks (TBB). `http://www.threadingbuildingblocks.org`.

[43] The Common Component Architecture Forum. `http://www.cca-forum.org`.

[44] Johannes Tomasoni, Jan Dünnweber, Sergei Gorlatch, Michael Claßen, Philipp Claßen, and Christian Lengauer. LooPo-HOC: A Grid Component with Embedded Loop Parallelization. Technical Report TR-0135, Institute on Programming Model, CoreGRID - Network of Excellence, April 2008.

[45] Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

Index

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe, dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind und dass diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, den February 5, 2009

Philipp Claßen