LOOP PARALLELIZATION FOR A GRID MASTER-WORKER FRAMEWORK*

Eduardo Argollo, Michael Claßen, Philipp Claßen, Martin Griebl
Fakultät für Informatik und Mathematik
Universität Passau, D–94030 Passau, Germany
[eargollo,classenm,classen,griebl]@infosun.fmi.uni-passau.de

Abstract Despite the evolution in Grid middleware, the development and execution of Grid applications is still not simple. We propose an approach to parallelizing applications straight to the Grid. Both the parallelization and application execution processes should be as simple as possible. We present a software architecture that combines loop parallelization with Higher-Order Components. We develop a Higher-Order Component for an extended master-worker framework in which tasks may have dependences, and we adapt the LooPo parallelization tool to generate the application-specific code for such Higher-Order Components. Experiments, automatically parallelizing the matrix multiplication algorithm with different parameter settings, are presented in order to validate our approach.

Keywords: Automatic Parallelization, Higher-Order Components, Master-Worker, Web-services

1. Introduction

In recent years, Grid computing has evolved to become an important platform for high-performance computing in scientific applications. But despite the evolution of Grid middleware, the development and execution of Grid applications is still not simple. For an application to be executable on the Grid, the program must be available in a distributable, i.e., parallel form, and, in order to run this parallel program, the Grid middleware must be configured properly for execution.

Our ultimate goal is to simplify both steps, so that the user eventually just needs to upload a sequential program using a Web service and press the execution button. In other words: the usage of the Grid as a huge parallel computer must be as simple

*This work was carried out for the CoreGRID IST project nr. 004265, funded by the European Commission.

as possible, and all Grid problems and challenges, e.g., heterogeneity, slow network connections, dynamic load balance or even failures, must be transparent to the user.

In order to achieve this goal, we divide the problem into two main parts.

As a first task, we use automatic parallelization to convert a sequential program to a parallel one for distributed memory. This technique is well developed for scientific programs, i.e., for programs that spend most of their time in loops of array computations. Corresponding methods are based on a mathematical model and implemented in a prototype compiler called LooPo [6, 8]. We have adapted LooPo so as to generate tasks for an extended master-worker framework that also allows tasks which depend on other tasks. Our experiments demonstrate the influence of parallelization parameters on the execution time in the distributed run-time environment.

As a second task, we map the distributed program to the Grid. For this purpose, we suggest to use a "master-worker with dependences" Higher-Order Component (HOC) [3], enabling the reuse of the master-worker standard code and the Grid environment configuration files. We explain here how, using this framework, the execution becomes transparent to the user.

We implemented the first task using an interface framework, that can be used for initial testing in a controled environment and then adapted to the HOC structure (second task). This framework takes tasks as they should be executed on the Grid and executes them on a distributed set of computers in our department. With this interface, we have run experiments that demonstrate that the automatic parallelization can be used to generate tasks and any other data structures that are needed for execution in a master-worker framework. The essential new part in this framework, when compared to a standard master-worker framework in a distributed environment, is the existence of a task dependence graph, which is automatically generated by LooPo. Thus, our framework can also deal with dependent tasks. Beyond the actual execution, the framework provides us with interesting insights about all performance parameters, e.g., the number of workers that are used at each computation step, the amount of communication, or the load of the processors. Furthermore, different buffering techniques have been implemented in the framework and can be compared, e.g., a single buffer framework in which task generation, execution and joining are sequential, or a double buffer framework, in which a worker obtains a second task while it still executes the first one.

Our first experiments with multiplying two square matrices yield the expected results: the performance results scale up to 18 processors, where the relative speedup for 18 workers is about 16 – an efficiency of 89%! Note that, for Grid applications, both the problem size and the number of processors must be increased – which should lead to similar speedups.

To summarize, our main contributions are: (i) we propose a novel approach that combines automatic loop parallelization with a Grid execution environment to simplify program installation and execution in the Grid; (ii) we extend the master-worker framework so as to deal with dependent tasks; (iii) we provide an interface frame-

work for validating the approach (which can furthermore be used directly for a parallel execution of an originally sequential program on a network of workstations or a cluster with distributed memory); (iv) we have experimental results that show that our approach scales well and has good efficiency.

The rest of the paper is organized as follows: Section 2 introduces our proposed system architecture and the master-worker HOC. Section 3 presents the necessary ideas from loop parallelization. Section 4 is about the special aspects when generating tasks. Section 5 discusses the application of our approach to the matrix multiplication and shows some experimental results. Section 6 contains conclusions and ideas for future work.

2. HOC and System Architecture

In order to achieve the goal of simplicity in the parallelization and execution of applications on the Grid, we propose a software architecture that combines a component based architecture and loop parallelization. For this purpose, we use the concept of Higher-Order Components introduced in [3].

Higher-Order Components (HOCs) were designed to provide Grid users with high-level programming constructs, prepackaged with (parallel) implementations and the required middleware configuration files. Each HOC implements a generic pattern of parallel behavior with a specific communication structure, e.g., a pipeline or a task farm. A HOC can be customized for a particular application by providing it with arguments which may be both data and application-specific code. HOCs are made accessible to the Grid user via Web services.

Fig. 1 shows the HOC service architecture [5]. At the HOC repository, the application programmer finds components that are written by Grid experts and represent optimized generic code for parallelization paradigms like master-worker, pipeline or divide-and-conquer, for example. The application programmer selects a HOC and programs the specific codeparts for its application, for example the source code of the steps of a pipeline. The application-specific source code is uploaded (only once) to the code service (Fig. 1 (a)). When an execution is requested from the HOC service (Fig. 1 (b)), the application-specific code is loaded from the code service (Fig. 1 (c)) and joined to the HOC code resulting in an application that is ready to run on top of the Grid middleware (Fig. 1 (d)).

Our proposal is to add a loop parallelization HOC to this service architecture and to develop another HOC that represents a master-worker with dependences. Our proposed system architecture can be seen in Fig. 2. The sequential source code is stored once in the code service (Fig. 2 (a)). The first time an execution is requested by the LooPo loop parallelization HOC (Fig. 2 (b)) this code is loaded (Fig. 2 (c)). The LooPo HOC then generates the necessary files that are joined with the master-worker code from the HOC repository (Fig. 2 (d)), stores them for further use (Fig. 2 (e)) and

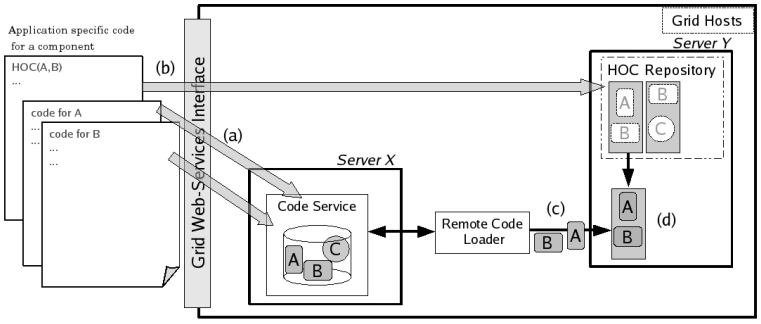

Figure 1. Higher-Order Components architecture.

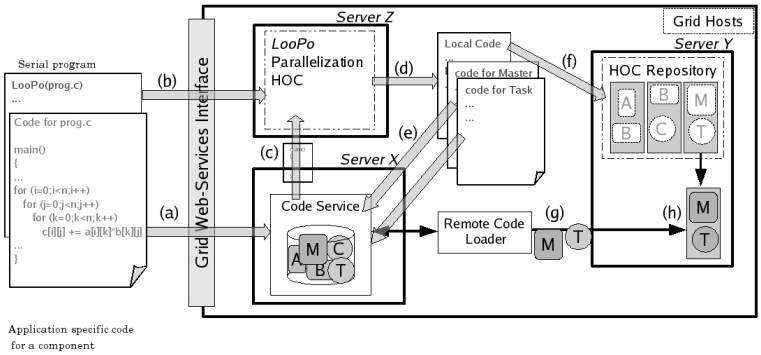

Figure 2. System architecture of our proposed solution.

requests a HOC execution (Fig. 2 (f)), followed by the steps previously explained (Fig. 2 (g) and (h)).

The master-worker programming model consists of two kinds of entities: one master and multiple workers. The master is responsible for decomposing a problem into tasks and distributing these tasks among a farm of workers, as well as for gathering the partial results in order to produce the final computation result. The workers execute in a cycle: they receive a message with the task (input), process the task, and send the result back to the master (output).

The "master-worker with dependences" HOC implements this master-worker generic behavior requiring as an input the application-specific codeparts. Doubly-buffered workers, allowing simultaneous computation and communication through threads, enhance the execution efficiency. In order to use this HOC, the application developer must provide two parameters (classes): the application master and the application task. The application master is responsible for the generation and joining of tasks.

Application Master

```
public class AppMaster extends UserMaster {
  //Master data
  public long startUp(String args[], UserScheduler sched) {
    //Startup procedure
    //Statically add tasks to scheduler...
    ...
  }
  public long joinTaskResult(UserTask taskResult) {
    //Joining result
    AppTask task (AppTask) taskResult;
    result += task.getResult();
  }
  public UserTask getNextTask() {
    //Dynamically add tasks to scheduler...
    AppTask task new AppTask(...);
  }
  public int end(){
  //Finalization procedure
  ...
  }
}
```

Application Task

```
public class AppTask extends UserTask {
  //Task data
  double result;
  ...

  public AppTask(...) {
    ...
  }

  public int execute()        {
    //Task execution functionality...
    ...
        }
}
```

Figure 3. Example code showing the required methods for an application to use the master-worker framework.

The application master might also have application-specific variables and activities for initiating and finishing the execution.

The application task must have the task data as well as the method that represents the task execution activity. An example of the required application master and application task code suitable for our HOC can be seen in Fig. 3. The methods of both the application master and the task are invoked by the master-worker HOC.

One of the basic characteristics of loop parallelization is the possibility of data dependences between different parallelized steps. To be able to handle dependences, our master-worker HOC, differently from the original, allows the existence of dependent tasks. Tasks can be identified by indices and, at initiation time of the application-master, it is possible to define a dependence graph of the tasks using these indices. The execution will then proceed following these dependences. Currently the master-worker framework provides one method for adding tasks and dependences (statically or dynamically). It is important to point out that the data load of a task just happens when all its predecessors were joined in the master.

When tasks and dependences are added, the master-worker framework builds a tasks dependence graph that can be used to determine the parallelization level (maximum number of workers) of this application. For example, the automatic parallelization of an $M \times M$ matrix in $T \times T$ tiles generates $(M/T)^2$ parallel tasks. Considering the double buffer strategy, it is possible to predict that the maximum amount of workers that can be used efficiently is half, i.e.

$$WorkersLimit = \frac{1}{2} * \left(\frac{M}{T}\right)^2 \tag{1}$$

3. Traditional Loop Parallelization

In the context of high-performance computing, the automatic parallelization of loop programs can be well performed using a mathematical model, the so-called *polytope model* [9]. Its applicability is restricted to perfectly or imperfectly nested

loops which compute on array variables and whose bounds and data dependences are affine expressions, i.e., linear in the indices of the surrounding loops and in symbolic and numeric constants. The advantage is that the complete parallelization process can be based on integer linear programming, i.e., on mathematical optimization techniques.

In the extended polytope model, the *polyhedron model*, a four-phase approach is applied to transform the input loop nest to a parallel target program that is adapted to the system architecture [6].

In the first phase, the input program is analyzed and the dependences are computed [2,4]. The result is a set of polyhedra that represent all computations – the so-called *index sets*, all *array accesses* of every computation point inside the index set, and a set of *dependence relations* between the elements of the index sets. These dependences determine the correct partial order of the computations and, also, the necessary communications in a distributed memory environment (a dependence between two different processors causes a communication). Note that, in the typical parallelization framework, the data are sent directly from the producer to the consumer in order to save communication cost; this must be changed if we move to a master-worker framework.

In the second phase, 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 processors. The idea is that all available parallelism is extracted, independently of any machine parameters, e.g., the number of processors. This step – the *space-time mapping* – can be used in the master-worker framework without change.

In the third phase, several time steps and/or virtual processors are aggregated to so-called *tiles*, which are then distributed across the available physical processors for atomic execution. This step is crucial for efficiency, since only coarse-grain parallelism can lead to speedup in distributed systems in which the network is typically orders of magnitude slower than the compute nodes.

Our method allows to adapt the granularity to the number of available processors and, independently, to the target architecture's cost ratio of computation vs. communication [7]:

- Tiling space loops means aggregating a set of virtual processors to be executed on the same physical processor, i.e., it allows adaptation to the number of processors.

- Tiling time loops means aggregating logical time steps, and allowing communication only at the border of the aggregated time steps. This reduces the number of communication startups, but at the cost of an increased computation duration, since the receivers of messages are delayed when the sends are postponed until a tile border is reached. A cost model can be used to adjust the tile size with the ratio of communication startup cost and computation cost.

Technically, once the tile's shape and size have been determined, tiling means enumerating all computations hierarchically, i.e., in two steps: first a set of loops enumerates the tiles, the *tile loops* and, nested inside, another set of loops enumerates the points within each tile, the *body loops*.

In the master-worker framework, these tiles essentially define the tasks. The main differences are as follows:

- For load balancing reasons, one must generate more but smaller tiles, which can then be distributed to the processors according to their load. A cost model can be used to select a tile size that is a good compromise between load balancing and the amount of communication startups.
- The ID of the physical processor to which a tile is mapped becomes useless; note that, nevertheless, computing a placement in Phase 2 still remains a good idea as this groups computations working on the same data close together – so that they tend to be within the same task after tiling, thus reducing communications and dependences between tiles.
- The remaining inter-tile dependences cause the tasks to be dependent. This dependence information is necessary for the master-worker framework to schedule the tasks, i.e., we need an export interface for this traditionally internal information. Note that this exported task dependence graph (TDG) works with the IDs of the tasks instead of the tasks themselves.
- In contrast to tiles, tasks do not exchange messages directly. Instead, all necessary input data must be sent with the task definition, and all output data, i.e., the result of the task, must be merged with the output data of the other tasks, i.e., the results of the tasks must be joint.

In the fourth and last phase, code for the computations and communications is generated [1]. This part of the parallelizer must be completely rewritten for the master-worker framework and, thus, discussed in more detail in the next section.

4. Generating Tasks

In this section, we describe briefly how the Java code for the *Master* and *Task* classes is generated.

4.1 Code generation for class Master

4.1.1 Important attributes. The Master class to be generated contains all parameters and data arrays of the application program. They are declared as *static* and are accessible by the master process. This allows an easy loading of the input data to new tasks, since they are created on the processor of the master before the framework sends each task to a worker.

Furthermore, the Master class contains a task scheduler including the task dependence graph (TDG) and, for keeping track of the progress of computation, the number of active (unfinished) tasks.

4.1.2 Important methods. The master takes care of starting up new tasks and merging their results after they have finished computation. For this purpose, two methods for the Master class are generated as follows.

startUp The startUp method is called only once. It creates the "structure" of the tasks. For this purpose, we generate code for enumerating all tasks and all dependences between the tasks. For each of these two steps, a set of loop nests is generated:

- The tile loops from traditional loop parallelization are used to enumerate all task IDs. These IDs, each coded as an int array, are added to the scheduler's list of tasks.
- In order to create the task dependence graph, the dependence relations of the application program are transformed by applying the space-time mapping and the tiling. For each resulting dependence relation, the tile loops of its sources and destinations are enumerated and a call is generated to add an edge from the source to the destination tile in the task dependence graph.

joinTaskResult After a task has finished its computation, the task's computed data has to be merged with data from other tasks by the master. This is achieved by generating code, that uses the already computed array accesses to determine which data has been written by the task that is being joined. In specific, loops are generated that enumerate all array indices for which the task writes data and, in the body, code is inserted that reads from the local buffer into the global static array at the position specified by the current loop iteration vector.

4.2 Code generation for class Task

4.2.1 Important attributes. The Task class uses a buffer to store elements that are exchanged with the Master. Besides other locally used variables, declarations are generated for parameters and arrays that are used during the actual computations.

4.2.2 Important methods. Beside the code for initializing the task's ID, we also have to generate code for populating the task with data and insert some method for the actual execution of the computation.

constructor The constructor takes the ID of the task to be generated and stores it locally. Then, the initial data must be uploaded to the task. The corresponding code is generated based on the array accesses already computed. Specifically,

loops are generated that enumerate all array indices for which the task reads data from the global arrays – which, in our context, are stored at the master. The body of such a loop nest then consists of a statement that reads from the array position specified by the loop indices and stores the data element in the local buffer.

execute In the execute function, we first generate code for two sets of loop nests. The first set is responsible for initializing the local arrays with the data that was stored in the buffer. For this purpose, we can reuse the loop nests that have been generated for the constructor, but we have to adjust the body statements in order to write to the local arrays instead of reading from the Master arrays.

The second set of loop nests is responsible for enumerating the iterations for the actual computation statements. For this purpose, we use the body loops that are generated during tiling. Note that we do not use the tile loops here but, instead, we treat those dimensions as parameters, whose actual values are given by the task ID.

5. Experiments

We designed experiments to prove that the automatic parallelization approach is suited to generate the required master-worker data and that it is possible to achieve scalability with this approach. We adapted the LooPo parallelization tool to generate the required source code. This code is passed to the developed master-worker with dependences framework and executed. For this first validation approach, both the Grid middleware Web service structure and the HOC architecture have not been used.

In order to analyze the speedup and efficiency, we performed the experiments on a homogeneous cluster with 20 nodes (up to 19 workers) using six different tilling options: 100-100, 200-200, 300-300, 400-400, 500-500, and 600-600. The first tiling parameter determines the width of the tiles in the time dimension. For matrix multiplication, this determines, for a fixed array cell of the result matrix, the number of summations to that array cell that are coalesced in one task. A larger number leads to fewer tasks and, thus, to reduced communication, but also to less parallelism. The second tiling parameter determines the number of rows of the result matrix that are coalesced in one tile. The tiling of the third dimension, i.e., the loop enumerating the columns of the result matrix is not mentioned explicitly as it is executed fully sequentially inside a task.

The speedups in the experiments are depicted in Fig. 4. The graphic shows the relation between tiling and speedup. For small tile sizes the system is quickly cupped by the tasks' communications between master and workers. As the tile grows, the speedup also increases, reaching 16 on 18 workers using a tile width of 600 – an efficiency of 89%.

It is interesting to observe that the automatic parallelization tiling must be big enough to have a good computation-communication ratio but small enough to enable

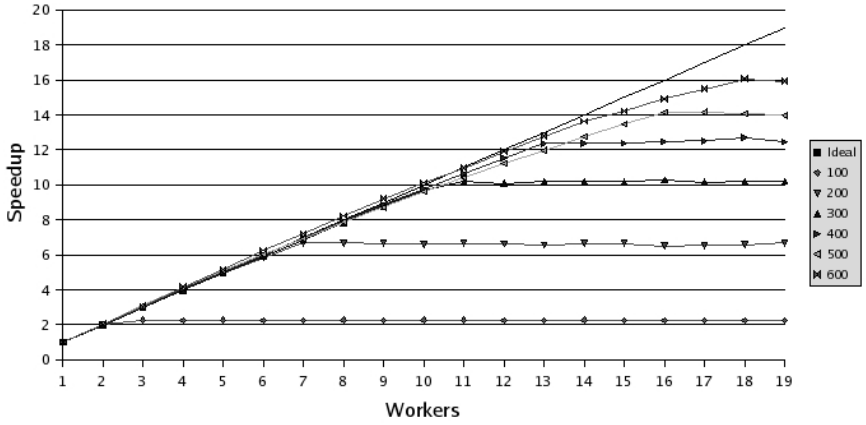

Figure 4. Speedup for different tiling options and different amount of workers.

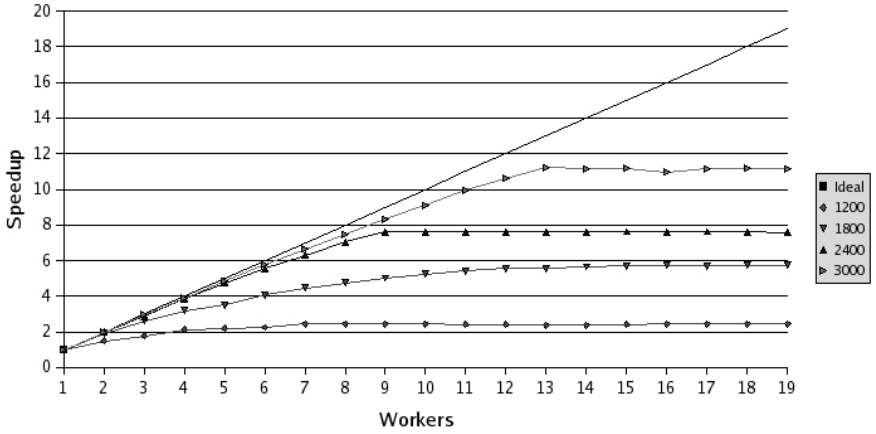

Figure 5. Speedup for the 600 tile with different matrix sizes.

a good number of simultaneous workers. As stated in Section 2, the maximal number of workers that can be used simultaneously for matrix multiplication is given by Equation (1). This means that, for a tile width of 600, the scalability for square matrices of widths 1200, 1800, 2400, and 3000 is limited to approximately 2, 4.5, 8, and 12.5 workers, respectively. Figure 5 shows the speedup for tiles of width 600 with different matrices, demonstrating its limited scalability.

These experiments show that scalability of the automatically generated code using our approach is attainable but depends on the problem size and the tile size. Once

targeting the Grid, both, the problem and the tiling will have to grow substantially, scaling the problem together with the system.

6. Conclusions and Future Work

Although, in the recent years, the Grid has emerged as an important high-performance computing platform, it is not a simple matter to develop Grid applications. It is imperative to have tools that can automatically parallelize applications and transparently execute them in the Grid, providing to Grid users the maximum of simplicity.

We have propose here a novel approach to providing the simple and transparent parallelization and execution of sequential applications in the Grid. This approach is based on combining loop parallelization and Grid Higher-Order Components. In this paper, the architecture of our proposal was detailed, showing the interconnection between the components and the Grid middleware.

In order to validate our approach we have developed a master-worker with dependences HOC and adapted the LooPo parallelization tool to generate the application specific code for this HOC. We tested the system parallelizing the matrix multiplication algorithm with some tiling options. The testbed was a cluster with 20 homogeneous computers.

The experiments showed the viability of our approach, presenting better performance and scalability with larger tiling. The experiments also denoted the relationship between the tiling and the tasks' parallelism: bigger tiles improved the computation-communication ratio but decrease the amount of parallel tasks.

The most important piece of future work is to adapt the HOC structure to the defined interface, as proposed in the system architecture, so that the parallel execution can be carried out directly in the Grid. We also have as future work to test the system with different loop applications and tilings, and to develop a model for supporting the tiling decision.

Acknowledgments

Financial support was gratefully received from the German Research Foundation (DFG) under project *CompSpread* and from the EU by the *CoreGRID* network of excellence. Thanks to Jan Dünnweber, Amin Größlinger, Genaro Costa and Maximilano Gaitan for discussions, help and technical support.

References

[1] Cédric Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002.

[2] 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, 12th International Workshop, LCPC'99*, LNCS 1863, pages 286–302. Springer-Verlag, 1999.

[3] Jan Dünnweber and Sergei Gorlatch. HOC-SA: a grid service architecture for higher-order components. In *Services Computing, 2004. (SCC 2004). Proceedings. 2004 IEEE International Conference on*, pages 288–294, 2004.

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

[5] Sergei Gorlatch and Jan Dünnweber. From grid middleware to grid applications: Bridging the gap with hocs. In *Future Generation Grids*. Springer Verlag, 2005.

[6] Martin Griebl. Automatic parallelization of loop programs for distributed memory architectures, 2004. Habilitation thesis. Also available as http://www.fmi.uni-passau.de/∼griebl/habil.ps.gz.

[7] 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.

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

[9] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.