# EXTENDING LOOP PARALLELIZATION FOR THE GRID TO LARGELY DECENTRALIZED COMMUNICATION

Michael Claßen, Philipp Claßen and Christian Lengauer
*Fakultät für Informatik und Mathematik*
*Universität Passau, D–94030 Passau, Germany*
[classenm,classen,lengauer]@fim.uni-passau.de

**Abstract** Since scalability is a critical issue for the Grid, a communication structure based on a peer-to-peer model is preferable to the more centralized client/server approach. However, in the field of high-performance computing for the Grid, most existing tools cannot deal with complex dependences in the input program or lack a decentralized approach to communication. We present a framework for the automatic parallelization of loop programs with dependences for the Grid using largely decentralized communication. A central server is only used for controlling the execution of tasks and initiating the communication of data between nodes.

## 1. Introduction

Several important advances have lately been made in Grid computing to make it a viable platform for high-performance computing in the scientific field. E.g., improvements have been made with regard to simplifying the development and execution of scientific Grid applications. In one of these efforts, we have pursued a component-based approach to combining automatic loop parallelization with the comfort of a Grid middleware, which enables the relatively simple generation of distributed parallel programs [1]. In the case that data dependences exist between different tasks, the scalability of the generated code depends largely on the amount of communication caused by the applied communication scheme [13]. In this context, decentralized (i.e., peer-to-peer) communication is generally the method which leads to maximum scalability [12].

We present a largely decentralized communication scheme for the use in automatic loop parallelization for the Grid. This method has been implemented using higher-order programming constructs [5]. We have validated our implementation by experimental case studies.

The rest of the paper is organized as follows. Section 2 introduces our communication structure. Section 3 presents relevant concepts of loop parallelization. Section 4 gives an overview of our implementation. Section 5 discusses experimental results from test of our method on the examples of matrix multiplication, polynomial product and successive over-relaxation. Section 6 concludes and discusses future work.

## 2. A Largely Decentralized Communication Scheme

In a massively parallel distributed computing environment, communication bottlenecks must be avoided in order to achieve scalability. Communication must be decentralized as much as possible, but the clients must still be made as light-weight as possible. A peer-to-peer communication scheme is usually considered the best approach [12]. However, existing implementations do not support inter-task dependences or do so only in simple special cases. For this purpose, we propose a peer-to-peer communication structure in combination with our prototype automatic parallelization compiler LooPo, which can handle very complex dependences [8, 10].

Despite the basic idea of decentralized communication, we still retain some form of central control: a dedicated controller keeps track of the dependences between tasks and initiates communication if necessary. This approach prevents clients from having to store information about other clients and to keep this information updated, but it still guarantees that the bulk of the communication and computation is performed independently by the clients.

Figure 1 gives an overview of the interaction between the controller and several clients. The individual responsibilities of controller and client are as follows.

### 2.1 The Controller

The controller has a number of responsibilities. It generates and schedules tasks for execution according to the restrictions given by a dynamic task graph. It also controls the communication between tasks, keeps track of the completion of tasks and collects the final results of the computation. The individual steps are described in more detail as follows.

1 At startup time, the controller uses run-time parameter values to obtain a list of all occurring tasks, which are represented as so-called *task ids*. Then, the dependence information provided by LooPo is used to generate a task graph that represents all dependences between task ids. The task graph is a DAG, whose edges represent dependences. Initial tasks are the sources of the DAG and final tasks are sinks. This graph

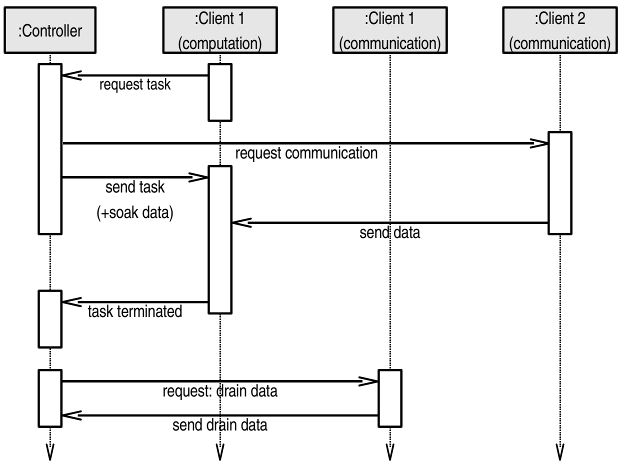

*Figure 1.* a typical communication sequence

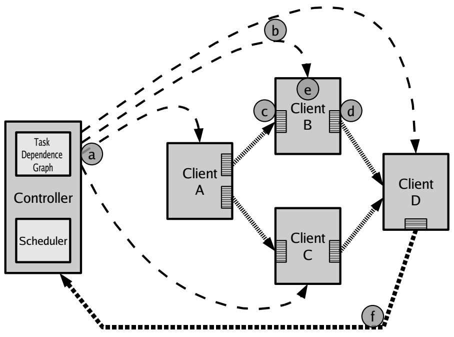

*Figure 2.* P2P communication scheme

can be used to determine the conditions for an execution order to be correct. It is also used to determine communication partners if data has to be exchanged between dependent tasks.

2 Tasks are generated on demand, if a client requests a task from the controller. At the generation of tasks, the initial tasks have to be provided with the initial set of data elements from arrays and variables. This procedure of distributing initial data is called *soaking*. In our implementation, the controller supplies each initial task with the corresponding set of soaking data (Fig. 2(a)). The corresponding collection of the final results of computation is called *draining* and is performed by communication between clients and controller (Fig. 2(f)).

3 Another responsibility of the controller is to organize the execution of tasks. For this purpose, a *task scheduler* is part of the controller class. It uses the task graph to determine a set of tasks that are ready for execution (i.e., that do not depend on unfinished tasks). Then it chooses the next task based on a scheduling strategy that can be adjusted by the programmer, if necessary. Note that, in a Grid middleware environment, the scheduling strategy of the middleware could be used [4].

4 After the execution of a task, the controller marks it as terminated. When a subsequent task is scheduled for execution, the controller uses the task dependence graph to determine all tasks on which this task depends. Then the controller notifies the corresponding clients to send their data to the destination client of the newly generated task. Subsequently, the communication of data between clients takes place in a largely decentralized communication pattern (Fig. 2(d)).

## 2.2 The Client

The main responsibility of a client is to execute tasks. However, in the peer-to-peer communication pattern used, clients are also responsible for communicating data to the dependent communication partner that requires the data for further computations. Each client requests tasks by the controller, which then assigns a new task based on the task dependence graph and the applied scheduling strategy. The main execution order of a task on a client is as follows.

1 Each task receives data from already terminated tasks if data dependences exist (Fig. 2(c)). In this case, only dependences are taken into account that induce a need for data communication, i.e., communication stemming from a write access followed by a read access to the same data element.

2 Data is written from the receive buffers into local arrays, thus updating the local version of the arrays with the current state of computation.

3 The computation is performed on the local arrays (Fig. 2(e)).

4 The client informs the controller that the computation has terminated and requests a new task.

5 If the controller notifies the client of a communication to another dependent task, the requested data is written to the corresponding send buffer for each target task id. Note that this step and the following communication steps are executed in parallel with the computation using a second thread. Thus, computation and communication are overlapping in a non-blocking fashion.

6 In order to determine the physical location of each communication partner, the client receives a physical destination processor for every destination task id (Fig. 2(b)).

7 Each prepared send buffer is sent to the corresponding target processor (Fig. 2(d)).

8 At the end of the computation, all final values of data elements that reside typically in locations scattered across the clients. The action of collecting these values and returning them to the controller is called *draining* and is governed by the controller (Fig. 2(f)). For more details on how data elements to be drained are identified, see Section 4.1.

In order to simplify the adaptation of this peer-to-peer communication scheme to a Grid infrastructure, we use a high-level approach for our implementation. We extend an existing taskfarming framework based on the use of so-called *Higher-Order Components* (HOCs) [5]. Further implementation details on the aspects of code generation are given in Section 4. In the next section, we describe how parallel tasks are extracted from a sequential input program.

## 3. Automatic Loop Parallelization

In order to obtain a distributed program consisting of tasks that can be executed in parallel, we use methods for the automatic parallelization of loop programs. These methods are based on a mathematical model, the so-called *polytope* (or *polyhedron*) *model* [11]. This moel supports the use of mathematical optimization techniques in the form of integer linear programming for the complete parallelization process. It can be applied to perfectly or imperfectly nested loops which compute on array variables. Loop bounds, array accesses and data dependences must be affine expressions, i.e., linear in the indices of the surrounding loops and in symbolic and numeric constants. Corresponding methods are implemented in a prototype compiler called LooPo [10, 8].

The transformation of the input loop program into a parallel target program is performed in four phases [8].

1 The input program is analyzed and the dependences are computed [3, 6]. The result is a set of polyhedra that represent all computations, all *array accesses*, and a set of *dependence relations* between computations. These dependences determine the correct partial order of the computations and, also, the necessary communications in a distributed-memory environment.

2 Two piecewise affine functions are computed: the *schedule* maps each computation to a logical execution step, and the *placement* maps each

computation to a virtual processor. The goal is to extract all available parallelism, independently of any machine parameters, e.g., the number of processors. This phase is called the *space-time mapping.*

3 Several time steps and/or virtual processors are aggregated to a so-called *tile.* Each tile represents a *task*, which can be assigned to an available client on a physical processor for sequential execution. This phase is crucial for efficiency, since only coarse-grain parallelism can lead to a speedup in distributed systems in which the network is typically orders of magnitude slower than the compute nodes [9].

4 Code for the computations and communications is generated [2]. This part of the parallelizer had to be redeveloped completely for our implementation and, thus, is discussed in more detail in the next section.

Note that we have adapted the classic approach (in Phase 3) of mapping the execution of tiles on statically predetermined processors for the use in a dynamic taskfarming approach [1]. One consequence of the changes is an increased number of tiles (more than the number of available physical processors) in order to achieve dynamic load balancing. Furthermore, a task dependence graph is constructed at run time for scheduling the execution of tasks dynamically at run time.

## 4. Generating Code

In order to simplify the adaptation to a Grid middleware, we use the high-level programming construct of a so-called higher-order component (HOC) [5]. We also reuse parts of a previous implementation based on taskfarming [1], which are not affected by the choice of communication pattern. Therefore, this section focuses on aspects of code generation related to peer-to-peer communication.

### 4.1 Communication

We distinguish three different kinds of communication:

- Before the start of the computation, the controller sends initial data to the clients. This is called *soaking.*
- While the computation proceeds, clients communicate with each other.
- After the computation has terminated, the clients send their final data back to the controller. This is called *draining.*

We specify the communication for soaking and draining by extending our original input program: additional soaking and draining statements are included before the first and after the last statement, respectively. These additional statements enumerate accesses to arrays from the input program. Thus, additional dependences are introduced which can be used to describe the flow of data from the input arrays on the controller to the virtual space-time coordinates of the computation operations on a client and back to the location of the output arrays on the controller.

The peer-to-peer communication of data between clients is described by dependences between statements of the original input program. Based on these dependences, we generate a loop nest for the client which is parametrized by the task id of the source and destination client of the communication. Depending on these parameters, the loop nest enumerates all data elements that are communicated between the two tasks. The data is stored in buffers and is transmitted when the controller notifies the client of the target processor id of the communication.

## 4.2 Controller

For the controller, the code for constructing a task dependence graph (TDG) can be reused from our previous taskfarm implementation [1]. The scheduler can also be reused, but it is sensible to take run-time information from the TDG into account to schedule tasks for clients in a way that prevents unnecessary peer-to-peer communication. The changes required have been implemented in a new scheduler class. In addition, the controller has to initialize new tasks containing soaking data, as mentioned in Section 4.1.

## 4.3 Client

The communication-related code generation for the client involves generating code for receiving data from other clients and loading the data from the receive buffers, writing data to send buffers and performing a communication to a destination processor. These communication operations have to be generated for all three kinds of communication mentioned in Section 4.1. For this purpose, a model representation of dependences and array accesses is used. We generate loops that enumerate all data elements that have to be transferred between either the controller and a client (in the case of soaking and draining) or between clients (in the case of peer-to-peer communication). These loops are parametrized with the task ids of the communication partners of the sender and receiver processor involved. Note that, for a given pair of communicating task ids (or a communication between the controller and a specific task), the same loop nests are used for both reading from buffers and writing to buffers, because of the ordering of elements in communication buffers.

## 5. Experiments

We designed experiments to prove that scalability can be achieved by combining a peer-to-peer communication scheme with an automatic parallelization of loop nests with dependences. For these first tests, we used our own run-time environment on a local area network. However, the current implementation is aimed at easy portability to a Grid middleware (such as HOC-SA [5]) by using the concept of HOCs.

### 5.1 Example Input Programs

In our experiments, we used the following numerical calculation examples as kernels for our input programs [7]:

- Matrix multiplication:

```
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];
```

- Polynomial product:

```
for (int i = 0; i <= N; i++)
    for (int j = 0; j <= N; j++)
        C[i+j] = C[i+j] + A[i] * B[j];
```

- One-dimensional successive over-relaxation (SOR1d):

```
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;
```

These example programs have been chosen as typical examples of mathematical calculations on arrays using nested for-loops. In all cases, there are dependences caused by accesses of shared array elements by different loop iterations.

We used our implementation to perform the parallelization steps described in Section 3 and obtain a distributed target program that uses the communication scheme as described in Section 2.

### 5.2 Results

We tested our target programs on a 100MBit Ethernet connecting various desktop PCs with processor ranging from AMD Athlon XP 3000+ to Intel Core 2 Duo 6400, using between 2 GB and 4 GB of main memory.

Figure 3 shows the resulting speedup of our parallelized target programs executed using the peer-to-peer communication scheme as compared to the run time of a sequential version running on one processor (AMD Opteron). In the case of matrix multiplication, we used a $4000 \times 4000$ matrix. In the case of the polynomial product, we used $N = 500k$ and, for SOR1d, we used problem sizes of $M = 500k$ and $N = 1000k$. In all three cases, we adapted the granularity of parallelism to the specific problem size and communication behavior by hand, by choosing the appropriate tile size.

The resulting speedup exhibits scalability for comparably large processor numbers (upto 32 processors) – especially in the case of polynomial product, which contain a simple dependence structure, leading to task "chains", with no dependences between separate chains. Our scheduler can exploit this structure to avoid inter-task communication by scheduling a chain of tasks on the same client. The dependence structure of SOR1d produces more inter-task dependences, which disables an exploitation of task chains. In case of matrix multiplication, the scalability is limited, for larger problem sizes, by the memory requirements. The reason is that our approach does not yet support distributed storage of arrays on clients and, thus, the memory used for the two-dimensional arrays in the matrix multiplication leads to swapping effects for larger problem sizes.

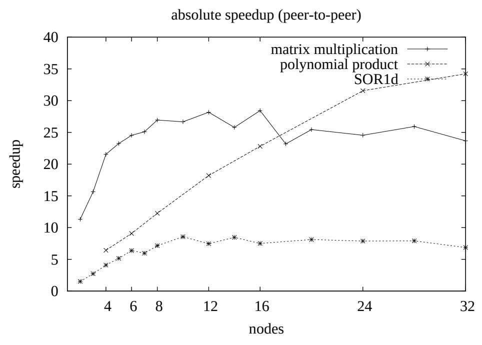

*Figure 3.* Absolute speedups in the peer-to-peer implementation

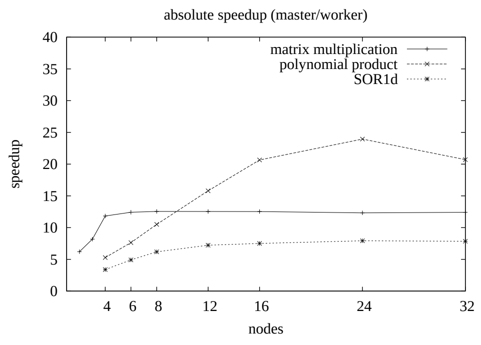

*Figure 4.* Absolute speedups in the task-farming implementation

Figure 4 shows the speedup on a previous implementation based on a modified master-worker taskfarming approach that has been extended to deal with inter-task dependences. Overall, there is an improved scalability using the new peer-to-peer approach. In the case of SOR1d, the available amount of parallelism obtained from our parallelization algorithms did not provide enough communication volume to demonstrate the benefits of a peer-to-peer communication scheme convincingly: for the problem sizes we tested, the overall consumption of network bandwith remains low (up to about 30% of the available bandwith). We expect the peer-to-peer approach to outperform the master-worker implementation in applications whose communication volume reaches the limit of available network bandwidth.

## 6. Conclusions and Future Work

Most current high-performance computing projects on the Grid focus on task-based parallelism without any inter-task dependences. In the presence of inter-task dependences, a decentralized communication scheme is required to achieve scalability. We have proposed a largely decentralized communication scheme for automatically parallelized code to be used in a Grid middleware infrastructure using Higher-Order Components (HOCs).

We have given an overview of the basic communication scheme and the structure based on a central controller and clients that perform decentralized communication. In order to validate our approach, we have implemented an automatic loop parallelization tool that uses our largely decentralized communication scheme for handling communication caused by inter-task dependences.

Our experiments proved that our largely decentralized communication scheme leads to better scalability than a previous taskfarming based implementation. In the example of a polynomial product algorithm, our generated target program scaled for up to 32 processors. The examples of matrix multiplication and SOR1d showed that the problems sizes were not large enough to achieve scalability for more than 16 processors. However, our largely decentralized implementation still resulted in speedup improvements compared to the centralized taskfarming based approach.

For future work, we will make the necessary adaptions to the HOC-SA Grid middleware. We also want to focus on developing methods for using distributed arrays for the computation to achieve scalability with regard to memory consumption in clients.

## Acknowledgments

Financial support from the German Research Foundation (DFG) for project *CompSpread* and from the EU FP6 Network of Excellence *CoreGRID* is gratefully acknowledged. Thanks to Jan Dünnweber, and Armin Größlinger for discussions, help and technical support.

# References

[1] 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*, pages 516–527. CoreGRID Tech. Report TR-0080, June 2007.

[2] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. 13th IEEE Int. Conf. on Parallel Architectures and Compilation Techniques (PACT 2004)*, pages 7–16. IEEE Computer Society Press, September 2004.

[3] Jean-François Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In Larry Carter and Jeanne Ferrante, editors, *Languages and Compilers for Parallel Computing (LCPC'99)*, Lecture Notes in Computer Science 1863, pages 286–302. Springer-Verlag, 1999.

[4] Catalin Dumitrescu, Dick Epema, Jan Dünnweber, and Sergei Gorlatch. User-transparant scheduling of structured parallel applications in grid environments. In *Workshop on HPC Grid Programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing (HPC-GECO+COMPFRAME)*. IEEE Computer Society Press, 2006.

[5] 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.

[6] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[7] Carl-Erik Fröberg. *Numerical Mathematics – Theory and Computer Applications.* Benjamin/Cummings, 1985.

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

[9] Martin Griebl, Peter Faber, and Christian Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, March 2004.

[10] Lehrstuhl für Programmierung, Universität Passau. The polyhedral loop parallelizer: LooPo. http://www.fmi.uni-passau.de/loopo/.

[11] 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.

[12] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical report, HP Labs, 2002.

[13] Eduardo Javier Huerta Yero and Marco Aurélio Amaral Henriques. Speedup and scalability analysis of master–slave applications on large heterogeneous clusters. *J. Parallel and Distributed Computing*, 67(11):1155–1167, 2007.