MAKING A TASK FARM COMPONENT PARALLELIZE LOOPS FOR THE GRID

Jan Dünnweber and Sergei Gorlatch
University of Münster, Department of Mathematics and Computer Science
CoreGRID Institute on Programming Models
Einsteinstrasse 62, 48149 Münster, Germany
duennweb@uni-muenster.de
gorlatch@uni-muenster.de

Martin Griebl and Christian Lengauer
University of Passau, Department of Informatics and Mathematics
CoreGRID Institute on Programming Models
Innstrasse 33, D-94032 Passau, Germany
griebl@fmi.uni-passau.de
lengauer@fmi.uni-passau.de

Abstract We present the combination of two approaches to the simplification of Grid application development: component-based programming and automatic loop parallelization. Components are reusable software building blocks which contain the code for solving recurring problems, together with the required middleware configuration, such that new applications can be generated by customizing components and combining them together. Automatic loop parallelization is a mechanism for the transformation of sequential code into parallel code, such that programmers can take advantage of a multiprocessor platform without having to deal with code parallelization. We study two particular implementations of these approaches: the Higher-Order Component Service Architecture (HOC-SA) and the LooPo loop parallelizer. The HOC-SA provides Higher-Order Components (HOCs) specifically configured to run on top of a Grid middleware, and LooPo generates efficient parallel code, taking dependences between data and/or tasks into account. As a case study, we use the Farm-HOC, a component for running a dependence-free task farm on the Grid. We explain how applications that exhibit complex dependences can be executed using our Farm-HOC, when the application code is preprocessed with LooPo. A combination of LooPo and the Farm-HOC allows to run distributed computations on the Grid by sending a simple operation request via a Web service. The remote server hosting the Farm-HOC then uses LooPo to create ordered groups of independent tasks which are processed on the Grid in parallel.

Keywords: Grid computing, Loop parallelization, Higher-Order Components, LooPo

1. Introduction

The Grid combines distributed resources to a networked virtual supercomputer with new potential for high-performance applications. However, the numerous aggregate resources of the Grid are far more difficult to handle than the processors in a traditional cluster architecture. The major reasons for this difficulty are the high network latencies of Internet connections and the heterogeneity of the available resources. Unfortunately, most current approaches towards Grid application programming address only one of these two problems.

We address both, 1) the latency problem by grouping tasks at the servers using loop parallelization with tiling and 2) the heterogeneity problem, by using software components which do not run directly on a particular hardware platform, but on top of a middleware.

Tiling was developed in the area of automated loop parallelization to increase the performance of distributed applications [11]: it exploits locality in the Grid by adapting the task granularity of a parallel program which improves the performance by decreasing effects of network latency. However, the communication issues arising when heterogeneous servers are used for processing the individual tasks of an application are usually not addressed by the work in this area.

In contrast, the recent efforts in middleware research have a strong focus on communication protocols, service orientation and loose coupling between clients and servers: clients may switch between multiple servers for executing remote operations in a single application [18]. Examples are the initiatives of the Globus Alliance (`www.globus.org`) and the Unicore Forum (`www.unicore.org`), which make use of Web services and XML for representing the data being exchanged or stored in durable resources using a structured portable format. However, efficiency is not guaranteed by structuring data, which actually can imply a throttling of the application throughput. Moreover, it is not obvious that middleware simplifies application programming in all cases: while programmers are relieved of marshalling their data by using so-called service containers, one such container has to be configured for every server, which is often even more cumbersome than writing a fully self-contained program that does not rely on any container, tool or middleware.

This paper offers a novel approach based on automatically parallelizing software components, which combines the advantages of two research areas: a) automatic loop parallelization, which is a code optimization technique based on ordering tasks, and b) component-based programming with reuse of customizable code and configuration files.

Section 2 of the paper explains how sequential code can be transformed automatically into more efficient parallel code. Section 3 shows how components help programmers to benefit from a middleware without configuring it themselves for executing a particular application, and we explain how code parameters are used for customizing components. In Section 4, we show how code transformation techniques can be applied to code parameters, such that a farm component which originally supported only dependence-free applications can run code with dependences in parallel. Section 5 demonstrates the advantages of our approach using two example applications: DNA sequence alignment and solving linear equation systems. We conclude the paper and discuss related work in Section 6.

2. Loop Parallelization in 3 Phases

In the context of high-performance computing, the automatic parallelization of loop programs is well understood. A three-phase approach can be used to transform the input loop to a parallel target program in which the granularity of parallelism is adapted to the number of available processors and, independently, to the cost ratio of computation vs. communication [11].

In the first phase, the input program is analysed and the dependences are computed. In the second phase, all computations are mapped to logical execution steps and to virtual processors by computing a space-time mapping. 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.

2.1 Phase 1: Program and dependence analysis

The loop parallelization methods used here are applicable 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. Note that, for loop parallelization, the various instances of a statement, that are generated by surrounding loops, must be distinguished.

For loop programs as just described (or for affine recurrence equations), the dependences can be computed automatically. To this end, several methods have been implemented in LooPo. They are all based on integer linear programming. To test whether any two array accesses, for possibly different values of the loop counters, reference the same array cell, the integer linear programming problem to be solved consists of equations that ensure that both accesses reference the same array cell, and of inequations that ensure that both accesses are enumerated by the surrounding loops. The result is a function that takes any access to an array, together with the counters of the surrounding loops, and that tells which instance of which statement most recently accessed the same array cell [8, 4].

2.2 Phase 2: Parallelization

Space-time mapping is a technique used to extract parallelism from a loop nest. The parallelism is expressed by (piece-wise) affine functions mapping every instance of every statement to coordinates in space, i.e., on different virtual processors, and in time, i.e., to different logical execution times. These functions are called placement and schedule, respectively. The schedule has to respect the dependences; computations that are scheduled at the same logical time can be executed in parallel. The placement is done by a heuristic for locating instances of statements depending on each other on the same virtual processor.

2.3 Phase 3: Granularity adaptation

In most cases, the granularity of the parallelism generated is much too fine to be efficient. Thus, we apply tiling techniques to the space-time mapped program.

After space-time mapping, we have loops that enumerate the virtual processors, and other loops that enumerate logical time steps. Tiling the respective target loops modifies different parameters of the target program.

Tiling space loops means aggregating a set of virtual processors. All virtual processors that are covered by one tile are executed on the same processor. The tile size must be a compromise between much, but fine-grained and, thus, communication-intensive parallelism on the one hand, and little parallelism but also a small communication volume on the other. A cost model can be used to adjust the size with the ratio of communication cost and computation cost.

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 number of time steps that must be executed (since the receivers of messages are delayed if the send is postponed until the tile border is reached). Again, a cost model can be used to adjust the tile size with the ratio of communication startup cost and computation cost.

3. Task Farming on the Grid

A task farm is a recurring structure in parallel programming, encountered in many applications. Besides reusability, a task farm has the advantage that it addresses a drawback of parallel programs generated automatically, as described in the preceding section: for many input loops, the load in the generated parallel program is unevenly distributed among the processors. Frequently, the parallel program first has a phase of increasing parallelism, and afterwards, sometimes immediately following, a phase of decreasing parallelism. Thus, if we exploit all existing parallelism, i.e., if we use as many processors as there can be used maximally, we end up with a theoretically maximal efficiency of 50%. The simple example of Section 5 is such a case. The only way to solve this problem is to reduce the number of processors used.

Tiling alone cannot improve the situation: affine functions, which are the mathematical basis of tiling, cannot map different tiles to the physical processors in a round-robin fashion, or in any other way so that the number of tiles to be mapped to a physical processor changes with the available parallelism. However, this kind of load balancing is one of the strengths of task farming.

Task farming can be applied to procedures with or without input data. To describe farming as general as possible, we introduce the term *workspace* which refers to the memory holding the data we operate on, regardless whether this data is provided by the user or generated at runtime. Parallelism is achieved in a task farm by applying the same code to distinct regions of the workspace. Thus, one can think of a task as a part of the workspace: a task is unambiguously defined by a data item, while, in some applications, a task definition may consist of the combination of data plus some individual code.

One process called `Master` divides the workspace into independent pieces. The resulting tasks are passed to multiple `Worker` processes, which apply the same procedure simultaneously to all data. Implementations of the task farm range from skeleton programs for specific parallel architectures [3] to scalable versions [16], in which the `Worker` processes are distributed across multiple networked machines. In the following, we describe our particular implementation of a task farm for the Grid, the Farm-HOC.

3.1 Handling specific requirements of the runtime platform

Grid software [9], and especially reusable component software [17] for the Grid, is typically integrated with a middleware which builds a uniform layer of abstraction above the Grid hardware. Due to the heterogeneity of the hardware resources in the Grid, any data transmission over the network must be handled in a portable format. Therefore, a modern Grid middleware, e. g. , the popular Globus Toolkit [9], usually requires from the application code to handle the exchange of data over the network using Web services. The middleware provides a Web service runtime environment consisting of programs for generating *service ports* from interface descriptions and of *containers*, which encode all data transmissions over the network into XML documents in the SOAP format. The advantage of using Grid middleware is that the computers interconnected via Web services can be of various architectures and can run code written in different programming languages, while only one computer must provide a service port to its communication partners (which either run a standalone client or host other Web services and access the remote service port from their own containers). The middleware programs used to automatically generate a service port require an interface description (in WSDL format) which is a part of the configuration that must be written by the user.

Thus, middleware users achieve data portability and code interoperability across the Grid, at the price that every remotely accessible component requires a configuration of the services used for accessing them. This configuration consists at least of two files, one WSDL file describing the remote interface and one WSDD file containing the deployment description. The WSDL file describes the component's access points (called *service operations*) and the formats of the data the component accepts and emits. In the WSDD file, the operations defined in the interface are mapped to the corresponding binary files which contain the actual code required for performing these operations when they are requested by a client or another Web service. The configuration of Web services for accessing Grid components includes even more than the description of interfaces and binaries, e. g. , the Globus-typical resource property configuration and a declaration of the types of the messages, which can be exchanged with the Web services asynchronously. These messages must be explicitly defined as *notifications*, since Web services usually process all operations synchronously with the program requesting them.

Our Farm-HOC [10] is an implementation of a task farm as a Grid-enabled component, i. e. , it includes, beside the task farm implementation code, the required configuration files to access the farm remotely via a Web service hosted in the Globus container. The implementation relies on a *code mobility* mechanism which allows clients to send to the Farm-HOC via its service port data and executable code. This mechanism is enabled by our Higher-Order Component Service Architecture (HOC-SA) [6] consisting of a *code service* for maintaining a database of code units shared among Grid servers and a *remote code loader*, used for uploading and downloading code from the code service.

The Web service used to access the Farm-HOC accepts parameters carrying executable code. Customizing code parameters specify the application-specific part of the `Master` and `Worker` behavior, e. g. , for running a particular filter algorithm on string data using multiple servers of the Grid, as required in the data-intensive applications of computational molecular biology. In this example, one `Master` code parameter is uploaded to specify how the genome data is partitioned and the `Worker` parameter carries the code for running the actual filter on each of the single partitions. Once the Farm-HOC starts processing input, it invokes our remote code loader, which instantiates on the server side an object providing a method for running the code uploaded by the client. This way, the `Master` and `Worker` parameters can be used as customizing parameters of the Farm-HOC, while standard Web services usually do not support the exchange of code over the network.

3.2 The Treatment of Dependences

Many customizing code parameters for the Farm-HOC (explained above, in Section 3.1) is that they introduce dependences between tasks, as one piece of data must be available before another one can be computed. Task farming, in contrast, implies a space-time mapping (see Section 2) where all tasks are scheduled at the same logical time and are placed separately or in groups (according to the tiling) onto different processors. This only works, if there are no dependences between the tasks.

In [7], an adaptation of the Farm-HOC is suggested: the workspace regions get ordered, such that computations follow a pattern, wherein the parallel processing of all elements in a subset of the workspace is possible via farming. Thus, the farm is transformed into a sequence of farms. This adaptation is enabled via our code mobility mechanism as follows: one code parameter uploaded to the Farm-HOC performs the initial ordering of the tasks (e. g. , regions of a matrix) and arranges them accordingly for their parallel execution. Due to the fact that this ordering must comply with nothing else than the task dependences, which are fully described by the `Master` and `Worker` code parameters passed to the Farm HOC, our adaptation of the Farm-HOC does not require writing any additional code or dependence description from the user. In the following section, we show how a task ordering, if one is required due to dependences, can be automatically derived by the loop parallelizer LooPo from the code parameters passed to the Farm-HOC.

One may argue that many of the possible patterns of parallelism that can be described by farming plus task ordering, i. e. , any composition of farming phases with variable sizes (e. g. , a systolic schema [13]) are fairly different from standard farming. Some users may prefer to have a more extensive collection of different HOCs, instead of using the Farm-HOC whenever possible and implicitly deriving a new behavior from it, when needed. However, when deriving new HOCs from the Farm-HOC, programmers benefit from the fact, that these HOCs share the same interface which takes two code parameters, the partitioning `Master` and the computation `Worker`. Moreover the data input and output format is the same for any farming program, namely data arrays representing, e. g , strings or matrices. The complexity of the necessary configuration specifying the required interface on top of a Grid middleware, including the setup for converting code to data for its transmission and vice versa for its execution, strongly motivates the use of the Farm-HOC, even for parallel computations different from the standard farming procedure.

4. Teaching the Task Farm Component to Parallelize Loops

Our approach is to apply loop parallelization methods (see Section 2) to the code parameters of the Farm-HOC (see Section 3) allowing it to deal with temporal dependences. The idea is to resolve the dependences between the tasks being processed in parallel by grouping and ranking the task partitions (unordered workspace regions) created by the `Master`. The ranking defines a sequence of groups where all the partitions in one group contain only tasks which are independent from the tasks in other partitions of the same group and only depend on each other or on the tasks in partitions which belong to a group with a lower rank. Following this sequence, the parallel `Workers` can be applied to the partitions of all groups in rank order.

We automate the detection of dependences required for the group ranking: starting with the description of the `Worker` code parameter, LooPo computes the dependences and the schedule (i. e. , the groups and their ranks), and eventually generates a loop nest enumerating the tasks in the right sequence. If desired, the tile size, which has been introduced as a user parameter (the partition size of the original farm), can be kept and used by LooPo. Alternatively, LooPo can compute the tile size automatically, when the user (or the Grid monitoring system [5]) provides a cost model and its architecture parameters. Within the scope of an ongoing CoreGRID fellowship, LooPo will be integrated with the Farm-HOC to allow not only for farming but also different parallelism structures [12] on top of the latest Globus middleware.

5. Example Applications

As a very simple example of an automatically derived task ordering for our Farm-HOC, we consider a loop program that is used for computing the distances of DNA sequences, i.e., the total number of the required transformations needed to transform one sequence into another [14]. A loop nest for computing a matrix of distance values (the *alignment matrix*) is as follows:

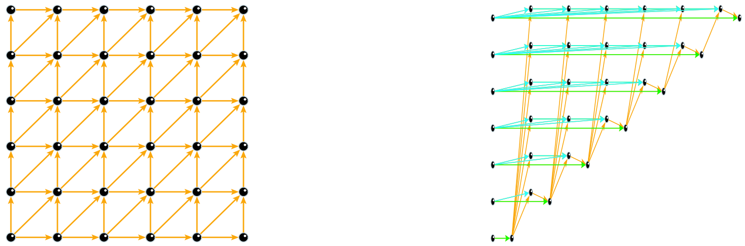
```
1: for (i=0; i<n; i++)
2:   for (j=0; j<m; j++)
3:     s[i,j] = max( s[i,j-1]+plt,
4:                   s[i-1,j-1]+delta(i,j),
5:                   s[i-1,j]+plt )
```

In this code, $delta$ can be interpreted as a side-effect-free function or as a read-only array that represents the (non-)difference between one character at position i and the other character to be compared to at position j; plt is a constant representing a (negative) penalty for mismatch.

Due to the assignment in line 3–5, every matrix element depends on its north, northwest and west neighbor and, therefore, the code must be adapted before it can be executed in parallel via farming. To order the tasks automatically, the above code is processed with LooPo which derives the dependences and a schedule, as discussed in Section 2. Here, the resulting schedule is $\theta(i,j) = i + j$. Furthermore, LooPo can be used to generate code that enumerates sequentially all virtual time steps t that are in the image of θ, together with a loop enumerating all instances i that are scheduled at time t. This loop on i (line 3–6 in the code shown below) is then free of dependences and can be forwarded to the Farm-HOC, described in Section 3, for parallel execution on the Grid.

The parallel code, automatically generated by LooPo, is represented in a format that supports parallel loops using, e.g., the `FORALL` keyword of HPF, or the `pragma omp parallel` directive of OpenMP as shown here:

```
1: for (t=0; t<=m+n; t++)
2:   #pragma opm parallel
3:   for (i=max(t-m,0); i<=min(t,n);  i++)
4:    s[i][t-i] = max( s[i][t-i-1]+plt,
5:                     s[i-1][t-i-1]+delta(i-1,t-i-1),
6:                     s[i-1][t-i]+plt );
```



(a) Uniform flow dependences

(b) Non-uniform dependences

Figure 1. Examples of automatically detected dependences for two applications

For running such code on top of the standard Java virtual machine (as required by the Farm-HOC implementation described in [7]), it must be converted to Java by inserting the body statements of each parallel loop into `Runnable`-objects, holding also the initialization code to provide an appropriate thread pool setup. Obviously, the length of the parallel code is increased by such conversions, but they can be fully automated and run within LooPo.

Besides code in various formats, LooPo can also generate an intuitive graphical representation of the dependences detected in an application: In the diagrams in Figure 1 (a) and (b), each point corresponds to an instance of a statement executed inside a loop, e. g. , the critical assignment in Line 3–5 of our above example. The points are positioned along axes which correspond to the loops surrounding the statements, as described in [2]. Dependences are depicted as arrows. Figure 1 (a) illustrates a run of the sequence alignment application for two equally sized input sequences of length 6 (The application illustrated in Figure 1 (b) is described in the following). As can be seen, only the statements on the antidiagonals are independent and can be processed in parallel.

In the context of matrix algorithms, DNA sequence alignment is one of the simplest cases, since it is a perfect loop nest and it has uniform dependences. Our example is a single matrix computation that takes place at every matrix entry. However, neither the Farm-HOC nor LooPo are limited to such simple matrix algorithms.

As a more complex example with imperfectly nested loops and non-uniform dependences that are not obvious from the code, we study the last step of solving a linear equation system: the backward substitution of the variables which is performed after the coefficient matrix has been converted to a triangular form. In this application, the code parameter passed to the Farm-HOC looks as follows:

```
1:    for (k=0; k<=n-1; k++) {
2: S:    sum[n-k] = b[n-k];
```

```
         for (l=0; l<=k-1; l++)
   T:       sum[n-k]=sum[n-k]-a[n-k][n-l]*b[n-l];
   U:    b[n-k]=sum[n-k]/a[n-k][n-k];  }
```

LooPo computes the schedule $t_S(k) = 0$, $t_T(k, l) = 2*l+2$, and $t_U(k) = 2*k+1$; a loop nest enumerating the tasks that must be processed is then derived and processed in parallel on the Grid by the Farm-HOC. The dependencies in this example are shown in Figure 1 (b).

6. Conclusion

We have shown that a combination of components with loop parallelization simplifies the programming of Grid applications in two respects. First, the programmers can use components without caring about dependences between tasks and/or data. Second, automatic parallelization benefits from load balancing which is implicitly incorporated within components, such as the Farm-HOC.

Our work represents one of the current approaches towards programming efficient applications for the Grid without dealing with the full complexity of their distributed execution. A related project is the integration of the KOALA scheduler [15] with HOCs for a user-transparent, automated choice of execution machines [5]. While scheduling allows to increase the performance of an application by adapting the environment, i. e. , switching between resources, we have shown how to reduce execution times by adapting the application code. Both approaches are complementary and can be combined to achieve even higher efficiency.

Once the integration of LooPo into the Farm-HOC is finished, we are planning to conduct a performance evaluation for various applications, including a comparison between LooPo-generated and hand-tuned parallel code on the Grid.

Acknowledgments

This research was conducted within the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] E. Argollo, D. Rexachs, F. Tinetti, and E. Luque. Efficient execution of scientific computation on geographically distributed clusters. In *PARA*, pages 691–698, 2004.

[2] C. Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory, LNCS 715*, pages 398–416, Hildesheim, August 1993.

[3] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[4] J.-F. Collard and M. Griebl. A precise fixpoint reaching definition analysis for arrays. In *Languages and Compilers for Parallel Computing*, LNCS 1863, pages 286–302. Springer, 1999.

[5] C. Dumitrescu, D. H. J. Epema, J. Dünnweber, and S. Gorlatch. User-transparent scheduling of structured parallel applications in grid environments. In *HPDC-15, France*, 2006, *to appear*.

[6] J. Dünnweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE SCC04, China*, pages 288–294. IEEE Computer Society Press, 2004.

[7] J. Dünnweber, S. Gorlatch, S. Campa, M. Danelutto, and M. Aldinucci. Using code parameters for component adaptations. *Integrated Research in Grid Computing*, Springer Verlag, 2005, *to appear*.

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

[9] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputing Applications*, 15(3), 2002.

[10] S. Gorlatch and J. Dünnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In *Future Generation Grids*, p. 241–263. Springer Verlag, 2005.

[11] M. Griebl, P. Faber, and C. Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, Mar. 2004.

[12] C. A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2000.

[13] O. H. Ibarra, T. Jiang, J. H. Chang, and M. A. Palis. Systolic algorithms for some scheduling and graph problems. In *J. of VLSI Signal Processing, Volume 1, Issue 4*, pages 307–320, 1990.

[14] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Dokl., Volume 10 (8)*, pages 707–710, 1966.

[15] H. Mohamed and D. Epema. The design and implementation of the KOALA co-allocating grid scheduler. In .-. LNCS 3470, *Proceedings of the European Grid Conference, Amsterdam*, 2005.

[16] M. Poldner and H. Kuchen. Scalable farms. In *Proceedings of the International Conference on Parallel Computing, Malaga, Spain*, September 2005, *to appear*.

[17] C. Szyperski. *Component software: Beyond object-oriented programming*. Addison Wesley, 1998.

[18] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.