The Grid-Occam Project

Peter Tröger, Martin von Löwis, and Andreas Polze

Hasso-Plattner-Institute at University of Potsdam,
Prof.-Dr.-Helmert-Str. 2-3,
14482 Potsdam, Germany
{peter.troeger, martin.vonloewis, andreas.polze}@hpi.uni-potsdam.de

Abstract. We present a new implementation of the old Occam language, using Microsoft .NET as the target platform. We show how Occam can be used to develop cluster and grid applications, and how such applications can be deployed. In particular, we discuss automatic placement of Occam processes onto processing nodes.

1   Introduction

The Occam programming language [17] was developed by INMOS, based on Hoare's idea of CSP [18]. Occam was originally designed for use on transputer systems [14], and the only Occam compiler that was in wide use would generate code only for the INMOS transputers (T222, T414, T800 etc.). Transputers are high-performance microprocessors that support parallel processing through on-chip hardware [28]. According to Michel J. Flynn's taxonomy [6] a transputer system can be declared as a multiple instruction multiple data stream (MIMD) system, more precisely as distributed memory system. Each transputer has four high-speed hardware links, allowing the creation of transputer networks. Communication between unconnected transputers would require routing of messages through intermediate transputers.

While Occam was primarily used on the transputers, its principles extend beyond special-purpose hardware, and Occam programs can be written without a specific interconnection network in mind. In principle, it is possible to execute Occam programs on a single processor, on a multiple-processor system using shared memory, or on a cluster of processor connected through some special-purpose or off-the-shelf interconnection network.

One of our primary aims in the project is the introduction of OCCAM to current distributed computing environments. In this context, we want to cover different distribution concepts and hardware types while minimizing the customization requirements for the programmer.

The paper is structured the following way: Section 2 gives an overview of the Occam language in general, while section 3 explains details of the historical INMOS Occam compiler. Section 4 introduces our Grid-Occam concept in detail. After that section 5 discusses concrete implementation details, and section 6 concludes the paper.

2 Occam Overview

The Occam language is based on the idea of communicating sequential processes. Each process has access to a set of (private) variables, and is defined as a sequence of primitive actions, namely

- assignment to variables (variable := value),
- output of a value through a channel (channel ! value),
- input from a channel (channel ? variable),
- SKIP, which does nothing, and
- STOP, which never terminates[1].

The communication through a channel uses the rendezvous style: an output operation will block until an input operation becomes available, and an input operation will block until the corresponding output becomes available. Then, the data is exchanged (i.e. the value of one process is stored into the variable of the other process), and communication succeeds. These primitive actions can be combined through complex processes using three combinators. This is an inductive definition of processes, where complex process can again be combined.

2.1 Sequential Combination of Processes

The keyword *SEQ* identifies a sequential process. The result of the entire process is obtained by executing one of the component processes after another; the entire sequential process terminates when the last subordinate process terminates. The sequence of component processes is indicated with an indentation of two spaces. As a very simple example, the following example shows a process that receives a value from one channel, performs a computation, and outputs a new value to another channel.

```
SEQ
  input ? variable
  variable := variable + 1
  output ! variable
```

2.2 Parallel Combination of Processes

The keyword *PAR* indicates parallel combination of processes. Like *SEQ*, the component processes are denoted by two spaces of indentation. The parallel statements simultaneously starts execution of all component processes. The entire statement is complete when the last component completes. A very simple example shows the interaction of the sequential process from the previous example in parallel with a process that interacts with the user:

---

[1] This is the formal definition of STOP. STOP indicates that a process is in error, and the typical implementation is that the process stops working, i.e. it ceases to act. Formally, this is the same thing as continuing to do nothing eternally.

```
PAR
  SEQ
    keyboard ? var1
    to.computer ! var1
    from.computer ? var1
    display ! var1
  SEQ
     to.computer ? var2
     var2 := var2 + 1
     from.computer ! var2
```

The first of the two parallel processes is a sequential one. It interacts with the user, through which it is connected with the channels keyboard and display. This process reads some input from the user, and sends it to the other process. That process performs the computation, and sends the result back to the first process, which then forwards the result to the display.

2.3 Alternative Combination of Processes

If a process has communication channels to many other processes, it needs to select which channel to receive data from. This is done with the *ALT* combinator, which consists of a set of guard inputs and associated processes. The channels are all watched simultaneously. Once a channel is ready, the input is performed, and the associated process is executed. The following process simultaneously guards a channel from the user and from another process, executing, forwarding the data to the other communication partner:

```
ALT
  keyboard ? var1
    to.computer ! var1
  from.computer ? var1
    display ! var1
```

Since the *ALT* statement eventually executes only a single process, it is typically nested within a loop which repeatedly executes the *ALT* statement.

2.4 Repeated Combinators

Each combinatory can be defined in a repetitive way. This is often combined with arrays, so that multiple processes operate on array data in parallel. In addition, the parallel processes have their individual channels, which is possible through the notion of channel arrays. The declarations

```
VALUE num.of.fields IS 1024:
[num.of.fields]REAL64 a,b:
[num.of.fields]CHAN OF REAL64 channel:
```

introduce arrays a and b of *REAL64* values, and a single array of channels communicating *REAL64* values. These structures can be used in the following way:

```
INT todo:
REAL64 total, value:
PAR
  SEQ
    todo := num.of.fields
    total  := 0.0
    WHILE todo > 0
      ALT I := 1 FOR num.of.fields
        channel[I] ? value
        total := total + value
        todo := todo - 1
    display ! total
  PAR k := 1 FOR num.of.fields
    channel[k] ! a[k] + b[k]
```

In this process, 1025 parallel processes are created. 1024 of them add two numbers and send the result to a channel. The remaining process receives one result after another and sums them up, eventually sending the total to the display.

2.5   Further Concepts

Occam provides a lot of concepts not presented in this language overview. There are conditional and case statements, loops (used in the previous example), constructed data types, protocols, timers, and many more concepts.

3   INMOS Occam

In the last section we showed that the Occam programming model includes a concept of parallel processes that communicate through unidirectional channels. One of Occam's major advantages with this concept is the abstraction of the parallel execution from the underlying hardware and software environment.

The original INMOS compiler supports the code generation for networks of heterogeneous transputer processors. Therefore the Occam process must be specifically placed on the correct hardware processor. Another problem is the interconnection of processors. A transputer processor can, but need not be connected to its 4 neighbors by the hardware links, which themselves can carry multiple Occam channels. Beside the 4-link star layout, it is possible to have other interconnection schemes for hardware channels, for example a tree or a pipeline scheme. In such cases, the compiler must be able to map the given set of named Occam processes on the right processors for a successful execution of

the program. This leads to the need for custom message forwarding, performed by special *routing kernel* code.

The classical Occam solved the problem of process placement by special configuration statements *PLACED PAR* and *PROCESSOR*. Later on, INMOS specified a separate *configuration language*. It supports the design of portable, hardware-independent parallel Occam code. The first part of the configuration (*NETWORK*) describes the interconnection of all available transputer processors. This includes also the specific type of hardware in terms of processor type and memory size. Each defined processor is labeled with a custom name. The second configuration part (*CONFIG*) describes the placement of named Occam processes on virtual processors, identified by the name given in the first part. The usage of a configuration language puts the developer in charge of the process placement. This is uncritical in the context of transputer systems, since the hardware architecture is static and well-known. The execution of such Occam code on another transputer system requires an adaption of the network configuration and a recompilation.

An example for the provisioning of infrastructure-independent Occam environments originates from the Esprit P2701 PUMA project at University of Southampton. They developed a virtual channel router (VCR) software for unrestricted channel communication across a network of transputers [4]. The software is able to map static virtual channels at runtime on communication resources, using a custom kernel on the particular transputer node. The topology of the network is described by an automatically generated network configuration file, after that the VCR generates a set of deadlock-free routes for messages in the system. The resulting message routing is realized during runtime, based on the precomputed informations.

The following section shows that we extend VCR idea of infrastructure-independent Occam execution to cluster and grid environments.

4   The Grid-Occam Approach

The last section showed that the original Occam compiler supports heterogeneity aspects through usage of a static configuration description. It is easy to see that such an approach is not appropriate in heterogeneous cluster or grid environments. Additionally the programmer wants to concentrate on the aspects of parallelization for his algorithm, not on specific mapping issues for a particular execution environment.

A good example of infrastructure-independent programming is the popular MPI [7] library. It enables the programmer to use message passing facilities for parallel program instances, while avoiding dependencies on the concrete hardware infrastructure, e.g. number of processors, interconnections, or the communication technology. Another advantage is an easier development process. A programmer can test its software on a local computer without claiming expensive resources like cluster time, simply by using a one-node enabled, local version of the MPI library.

In the current approach of our Grid-Occam project, we want to achieve a similar infrastructure independence within the concept of a .NET [20] Occam runtime system. As a first step, we divide process-independent parts of the implementation from the application-specific code. The process-independent functionality is similar across all Occam programs and covers mainly the distribution aspects. We call the result a *Occam runtime library* that is being responsible for all distribution-related issues within the execution of the program. In an ideal situation, a programmer can adopt her already existing Occam program by simply exchanging this runtime library without any need for reconfiguration or recompilation.

In an implementation every Occam runtime has to solve several critical issues:

– What is the given infrastructure in terms of virtual processors and their interconnection?
– What is the best possible placement strategy for Occam processes?
– What is the precise instantiation mechanism for a *PAR* block on a virtual processor?
– How could global Occam variables be implemented?
– What is a channel instance on this particular infrastructure?
– How is such a channel being allocated and how does the addressing of an other specific process instance work?
– What are the implementation strategies for rendezvous behavior of a channel?
– Which kind of networking optimization, for example with channel arrays, can be done for the given infrastructure?

As it can be seen, in classical Occam most of these questions were solved by manual work in the configuration file. In contrast, we have to work on best-effort placement algorithms for Occam processes on the one side and on automated detection of infrastructure information on the other side.

4.1 Grid Computing

Grid Computing is defined as the coordinated, transparent and secure usage of shared IT resources, crossing geographical and organizational boundaries [9]. It splits up into the research areas of computational grids, data grids and resource grids [11]. The development of research and commercial usage of grid technologies has increased heavily in the last years. The community established the Global Grid Forum (GGF) as standardization group. Major industrial companies like IBM, Sun or Oracle invest in the development of standardized interfaces to their cluster software. Research groups from all kinds of nature sciences are using collaborative grid resources for the computing-intensive applications.

The actual development shows that there is a large range of possible grid technology users. Most of them are not professional computer scientists, therefore the design and usability of grid interfaces becomes an important factor. Actually most users have the choice between basic command-line or library functions

and high-level job submission web interfaces, so-called Grid portals. There are several approaches to make the overall usage more easier, even in the GGF standardization process [13]. In this context we see Occam as a useful solution in the context of scientific computing-intensive grid applications. Occam supports parallelism as first-level language construct, which allows a very natural kind of programming for distributed computation. Due to the nature of .NET it is possible to combine old legacy source code (e.g. written in Fortran) with a new Occam program for the execution.

4.2 Process Placement

In our concept we define a set of possible classes for runtime libraries. They mainly differ in their unit of distribution. We assume in the first step that possible granularity levels could be *threads*, *processes*, *cluster nodes* and *grid nodes*. The granularity can be defined by the ratio of communication and execution costs. In every level of granularity, we have a different meaning for the concept of a virtual processor - a single thread, a single process, a single cluster node or a single grid node. The respective interconnection mechanisms for the different distribution could be shared memory, communication pipes, cluster networking techniques like TCP and Myrinet, and grid networking techniques, for example TCP over multiple hops or ATM.

The LogP model For a classification of the described granularity levels we make use of Culler's LogP model [3]. It defines a model of a distributed-memory multiprocessor system using 4 parameters:

- Latency: upper bound for the network data transfer time. This parameter increases linearly for the levels of threads, processes and cluster nodes. Latency for grid environments is substantially higher in most cases.
- Overhead: exclusive time needed by a processor for sending or receiving a data packet. This parameter is usually optimized through hardware and should be small enough in all cases to be ignored.
- Gap: minimal interval between two send or received messages. The gap is fixed for a given type of networking infrastructure and mainly influenced by the node communication hardware. The usage of shared memory and local communication pipes for threads and processes leads to a very small gap. The gap for cluster architectures is typically smaller than for grid environments, reasoned by the usage of specialized networking hardware. Grid infrastructures tend to rely on standard TCP/IP communication mechanisms.
- Processors: number of processors in the system. In the case of threads and processes, the number of virtual processors is limited by the number of physical processors in a SMP system. A cluster system could consists of a large number of machines, up to multiple hundreds. We assume that in most practical cases the number of cluster nodes in a cluster is roughly equal or even higher than the number of grid nodes potentially available to a programmer.

As a concretion of the model we consider the fact that most usual SMP systems work on the scheduling granularity of threads. Since both thread and the process granularity levels are designed for single machines, we leave out the level of processes in our further observations.

Task Graphs After classifying the possible execution environments we have to think about a mapping of communicating Occam processes to a resource network of a chosen granularity level. We want to represent the Occam program as a data-flow or task graph, which is a common approach in parallel computing programming [25]. Nodes in the graph represent a set of sequential processes, edges the data dependencies between them during parallel execution. A data dependency naturally leads to a communication channel between parallel processes in Occam.

We plan to acquire a best possible structural information during the compilation process. It could be possible that the design of a Occam program prevents a complete static analysis, for example in presence of variable-sized channel arrays. However, for the first implementations we simply ignore such information in our architectural graph.

Mapping a task-graph to a given network of processors is a well-known problem in research. There are several approaches to solve this NP-complete problem heuristically. However, the usual communication model in these scheduling algorithms concentrates only on the latency aspect of the processor connection. Other research uses heuristic approaches for mapping task graphs to LogP-described networks of processors. The MSA LogP scheduling algorithm [1] even considers the bundling of messages.

We can conclude that the existing approaches give us the relevant mechanisms to map a task-graph to a graph representation of the infrastructure. Most algorithms rely on 4 parameters: The set of tasks, their interconnection scheme, the execution cost for a particular task (node weight) and on the amount of data transmitted over an interconnection (edge weight).

The overall set of tasks and their interconnection scheme can be determined by the compiler. A single task (graph node) can be seen as a set of sequential atomic Occam instructions after reading a channel value up to the end of the surrounding SEQ block. The node weight could be simply determined by counting the number of atomic operations following the read operation. In fact several issues (like repeated instructions) must be considered in this case, although it should be possible to get a useful estimation. Another major factor is the edge weight, representing the number of fixed-sized messages transmitted through a particular channel. The compiler can analyze the write operation(s) to the typed channel and the amount of data transmitted through this particular instance. Again it could be possible that we have to consider non-static conditions, like conditional or repeated code blocks. We will investigate whether the compiler can simply leave out such information or if there is need for a restriction of the language [21] to achieve valuable results.

Another important characteristic of the Grid-Occam approach is the nested nature of the different granularity levels. An Occam process for a particular type of virtual processor, for example a cluster node, can itself be executed on a network of higher granularity virtual processors, for example with multiple threads. We combine this model of nested granularities with the representation of Occam programs as unidirectional graph, simply by repartitioning a subset of tasks for a higher granularity level. Due to the nature of the different execution levels, the ratio of computation time to communication effort should become higher with decreasing granularity. It must also be considered that we have a fixed number of virtual processors for the thread level. There is no possibility, like in the grid case, to request more processors for a successful execution. This leads to a need for algorithms that are able to perform a partitioning for a fixed set of processors. This could also be relevant if the number of available cluster processors is not large enough for a particular task sub-graph. Beside classical graph partitioning algorithms [5] there are approaches for cluster computing environments that can consider this restriction.

5   Implementation Strategy

The following section will concentrate on specific techniques for the implementation of our Grid-Occam vision. We will explain our choice for the .NET framework and give details about the implementation strategies for the different runtime libraries.

5.1   The .NET framework

The implementation of our Occam compiler will generate intermediate language (IL) byte code, executable within the .NET framework. Microsoft has published the .NET framework [20] in 2000. An explicit design goal was the support for multiple languages. Major parts of the runtime specification were submitted to standardization organizations like ECMA and ISO. This allows other software vendors to re-implement .NET for other systems. The Microsoft .NET framework product (in its current version 1.1) is only available on Win32/X86 and Windows CE systems. Additionally Microsoft Research has published the Rotor package for non-commercial purposes, which ports .NET to MacOS X and FreeBSD/X86 [30]. The GNU Mono project [34] allows execution of .NET applications on Linux systems, using various processors: x86, PowerPC, Sparc, IBM S/390, and Intel StrongARM. The upcoming next version of .NET (Whidbey) will have support for 64bit Itanium/Opteron systems.

In the context of Grid-Occam, we see .NET as a useful runtime environment that offers execution on differing platforms and the integration of multiple language libraries. On example could be the integration of legacy FORTRAN code by using the commercial Lahey compiler for .NET [22]. .NET offers, in contrast to other virtual machine implementations, an integrated code-access security concept, which becomes relevant in the context of remote execution on

cluster grid nodes. Performance investigations for the .NET runtime [32] showed that .NET can give an sufficient result for high-performance computing. The .NET framework offers an integrated support of current web service technologies, which eases the use of OGSI-based grid infrastructures [19]. There also support classes for the dynamic creation and compilation of new IL code, which becomes important in the context of code generation for a cluster node.

5.2 Program Execution

The figure 1 gives a general overview of the execution environment in the grid context.
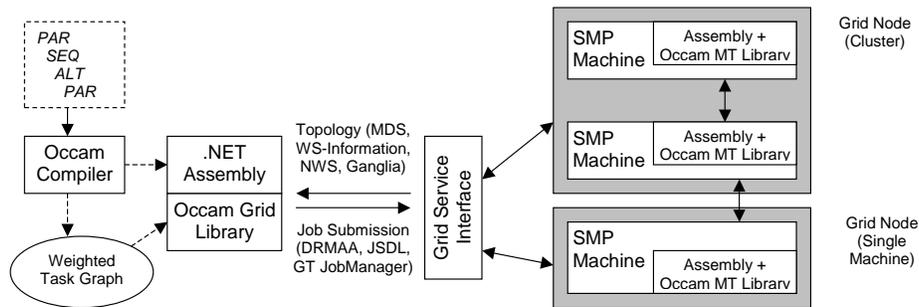
Fig. 1. Overview of the Grid-Occam architecture

In a first step, the Occam compiler generates executable IL code, packaged in an *assembly file.* All instructions relevant to parallel execution, like channel operations or instantiation of PAR blocks, are implemented as calls to the Occam runtime library. The compiler will generate C# code in the first version, which can be easily compiled to IL in a post-build step. The compiler also produces a task graph for the different PAR blocks and their regarding channels, augmented with an estimation of the node weights (computational load) and edge weights (channel load). This data is used by the respective runtime library to choose a job placement, based on the detected topology of virtual processors. After that the library initiates all planned job submission actions to execute the program. We assume that dynamic changes in the execution environment (e.g. unavailable nodes) are handled by the cluster or grid mechanisms.

5.3 Runtime Libraries

As defined earlier we see 3 possible kinds of runtime libraries, namely for threads, cluster nodes and grid nodes. In figure 1 the resulting .NET executable uses the Occam grid runtime library. It would also be possible to exchange it with the multi-threaded or the cluster version of the library, which enables the user to test

his development on the local computer before submission to a larger resource network.

Multi-threading Runtime Library The multi-threaded runtime library allows the execution on a single- or multi-processor machine. The number of available physical processors on the machine must be considered in the runtime library. For .NET this information is available through the *System.Diagnostics* namespace.

The realization could be improved with the upcoming new version of the .NET framework (Whidbey). It will be possible to use the popular OpenMP library [2] in .NET, which allows an even more direct mapping of the Occam semantic to .NET code.

The channel communication for parallel processes will be realized by an interlocked shared memory segment. The rendezvous behavior can be achieved by the coordinated usage of multiple semaphore variables, which are available in the .NET environment.

Cluster Runtime Library The cluster runtime library is intented for usage in a pure cluster environment, without using grid interfaces for the job submission. The cluster runtime needs some kind of topology information to perform a best-effort placement of jobs. The minimal requirements are an identifier for the own node and the list of nodes involved, if we assume that the cluster forms a complete graph. The communication mechanisms, for example an available MPI installation, should be able to address a particular node.

We plan to implement a first prototype based on the Condor [23] system, which offers all the relevant information and coordination services with Condor DAGMan [31]. Therefore it is possible to coordinate the instantiation of parallel jobs on the Condor cluster. Every instance in the cluster has to get informed about its own identification number, usually with a command-line argument. This can be specified in the Condor submit file, which has a notion of the *process id*. The list of involved nodes can be distributed on a shared filesystem or through the automated file transfer mechanisms. Since all Condor machines get the same executable it must be ensured that a node instance only execute a particular part of the Occam code. The decision for a code part is based on the own identity in the task graph of the Occam program. The assembly itself could also use the multi-threaded runtime library to perform a more fine-granularly execution. This could be the case if one Condor node executes a SEQ block with multiple PAR blocks. Condor offers information services to determine the hardware characteristics of a node. This enables the cluster runtime to choose whether it is appropriate to perform a more granular parallel execution on one node.

Grid Resource Broker Our Grid-Occam architecture has two possible scenarios for the usage of grid resources.

In the first case the grid acts only as source for computing resources. We use the resource information and job submission services of a grid to utilize a particular cluster resource for execution. Popular examples for information services are the Globus Meta Directory Service (MDS) [8] or the upcoming WS-Information services in WSRF [10] architectures.

The identification of an appropriate resource is followed by a job submission to the chosen grid resource. Globus offers the job manager functionalities for this, but there are also concurrent implementations from other projects [24]. We plan to consider ongoing standardization efforts from the GGF for this problem. There are promising developments in the DRMAA [27] and the JSDL [29] working group for a standardized job submission interface. With the availability of actual implementations we will support such interfaces in addition to existing Globus mechanisms.

Another interesting issue is the availability of input and output files for a job. In the cluster case we can expect a common file system like NFS, or even a transport mechanisms from the cluster software. Similar to the INMOS *hostio* library it should be possible to read and write files in jobs send to a grid node. Most grid environments allow a specification of files to be transfered before and after job execution, in this case standardized mechanisms like GridFTP [8] are used.

Grid Execution Environment The ongoing development in Grid Computing focuses more and more on an improved support for widely distributed grid applications, called "managed shared virtual system" [26]. In this context we see Grid-Occam as a useful approach for the development of widely-distributed applications. From the investigation in the section 4.2 we know that inter-node communication for grids can have unpredictable and slow behaviour. Therefore it must be ensured that the placement algorithm considers this fact carefully. Monitoring services like NWS [33] offering their information through grid interfaces, which allows the collection of all relevant data for the task of process placement.

The implementation of channels is another interesting issue for grid execution. Globus 2 introduced the MPICH-G2 library [8] for cross-node communication facilities, the latest development is Globus XIO [12]. In a service oriented grid it could be possible to implement a channel service, both reader and writer perform there operation against an instance of this service. The service implementation is responsible for rendevouz behavior of the channel, for example through notification [15] of the client. More sophisticated approaches [16] use the tuple-space approach for the implementation of a rendevouz communication model. This could also be realized with an OGSI conformant service as front-end.

6   Conclusion

In this paper we presented our vision of an Occam implementation, capable of being used in modern cluster and grid environments. The availability of distribu-

tion as first-level language concept can ease the usage of complicated cluster and grid infrastructures, which is one of the active topics in grid computing research. Occam has a long history in distributed programming and is widely accepted in its suitability for such environments.

We plan to automate the process placement, as far as possible, through infrastructure information services available in modern cluster and grid environments. We will rely on latest developments in this area, including the standardization efforts in the GGF.

We are actually developing an Occam compiler for .NET / Rotor in the context of a lecture. The first version will include prototypes for all presented types of runtime environment and will act as foundation for our further research.

7 Acknowledgements

References

1. Cristina Boeres and Vinod E. F. Rebello. A versatile cost modeling approach for multicomputer task scheduling. *Parallel Computing*, 25(1):63–86, 1999.
2. Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.
3. David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
4. M. Debbage, M.B. Hill, and D.A. Nicole. Towards a distributed implementation of occam. In *Proceedings of the 13th Occam Users Group*. IOS Press, 1990.
5. Per-Olof Fjllstrm. Algorithms for graph partitioning: A survey. *Linkping Electronic Articles in Computer and Information Science*, 3(10), 1998.
6. M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.
7. MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, Tennessee, July 1997.
8. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
9. Ian Foster. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.
10. Ian Foster, Jeffrey Frey, Steve Graham, Steve Tuecke, Karl Czajkowski, Don Ferguson, Frank Leymann, Martin Nally, Igor Sedukhin, David Snelling, Tony Storey, William Vambenepe, and Sanjiva Weerawarana. Modeling stateful resources with web services. *IBM DeveloperWorks Whitepaper*, March 2004.
11. Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.
12. The Globus Alliance. *Globus XIO*, 2004.

13. Tom Goodale, Keith Jackson, and Stephen Pickles. Simple API for Grid Applications (SAGA) Working Group. http://forge.ggf.org/projects/gapi-wg/.

14. Ian Graham and Tim King. *The Transputer Handbook*. Prentice Hall, January 1991.

15. Steve Graham, Peter Niblett, Dave Chappell, Amy Lewis, Nataraj Nagaratnam, Jay Parikh, Sanjay Patil, Shivajee Samdarshi, Igor Sedukhin, David Snelling, Steve Tuecke, William Vambenepe, and Bill Weihl. Publish-subscribe notification for web services. *IBM DeveloperWorks Whitepaper*, March 2004.

16. K.A. Hawick, H.A. James, and L.H. Pritchard. Tuple-space based middleware for distributed computing. Technical Report 128, Distributed and High-Performance Computing Group, University of Adelaide, Adelaide, Australia, October 2002.

17. C.A.R. Hoare. *Occam 2 Reference Manual: Inmos Limited*. Prentice-Hall, 1988.

18. C.A.R. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare, 2004.

19. Marty Humphrey. From Legion to Legion-G to OGSI.NET: Object-Based Computing for Grids. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*. IEEE Computer Society, April 2003.

20. Jeffrey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

21. U. Kastens, F. Meyer auf der Heide, A. Wachsmann, and F. Wichmann. Occam-light: A language combining shared memory and message passing (a first report). In *Proc. 3rd PASA Workshop, PARS Mitteilungen*, pages 50–55, 1993.

22. Lahey Computer Systems Inc. LF Fortran Manual. http://www.lahey.com/.

23. M.J. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 104–111, 1988.

24. Stephen McGough. A Common Job Description Markup Language written in XML. http://www.lesc.doc.ic.ac.uk/projects/jdml.pdf.

25. H.E. Motteler. Occam and dataflow. Technical report, UMBC Technical Report, September 1989.

26. Jarek Nabrzyski, Jennifer M. Schopf, and Jab Weglarz. *Grid Resource Management*. Kluwer Academic Publishers, 2004.

27. Hrabri Rajic, Roger Brobst, Waiman Chan, Fritz Ferstl, Jeff Gardiner, Andreas Haas, Bill Nitzberg, and John Tollefsrud. Distributed Resource Management Application API Specification 1.0. http://forge.ggf.org/projects/drmaa-wg/, 2004.

28. Ram Meenakshisundaram. Transputer Information Home Page. http://www.classiccmp.org/transputer/.

29. Andreas Savva, Ali Anjomshoaa, Fred Brisard, R Lee Cook, Donal K. Fellows, An Ly, Stephen McGough, and Darren Pulsipher. Job Submission Description Language (JSDL) Specification Version 0.2. http://forge.ggf.org/projects/jsdl-wg/, 2004.

30. David Stutz, Ted Neward, Geoff Shilling, Ted Neward, David Stutz, and Geoff Shilling. *Shared Source CLI Essentials*. O'Reilly, December 2002.

31. Condor Team. *Condor Manual*. University of Wisconsin-Madison, 2004.

32. Werner Vogels. HPC.NET - are CLI-based Virtual Machines Suitable for High Performance Computing? In *SC'03*, Phoenix, Arizona, USA, November 2003.

33. Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

34. Ximian Inc. Mono Framework. http://www.go-mono.org, 2004.