European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

LooPo-HOC: A Grid Component with Embedded Loop Parallelization

Johannes Tomasoni, Jan Dünnweber, and Sergei Gorlatch
{jtomasoni | duennweb | gorlatch}@uni-muenster.de
Department of Computer Science
University of Münster

Michael Claßen, Philipp Claßen, and Christian Lengauer
{classenm | classen | lengauer}@fim.uni-passau.de
Department of Informatics and Mathematics
University of Passau

CoreGRID Technical Report
Number TR-0135
April 9, 2008

Institute on Programming Model

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

LooPo-HOC: A Grid Component with Embedded Loop Parallelization

Johannes Tomasoni, Jan Dünnweber, and Sergei Gorlatch
{jtomasoni | duennweb | gorlatch}@uni-muenster.de
Department of Computer Science
University of Münster

Michael Claßen, Philipp Claßen, and Christian Lengauer
{classenm | classen | lengauer}@fim.uni-passau.de
Department of Informatics and Mathematics
University of Passau

CoreGRID TR-0135
April 9, 2008

Abstract

This work integrates two distinct research areas of parallel and distributed computing, (1) automatic loop parallelization, and (2) component-based Grid programming. The latter includes technologies developed within CoreGRID for simplifying Grid programming: the Grid Component Model (GCM) and Higher-Order Components (HOCs). Components support developing applications on the Grid without taking all the technical details of the particular platform type into account (network communication, heterogeneity, etc.). The GCM enables a hierarchical composition of program pieces and HOCs enable the reuse of component code in the development of new applications by specifying application-specific operations in a program via code parameters. When a programmer is provided, e. g. , with a compute farm HOC, only the independent worker tasks must be described. But, once an application exhibits data or control dependences, the trivial farm is no longer sufficient. Here, the power of loop parallelization tools, like LooPo, comes into play: by embedding LooPo into a HOC, we show that these two technologies in combination facilitate the automatic transformation of a sequential loop nest with complex dependences (supplied by the user as a HOC parameter) into an ordered task graph, which can be processed on the Grid in parallel. This technique can significantly simplify GCM-based systems which combine multiple HOCs and other components. We use an equation system solver based on the successive overrelaxation method (SOR) as our motivating application example and for performance experiments.

1 Introduction

We demonstrate the benefits of using software components together with loop parallelization techniques for Grid programming. In recent years, component technology [22] has reached wide-spread acceptance for the development of large-scale distributed software infrastructures. Almost no project that requires an interconnection of multiple resources, e. g. , databases, compute clusters and Web clients, is started from scratch anymore. Developers rather rely on modern component frameworks, which provide them with reusable implementations of the functionality needed for their applications. This approach to code reuse goes much further than traditional libraries, since frameworks usually provide not only executable code but also the required configuration (i. e. , setup descriptions, typically in the form of XML files) for deploying the components in the target context. This context may be, e. g. , a middleware

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

like Globus [19] for interconnecting multiple components and remote clients, across the boundaries of heterogeneous hardware and software.

In the CoreGRID community [24], the *Grid Component Model* GCM [16] has recently become the commonly agreed reference specification of software components for the Grid. The GCM combines efforts of multiple CoreGRID partners, e. g. , the GCM predecessor *Fractal* [1] whose principle of hierarchical composition has been adopted, the ProActive library [3] for asynchronous communication among components, and Higher-Order Components (HOCs [17]) that accept as input not only data but also pieces of code supplied via an Internet connection.

For assisting programmers in building parallel applications of Grid components, this work combines HOCs with the automatic loop parallelization tool LooPo [7]. The idea is to apply the LooPo loop parallelization mechanism to HOC parameters, i. e. , code pieces supplied to a HOC as parameters. These parameters often carry a loop nest to be executed by some worker hosts (i. e. , any free processing nodes) in the Grid. A typical example is the Farm-HOC, implementing the popular master/worker pattern for running a set of independent tasks [17]. The original Farm-HOC is not able to deal with inter-task dependences: they would make it necessary either to design a new HOC which takes the dependences into account or to remain with a sequential, less efficient solution. Instead of requiring the developer to build one new HOC per possible dependence pattern, we suggest a more flexible component, called LooPo-HOC, which embeds the LooPo loop parallelizer [20].

Dependences in code parameters of the LooPo-HOC in the form of nested loops are automatically resolved: code parameters (the loop nests) are transformed into an ordered task graph. The processing pattern employed by the LooPo-HOC can be viewed as an adapted farm whose master schedules the tasks as specified by this graph.

In our previous work, we suggested to combine HOCs with LooPo [13] and discussed a farm implementation version for processing task graphs [18]. This paper presents the implemetation of the LooPo-HOC plus application examples and performance experiments.

HOCs use a Web service-based code transfer technology that extends the Globus middleware [19] by the *Code Service* and the *Remote Code Loader* (both are available open-source, within the scope of the Globus incubator project HOC-SA [15], see Fig. 1). The Code Service and Remote Code Loader can be viewed as an add-on to the Globus Resource Allocation Manager WS GRAM [10]. Their purpose is facilitating software components, which hold the code for solving recurring problems and expect the user to supply only application-specific code pieces via the network. The Code Service and the Remote Code Loader support the transfer of such code pieces across the network [12]. In contrast, programmers using only GRAM are supposed to transfer their programs on the whole rather than in pieces, which limits the potential of code reuse in component-based software architectures.

The structure of the paper is as follows. In Section 2, we introduce the LooPo-HOC: Section 2.1 introduces the `mwDependence` service which implements our adapted version of the master/worker processing pattern, for executing ordered task graphs (taking dependences into account). Section 2.2 explains the automatic parallelization mechanism of LooPo. Section 2.3 describes the challenges of integrating LooPo into a Grid-aware component and how we addressed them in the LooPo-HOC. Section 2.4 shows how the internal workload monitor of the LooPo-HOC works. Section 3 introduces an example application: the parallel SOR system solver. Performance measurements are presented in Section 4, and we draw the conclusions from our work in Section 5.

2 Implementation of the LooPo-HOC

The LooPo-HOC is composed of LooPo itself for transforming code (Section 2.2), the Web service for clients to connect (Section 2.3), controller software for task queue management and workload monitoring (Section 2.4), and an internal farm implementation for running the actual application tasks. These parts are available to the client via a single Web service, as shown in Fig. 1.

2.1 The Internal Compute Farm of the LooPo-HOC

To explain how the compute farm in the LooPo-HOC works, let us briefly recall the functionality of the Farm-HOC [17] and explain the setup shown Fig. 1: clients upload (sequential) application code to a central Web service. This service is provided by the master server which stores the code at the Code Service where it is assigned a key and saved in a database (using OGSA-DAI [25]). Clients can send the master a request to reload the code and run it on multiple remote worker nodes for processing data in parallel. The master controls the distributed computations without requiring the user to be aware about the number of involved workers and the (Web service-based) communication between itself and the workers.

The compute farm in the LooPo-HOC differs from a common compute farm implementation for Grids [2] in two ways:

1. the LooPo-HOC embeds, besides a farm of workers, the LooPo tool and uses it for ordering tasks in the form of a task graph taking dependences among tasks into account. The farm executes the tasks according to this order, freeing the user from dealing with task dependences.
2. the communication does not rely on a single protocol, but to increase the efficiency, a Web service is used only as the remote interface for supplying input to the farm via an Internet connection. All internal communication (between master and workers) is handled using a light-weight protocol, specifically developed for this component which is a distributed, version of MPI, supporting all the basic and most of the collective operations, using only Java and TCP sockets [5].

The LooPo-HOC offers a universal farm implementation for Java code [2], i. e. , this farm is capable of executing applications without dependences as well (and has shown almost linear speedup in various experiments) It is included in the open-source distribution of the HOC-SA [15] in the package `mwDependence`.

The worker nodes in Fig. 1 are fully decoupled from each other, i. e. , they need no communication between each other, and are supposed to run in a distributed environment. In the following, we describe in more detail the transformation process, the scheduling and the workload monitoring, which make up the core of the LooPo-HOC and which are supposed to run locally, on the same server, ideally on a multiprocessor machine.

2.2 Transforming Loop Nests into Task Graphs

For the automatic parallelization of loop nests, LooPo uses a mathematical model, the so-called *polytope model* [20]. In this model, affine linear expressions are used to represent loop iterations, dependences and accesses to array elements. LooPo is an implementation of various methods and tools for analyzing a given loop program, bringing it into model representation and performing a dependence analysis and the actual parallelization using integer linear programming. The result of the code transformation done by LooPo is a task graph in which groups of independent tasks are arranged in a sequence.

For the automatic parallelization of loop nests using LooPo, there are a number of steps involved, as follows [18].

The first step is to analyze the input program and bring it into the polytope model representation. This is done by analyzing the (affine linear) expressions in loop bounds and array accesses. The resulting model consists of one so-called *index space* per statement. The index space contains the coordinates, i.e., the values of the loop variables, of all steps in which the statement is executed. LooPo keeps track of all array accesses and computes the resulting data dependences.

In the second step, we use mathematical optimization methods to compute two piecewise affine functions: the *schedule* maps each computation to a logical execution step, and the *placement* maps each computation to a virtual processor. The objective is to extract all available parallelism, independently of any machine parameters, e. g. , the number of processors. The result of this step is the so-called *space-time mapping*.

In order to adjust the granularity of parallelism to a level that is optimal for task farming (our method for the distributed execution of the parallel tasks, as discussed in Section 2.1), the *tiling* technique is used in the third step to aggregate time steps and virtual processors into larger chunks, called *tiles*.

Each tile produced by LooPo represents a *task* for the LooPo-HOC and contains the corresponding set of computation operations for the time steps and virtual processors that were aggregated. Information about data dependences between tasks is stored in the form of a *task graph* that is used by the master for scheduling them, i. e. , to choose an order of execution between dependent tasks. Thus, the master is responsible for arranging the execution order, whereas the target processor for the execution can be determined using an advanced scheduling system [11] to exploit task locality. In Grid environments which do not provide a scheduling system with tunable policies (e. g. , KOALA [11]), users of the LooPo-HOC can also directly adapt the master, such that the complete scheduling is handled there. This way, programmers can, e. g. , arrange chains of tasks that should be executed on the same worker. For data dependences between tasks that make the exchange of computed data elements necessary, the master provides a method to *join* a new (dependent) task with a finished task. This way, the dependent task is decoupled from its predecessor, gets the updated data and is scheduled for execution.

2.3 Integration of the LooPo-HOC with the Middleware

Beside the workers (executing the single tasks, as described in Section 2.1) and the master (running LooPo, as described in Section 2.2), the LooPo-HOC comprises a Web service for remote access and a resource configuration for maintaining the distributed application state (status data and intermediate results), as is typical in the Web Service Resource Framework (WSRF) [19].

While the service interface itself is stateless, the resources connected to it (as configured in a setup file) hold their state (in the form of transient variables, called *resource properties* in WSRF [19]) even past the scope/duration of a session. The LooPo-HOC makes use of this feature, e. g. , for parallelizing a loop nest and preserving the resulting task graph as a data record in a resource, which can be referenced by a key and reused in multiple applications.

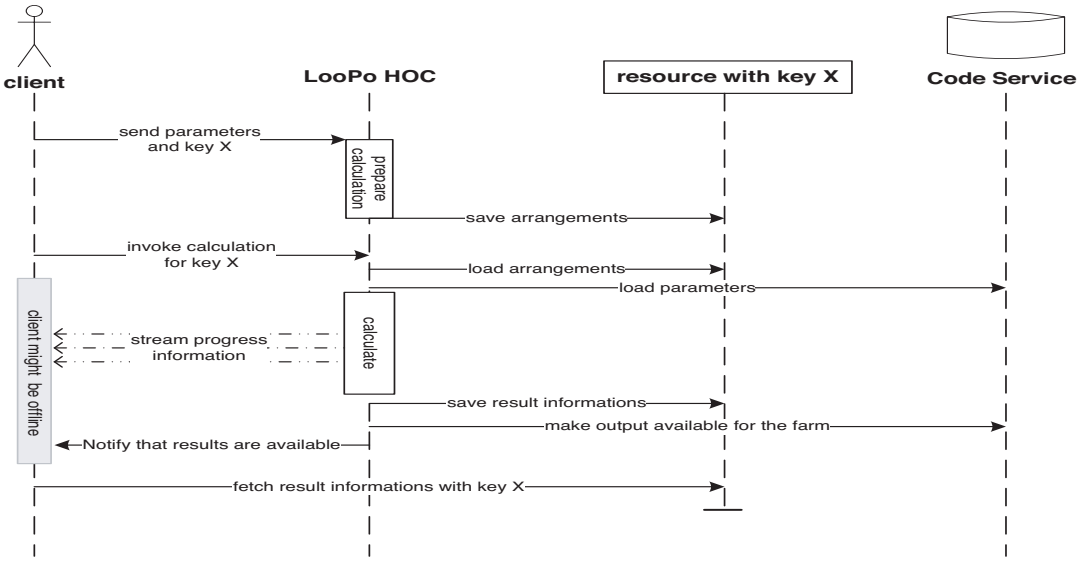

Figure 2: Sequence Diagram for using the LooPo-HOC

Another feature, through which the LooPo-HOC benefits from the WSRF middleware, is its support for asynchronous operations. While LooPo transforms loop nests, the client can disconnect or even shut down. The LooPo-HOC can restore the task graph from a former session, when the client sends it the corresponding resource key. The LooPo-HOC uses two types of WSRF resources. For every code transformation request, one new resource instance (i. e. , transient storage) for holding the resulting task graph is created dynamically. The other resource is static (i. e. , instantiated only once and shared globally among all processes), called *monitor* and explained in Section 2.4.

The task graph resources are instantiated following the factory pattern, returning a unique remote reference (the resource key) to the client. As shown in Fig. 2, the client sends the resource key on every communication with the

LooPo-HOC, which uses the key afterwards to retrieve the corresponding resource data (the task graph and intermediate results). Thus, a LooPo-HOC server is not a single point of failure, but rather a service provider that permits clients to switch between mirror hosts during a session.

2.4 Workload Monitoring in the LooPo-HOC

The transformation of loop nests into tasks graphs is a computation-intensive operation, which is quite unusual for Web services: typically, a Web service operation retrieves or joins some data remotely and terminates immediately. Due to the asynchronous operations of the LooPo-HOC, the clients produce processing load right *after* their requests are served, since this is, when the code transformations begin (concrete time costs follow in Section 4).

From the user's viewpoint, the asynchrony is advantageous, since local application activities are not blocked by the code transformations running remotely. However, when multiple users are connected to the same LooPo-HOC server, the workload must be restricted to a certain (server-specific) maximum of concurrrent requests. For this purpose, the LooPo-HOC has an integrated workload monitor (see Fig. 3) which provides status information to the clients.

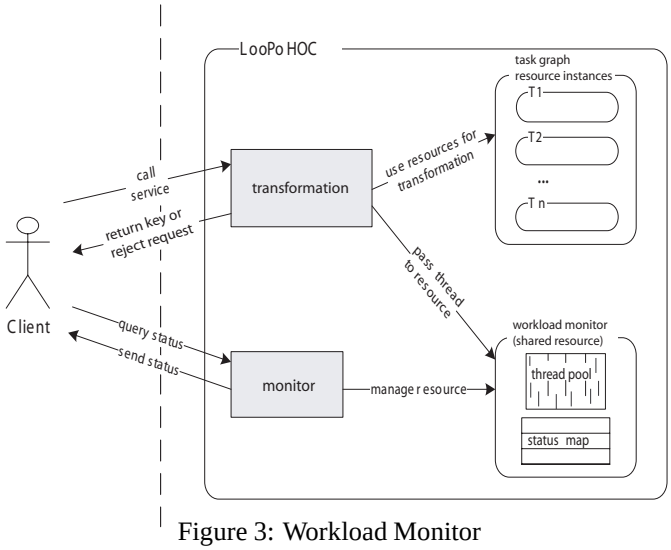

Figure 3: Workload Monitor

The monitor consists of two parts, a fixed-size thread pool and a status map. For every transformation, the LooPo-HOC first checks if an idle thread is available. If the thread pool is fully loaded, then the LooPo-HOC creates a new transformation thread and adds it to the pool. The maximum threshold for the thread pool is set by the server administrator and is usually equal to the number of CPUs of the hosting server. Once the number of executing threads has reached this maximum, incoming requests are queued.

The status map (shown bottom right in Fig. 3) is a structured data store, used to keep track of the successive transformations. The client can read the map by issuing an XPath query [9] to the monitor at any time. This feature is useful when the client reconnects during a loop transformation. The map also allows one application to execute the tasks resulting from transforming the sequential loops submitted by another application: via the map, users can track the status of transformations (and run the resulting tasks), even if they connect to the LooPo-HOC for the first time (and, consequently, receive no automatic notification about status updates). This scenario arises, e. g. , if the Web service for connecting to the LooPo-HOC is deployed to multiple servers, allowing clients to switch between hosts, when a connection fails or some host's request queue is full.

As future work, we are considering to use the workload monitor and the status map, for automatically balancing workload: instead of queuing requests that exceed some threshold, another server will take over the processing load. Implementing load balancing this way is probably also an interesting case study for on-demand deployment of HOCs and combining multiple code transfer technologies [12].

3 Case Study: The SOR Equation System Solver

As an example application, we have implemented a solver for linear equation systems, $A\phi = b$, using the successive overrelaxation method (SOR). The SOR method works by extrapolating the Gauss-Seidel method [6], as shown in the following iteration formula:

$$\phi_i^{(k+1)} = (1-\omega)\phi_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\phi_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\phi_j^{(k)}\right)$$

Here, vector $\phi^{(k)}$ denotes the kth iterate, the a_{ij} are elements of the input matrix A, and ω is called the *relaxation factor* which is reduced in each iteration, until it declines below some tolerance. Roughly speaking, the SOR algorithm computes weighted averages of iterations, leading to a triangular system of linear equations, which is much simpler

to solve than the original arbitrary system. There is one control dependence in the SOR solver, i. e. , in each pair of successive iterations the follow-up statement depends on its predecessor.

```
1: @LooPo("begin loop", "constants: m,n; arrays: a{n+1}")
2: for (int k = 1;  k <= m;  k++)  {
3:    for (int i = 2;  i <= n - 1;  i++)  {
4:        // average computation
5:        a[i] = (a[i - 1] + a[i + 1]) / 2.0; ... } }
6: @LooPo("end loop")
```

Figure 4: The sequential code parameter

To run this application in parallel on the Grid, the user supplies the LooPo-HOC with the application name (here, `SOR`) and a sequential description of the computations expressed in Java notation, as shown in Fig. 4. Any loop nest (with metadata, as the delimiting annotations in lines 1 and 6) can be used as input for the LooPo-HOC. The annotations have the purpose of delimiting the code that is automatically parallelized. The parallelization itself is applied to the Java source code using the steps from Section 2.2, resulting in a task graph. First, a model of the input program is derived (Fig. 5). For the space-time transformation, the following schedule θ and placement π were determined: $\theta(k, i) = 2 * k + i$ and $\pi(k, i) = k$.

For obtaining the task dependence graph, tiling is applied on the transformed target program. Fig. 6 shows the representation of the transformed program after tiling is applied using the same color tone for tiles that can be executed independently. The final model is derived by joining each tile as a node into a graph with every inter-tile dependence as a directed edge in that graph. From the task graph model representation, the LooPo-HOC generates three Java classes

Figure 5: The input program (M=7, N=5) Figure 6: Transformed program after tiling

as output which are stored in the Code Service [17]: `SORMaster`, `SORData` and `SORTask`. The `SORMaster` holds the dependence graph (it implements the interface `SchTaskDependence` from the `mwDependenceService` package) and provides the `join`-method (required for distributing data to task groups; see Section 2.2). `SORData` objects are used for buffering application data and the `SORTask` class describes a single task (as an implementation of the `execute` method from the interface `mwDependenceService.UserTask`).

Since these files comply with the interface definitions in the HOC-SA [15], the user can directly load the three output files for parallel processing as code parameters of the farm described in Section 2.1.

4 Experiments

Fig. 8 shows the computation times for matrices of different sizes using from 1 to 10 workers running on common 1.7 GHz PCs. We also experimented in a more heterogeneous environment (and observed promising speedups), but here, we only report the most regular results (for homogeneous workers), since these results are the most comprehensible ones.

The experimental environment was set up in a high-performing network reaching a data throughput of approximately $3.4MBit/s$. The strong time decay in the left of Fig. 8 (from 1 to 3 workers) shows that especially the adding of the first 2 workers leads to a strong performance improvement, as compared to the sequential computation time. The corresponding efficiency values support this assumption: for 3 workers, the efficiency was above 80% and for 2 workers even around 90% in multiple measurements.

The decline of the plane along the z axis (matrix size) shows that using more than 5 workers is only profitable for large matrices, while, for the $100K \times 200K$ matrix there are not enough tasks (using a 5×5 tiling [18]) to take advantage of more than 4 workers.

The eight bars in Fig. 9 represent the initialization times for 10 workers (i. e. the time that passes by after the client sends a request, until the remote computations in the farm start). The time required to establish an `ssh` connection

Figure 7: Computation Times

Figure 8: Initialization Times

between the master and all workers varied between 4 and 5 seconds. As can be observed by comparing the bars in the front row and the back row, there is no correspondence between the time required to connect and the full initialization time (including the remote code loading), which exhibits strong variations between 30 and 90 seconds (the standard deviation σ from the mean value of 50 seconds was 22). This is due to the connection between the farm and the database: as explained in Section 2.1, the farm workers load the code for processing the single tasks from the Code Service using OGSA-DAI [25], which is known to deliver unreliable performance under certain conditions, especially when it is deployed on a single server together with other Web services [14]. In relation to the much longer computation times of the SOR application (from several minutes to several hours for large matrices), the initialization time can, thus, be disregarded. It should also be noted that the initialization is only performed once per worker and application. After the first set of input data (a matrix in the SOR example) has been processed, the same parallel code is used to process any number of successive inputs without repeating its generation (using LooPo) and its transfer from the Code Service to the workers.

The transformation of the single loop nest used in the SOR example in Section 3 takes approximately 1 min on a contemporary dual-core PC, utilizing 50% of its overall CPU capacity. From this quick increase of computational load, we conclude that, if only one server is used to run the code transformations in multiple different applications of the LooPo-HOC, this machine should be a powerful multiprocessor server.

5 Conclusion

The idea of using LooPo for transforming the code parameters of a HOC was suggested in an earlier paper [13] and a prototype of such a component was tested for local area networks [5]. By now, the implementation of a Grid-aware version, called LooPo-HOC, has been completed and extended on the server side: instead of a farm, that supports only dependence-free applications, the distributed master/worker implementation, described in Section 2.1, now provides a distributed environment for Java programs that is capable of processing dependent tasks using a task graph scheduler. Using the LooPo-HOC, the treatment of dependences becomes fully transparent, i. e. , the Grid application programmer is no more responsible for scheduling independent task groups [13], but there is an internal scheduler in the master.

Using the SOR program from Section 3 as an example, we have shown that the LooPo-HOC provides a promising scalability and the time needed for the initial code transformations does not critically impact the overall application performance.

Another approach to automating the generation of parallel code was developed within the recent research on OpenMP programs and *reparallelizing* them for the Grid [8]. This work also covers Java programs and the use of *distributed shared memory* (DSM) for data exchange among tasks, but still requires from programmers dependence-free input and the explicit declaration of parallel loops via OpenMP directives. The LooPo-HOC, on the contrary, offers a fully transparent programming interface that requires only sequential code. The required data sharing could have been implemented using Sun's standardized DSM implementation in *JavaSpaces* [4]. However, the LooPo-HOC requires only the joining of single tasks and no support for distributed transactions, and, thus, relies on a more light-weight implementation [5], which provides much better performance.

The LooPo-HOC (including the source code) can be downloaded from the Internet as a part of the HOC-SA Globus incubator project [15]. It is interoperable with any other Globus-based Grid software. For integrating the presented parallelization technique into the GCM, the task graph may also be included into an automatic manager in the membrane of a GCM component [16]. The suggested combination of components with loop parallelization is not only useful for the GCM, but also for other popular component models, such as CCA [23] and CCM [21]. Beside the code transfer mechanism used by HOCs [12], no other special features of this component technology are required.

References

[1] Institut National de Recherche en Informatique (INRIA), *The Fractal Web Site*, 2007. http://fractal.objectweb.org.

[2] Marco Danelutto, *Task Farm Computations in Java*, International Conference on High-Performance Computing and Networking, Amsterdam, NL, 2000, pp. 385–394.

[3] INRIA, *The ProActive Web Site*, 2007. http://www-sop.inria.fr/oasis/ProActive.

[4] 1994–2007 Sun Microsystems, *The JavaSpaces Specification*. www.sun.com/software/jini.

[5] Eduardo Argollo, Michael Claßen, Philipp Claßen, and Martin Griebl, *Loop Parallelization for a Grid Master-Worker Framework*, CG Integration Workshop Heraklion, Greece, 2007, pp. 516–527.

[6] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, SIAM U.S.A., 2003.

[7] University of Passau, *The Polyhedral Loop Parallelizer: LooPo*, 1997. http://www.infosun.fim.uni-passau.de/cl/loopo.

[8] Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen, *Reparallelization and Migration of OpenMP Programs*, International Symposium on Cluster Computing and the Grid, Rio de Janeiro, Brazil, 2007, pp. 529–540.

[9] James Clark and Steve DeRose, *XML Path Language*, W3C Recommendations, 1999–2007.

[10] Ian Foster, *Globus Toolkit Version 4: Software for Service-Oriented Systems*, International Conference on Network and Parallel Computing, 2006, pp. 2–13.

[11] Cătălin L. Dumitrescu, Dick H.J. Epema, Jan Dünnweber, and Sergei Gorlatch, *User-transparant Scheduling of Structured Parallel Applications in Grid Environments*, Workshop on Grid programming Environments and Components, Paris, France, 2006, pp. 85–92.

[12] Cătălin L. Dumitrescu, Jan Dünnweber, Philipp Lüdeking, Sergei Gorlatch, Ioan Raicu, and Ian Foster, *Simplifying Grid Application Programming Using Web-enabled Code Transfer Tools*, *in* Toward Next Generation Grids, *Springer 2007*, pp. 225–235.

[13] Jan Dünnweber, Sergei Gorlatch, Martin Griebl, Eduardo Argollo, and Christian Lengauer, *Making a Task Farm Component Parallelize Loops for the Grid*, CG Integration Workshop (CYFRONET), 2006, pp. 93–104.

[14] William Hoarau, Sébastien Tixeuil, Nuno Rodrigues, Décio Sousa, and Luis Silva, *Benchmarking the OGSA-DAI Middleware*, CG Integration Workshop (CYFRONET), 2006, pp. 357–368.

[15] Jan Dünnweber, Philipp Lüdeking, Cătălin L. Dumitrescu, Eduardo Argollo, and Sergei Gorlatch, *The HOC-SA Globus Incubator Project*, 2006. http://dev.globus.org/incubator/hoc-sa.

[16] CoreGRID Network of Excellence `www.coregrid.net`, *Basic Features of the Grid Component Model (GCM)*, Technical Report D.PM.04, Institute on Component-based Programming, 2005.

[17] Sergei Gorlatch and Jan Dünnweber, *From Grid Middleware to Grid Applications: Bridging the Gap with HOCs*, in *Future Generation Grids*, Springer Verlag, 2005, pp. 241–261.

[18] Martin Griebl, Peter Faber, and Christian Lengauer, *Space-time Mapping and Tiling – A Helpful Combination*, Concurrency and Computation: Practice and Experience **16** (March 2004), no. 3, 221–246.

[19] Jarek Gawor, Ian Foster, and Stephen Pickles et al., *State and Events for Web Services*, Intl' Conference on High-Performance and Distributed Computing, 2005, pp. 3–13.

[20] Christian Lengauer, *Loop Parallelization in the Polytope Model*, CONCUR, 1993, pp. 398–416.

[21] Object Management Group, *The Corba Component Model*, 1997. http://www.omg.org.

[22] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Weseley, 1998.

[23] The CCA Forum, *CCA Glossary*. http://www.cca-forum.org/glossary.

[24] European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies, *CoreGRID*. http://www.coregrid.net.

[25] UK Grid Database Task Force, *OGSA Data Access and Integration*. http://www.ogsadai.org.

Acknowledgement

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265) and has received financial support from the German Research Foundation (DFG) for project CompSpread.