Programming for Malleability with Hybrid MPI-2 and OpenMP – Experiences with a Simulation Program for Global Water Prognosis

Claudia Leopold, Michael Süß, Jens Breitbart
University of Kassel, Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, 34121 Kassel, Germany
{leopold,msuess}@uni-kassel.de, brietbar@student.uni-kassel.de

Keywords—**Parallelization of Simulation, Libraries and Programming Environments, Message Passing**

Abstract— **This paper reports on our experiences in parallelizing WaterGAP, an originally sequential C++ program for global assessment and prognosis of water availability. The parallel program runs on a heterogeneous SMP cluster and combines different parallel programming paradigms: First, at its outer level, it uses master/slave communication implemented with MPI. Second, within the slave processes, multiple threads are spawned by OpenMP directives to exploit data parallelism. Time measurements show that the hybrid scheme pays off. It adapts to the heterogeneity of the cluster by using multiple threads only for the largest tasks and mapping these to multiprocessor nodes. Third, the program is malleable, which has been accomplished with the dynamic process management facilities of MPI-2, based on the MPICH2 implementation. In particular, it is possible to increase the number of processes while the program is running. Malleability is an important feature in both batch systems and grid environments. We discuss the support that MPI-2 provides for malleability.**

I. Introduction

Parallel programming today is dominated by the Message Passing Interface MPI [3], [9], OpenMP [10], and lower-level threading libraries [7], [14]. While MPI is most appropriate for distributed-memory architectures, OpenMP has been designed for shared-memory machines. A common architecture nowadays are SMP clusters, i.e., clusters of multiprocessor nodes, which combine shared memory in each node with distributed memory in between the nodes. Accordingly, hybrid forms of programming, using MPI at the outer level of the program and OpenMP for decomposing individual processes into threads have been proposed [11], [12].

Other properties of current architectures, especially heterogeneity and dynamic behavior, have received far less consideration as of yet in the parallel programming community. We speak of heterogeneity if an architecture is composed of different computing resources, for instance if the nodes of an SMP cluster differ in their number of processors. We speak of dynamic behavior if the number of processors available to an application varies during the program's execution. The term *malleability* has been coined to denote the ability of programs to adapt to changes in the number of processors [15]. Malleability is helpful in batch systems, where it gives the scheduler more freedom in assigning jobs, but it is particularly important in grids.

This paper evaluates the opportunities that MPI and OpenMP provide for making use of modern architectures, based on an example program from the simulation domain. The program is called WaterGAP, which stands for "Water – Global Assessment and Prognosis". It has been developed at the Center for Environmental Systems Research of the University of Kassel, and is explained in Section II of this paper. Briefly stated, WaterGAP partitions the surface area of continents into equal-sized grid cells. Based on input data for climate, vegetation etc., it simulates the flow of water, both vertically (precipitation, transpiration) and horizontally (routing through river networks). Presently, the input data are being refined, and to cope with the resulting increase in computational expense, the program was parallelized.

From a computational point of view, the vertical simulation in WaterGAP gives rise to data parallelism, as grid cells are computed independently. The horizontal simulation requires communication, but one can observe that the world is partitioned into independent basins. Independent means in this context that there is no flow of water between basins, a property that is inherent in the hydrological model behind WaterGAP. Thus, different basins can be computed by different processes, without any need for communication. The basins differ in size, from a few very large basins to many small ones. The size of the basins is determined by natural conditions, and thus cannot be changed.

As target architecture for our parallelization efforts, we consider the compute cluster of the University of Kassel. This Linux cluster consists of 58 double-processor nodes, and one 8-processor node that comprises 4 dual-core chips. Jobs are submitted through the Torque batch system, which is an OpenPBS derivative. The cluster is usually operating at full capacity, with the majority of jobs being sequential and long-running. Before a WaterGAP run can be started, it may take for several hours until the requested number of nodes has become free. During this time, the batch system assigns processors to WaterGAP as they become available, but it will not start the program until all the requested nodes are available. Thus, the other processors are idle for several hours, which is a waste of resources. On the cluster, both MPI-2, with the MPICH2 implementation, and OpenMP, with the Portland compiler, are available. The parallelization has been accomplished in three steps:

1. Master/slave structure: This standard pattern has been implemented with MPI. It is used to distribute the basins to be computed among processes. The resulting program scales almost linearly up to about 8 slaves. Beyond that, the computation of the largest basin prevents any further speedups.

Proceedings 20th European Conference on Modelling and Simulation
Wolfgang Borutzky, Alessandra Orsoni, Richard Zobel © ECMS, 2006
ISBN 0-9553018-0-7 / ISBN 0-9553018-1-5 (CD)

2. Hybrid parallelism: Keeping the master/slave structure unchanged, we used OpenMP to split individual slaves into multiple threads. These threads exploit the data parallelism among grid cells within a basin, during the vertical and part of the horizontal simulation. The speedup from threading is sublinear, but the hybrid scheme can be used to overcome the bottleneck at the largest basin. In our heterogeneous cluster, the scheme is particularly efficient if the largest basin is mapped to the 8-processor node, and all but the two largest basins are computed by a single thread only.
3. Malleability: To avoid idling of processors before the start of WaterGAP through the batch system, we rewrote the application so that it can be started with a small number of slaves, and later incorporates more and more slaves as the processors become available. We implemented this feature using the dynamic process management facilities of MPI-2.

While master/slave parallelism and hybrid parallelism are already quite well understood, we do not know of any previous work on the implementation of malleability with MPI-2. This paper describes our experiences, and discusses design decisions in our program.

The paper is organized as follows. First, Sect. II describes the WaterGAP application. Then, the main part consists of sections III–V, which are devoted to the three steps of parallelization as outlined above. Sect. VI reviews related work, while finally, Sect. VII summarizes the paper and mentions directions for future research.

II. WaterGAP

The WaterGAP program [1], [6], [16] has been developed at the University of Kassel, with the goal to investigate current and future water availability worldwide. Several projects have already been carried out with this program, including studies on the impacts of global changes in climate as well as water withdrawals from households, factories, irrigated farms, and so on.

The WaterGAP program operates on several hundred input files, and generates about 15 output files per year of the simulation period. A typical simulation period is 30 years. Among others, input data refer to climate, vegetation, and water use. Most input data are available for each grid cell; part of the data per day, others per month or per year. Among the input data is a flow direction map that assigns a single neighbouring cell to each grid cell, where the surplus water from this cell flows into. Output data include results on various specific measures such as groundwater runoff and snow cover, and are collected per grid cell. Most outputs are generated annually, some monthly.

The WaterGAP program has been written in C++ and comprises about 15.000 lines of code. The program can be run in different modes (e.g. calibration mode), of which this paper only considers global computation mode, i.e., the simulation is carried out for each grid cell worldwide.

After an initialization phase, the main loop of WaterGAP iterates through the days of the simulation period. For each day, it carries out a vertical simulation first, and several steps of horizontal simulation thereafter. During the vertical simulation, transpiration and water balance are computed for each grid cell, based on data for precipitation, temperature, etc. The horizontal simulation calculates the flow of water through the networks of rivers, lakes etc. Whereas the vertical simulation runs once per day, the horizontal simulation is carried out several times per day, with the exact number of time steps depending on river velocity.

During each vertical simulation step, the computation of different grid cells is independent. During horizontal simulation, water is exchanged between grid cells, but this exchange is restricted to occur between cells of the same basin. For hydrological reasons, the world is divided into a total of about 10.000 basins. A few of these are very large, such as the Amazon area, but the majority of basins comprises only a few grid cells. The basins are stable, i.e., the set of basins does not depend on any inputs other than the flow direction map, and even the relative computational expense of the basins is only lightly dependent on input.

Currently, the input data of WaterGAP are being refined to increase the spatial resolution from 0.5° to 5 minutes side lengths of grid cells. With the 0.5° version, WaterGAP had a running time of about 10 hours on a standard PC, for a simulation period of 30 years. With the new resolution, the running time increases by a factor of about 36, which can only be handled on a parallel machine in a reasonable amount of time.

III. Master/Slave Parallelism and I/O

The first step of our parallelization uses the fact that basins are computationally independent. It takes the simulation of each basin, over the whole simulation period, as a task. Since task sizes differ, we use the master/slave scheme for mapping tasks to processes. This well-known scheme deploys one master process and several slaves. The master starts by sending a task to each slave. Whenever a slave has finished its work, it reports the result to the master and gets the next task, until all tasks have been processed.

In a preprocessing step, we group grid cells into basins, referring to the information in the flow direction map. The result contains many small basins, for which the overhead of master/slave communication and I/O would outweigh any gains in performance. Therefore, we group the small basins into larger working units to be distributed to the slaves, and store the result in a file. For ease of presentation, we denote the working units as basins, as well. The preprocessing does not need to be repeated in each program run, but only when the flow direction map changes instead.

Table I depicts the distribution of sizes among basins. The numbers have been obtained by grouping natural basins of size up to 100 grid cells together in a working unit. Threshold 100 has been selected experimentally to maximizes the overall speedup, and is used throughout this paper. As can be seen in the table, there are a few very large basins, with the largest basin taking more than twice the compute time of the second largest. The master assigns the basins in decreasing order of size, i.e., the largest basin is assigned first.

Another aspect of master/slave parallelization is I/O. Both the input files and the output files follow a standard format that allows their contents to be visualized with a tool. The format requires a particular arrangement of grid cells, which is not compatible with the assignment to basins. Thus, in each file, the grid cells of any single basin are scattered throughout the file. Therefore each slave must read and

Number of basins	Size (in grid cells)	Time (in sec.)
1	3165	275
1	1946	120
5	1000 – 1700	50 – 90
400	100 – 1000	2 – 40

TABLE I: Size distribution of basins (units). Running time has been measured for a simulation period of two years on a single processor.

write discontinuous data that do not follow a regular pattern.

MPI-2 supports the efficient access of multiple processes to the same file. The focus, however, is on regular patterns that can be described by a MPI datatype. Although an explicit positioning of file pointers is possible, the use of MPI-2 I/O slowed down our program considerably. Therefore, we implemented a conservative approach using language I/O (`fread`, `fwrite` etc.). Input files are read by all processes, while output data are collected and written by the master.

The master/slave scheme scales almost linearly up to about 8 slaves (see Sect. IV). With more slaves, the largest basin becomes a bottleneck and prevents any further speedup.

IV. Hybrid Parallelism and Heterogeneity

To scale the number of slaves beyond 8, we have to internally parallelize the computation of basins. As already stated, the program has potential for data parallelism, since grid cells are independent during vertical simulation. While this potential can, in principle, be exploited with MPI, the distributed-memory programming model of MPI would force us to partition data structures among processes, and to realize all communication between grid cells through explicit message passing.

Therefore, we chose OpenMP for intra-basin parallelization. OpenMP has a shared-memory model, and so there was no need to change the existing data structures. Since the two parallel programming systems are used for orthogonal aspects of parallelism (inter-basin vs. intra-basin), their combination only slightly increases the complexity of the program. In fact, OpenMP parallelization was easy. We identified and parallelized some central loops: the main loop of vertical simulation, and several loops over grid cells during horizontal simulation.

Figure 1 shows timing results of the hybrid MPI / OpenMP program with different numbers of slaves and threads. In addition to the slaves, one single-threaded master process was used. Running times are in seconds, and refer to a simulation period of two years. The programs have been compiled with `mpich1.2.7`, using the Portland compiler with optimization level `-fast -Mconcur`. Measurements were carried out on three architectures: the compute cluster of the University of Kassel (double-processor AMD Opteron 248 nodes), the compute cluster of the University of Frankfurt (double-processor AMD Opteron 244 nodes), and a more powerful node of the Kassel cluster (4 AMD Opteron 875 dual-core processors). Jobs have been submitted through the batch system so that each process and thread had exclusive access to a processor (except for the

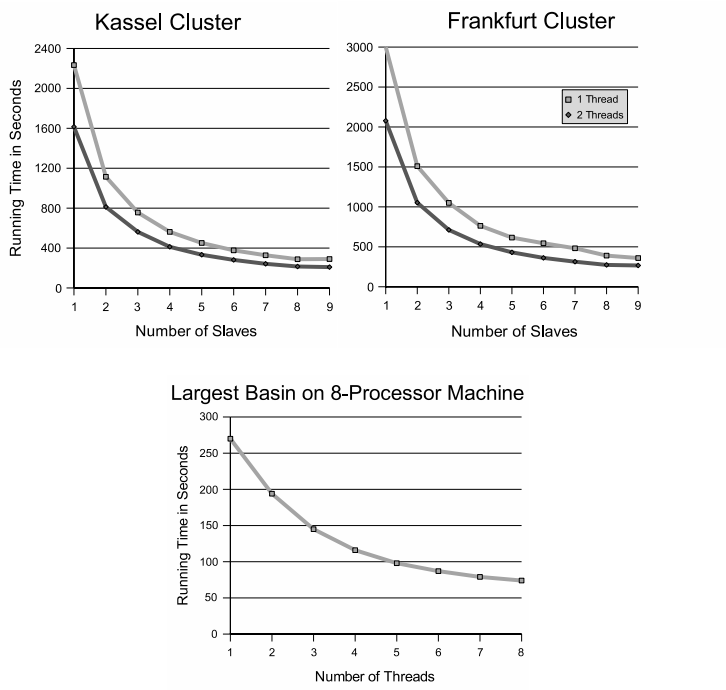

Fig. 1. Running time of hybrid program

largest run on the 8-processor node).

The program scales almost linearly up to 8 or 9 slaves, both in the pure MPI version, and in the hybrid version with two threads. Beyond that, the running time remains constant, and the computation of the largest basin forms a bottleneck (not shown in the figure, but was observed in the experiments). The numbers show that OpenMP alone contributes to a significant, although sublinear, performance gain, and that this gain is orthogonal to the gain from MPI parallelization.

Since inter-basin parallelism yields higher speedups than intra-basin parallelism, the hybrid scheme pays off most if it is coupled with heterogeneity, i.e., if different processes use a different number of threads. Heterogeneity must be supported by the batch system. The Torque system accepts requests such as: 4 nodes with 2 processors plus 1 node with 8 processors plus 1 processor from any node. To start a heterogeneous run, some programming is required in the batch script, to control the assignment of MPI processes to nodes. The WaterGAP program profits from heterogeneity in three ways:

- First, in our realization of the master/slave scheme, the master does not participate in the computation (we chose this variant for simplicity). Instead, it concentrates on communication with slaves and I/O, which is not compute-intensive. Therefore, it is sufficient to use a single thread for the master. The numbers in Fig. 1 have been generated this way. Additional experiments with a double-threaded master led to almost identical results, despite the higher consumption of resources.
- Second, the largest basin is mapped to the most powerful, in our case the 8-processor, node. As the batch script controls the assignment of MPI processes to nodes, the master program was easily adapted to distribute work accordingly.
- Third, only the largest basins profit from the use of mul-

tiple threads to overcome bottlenecks, whereas the smaller basins gain more if the two processors of a node are used for two processes (since inter-basin parallelism yields higher speedups than intra-basin parallelism).

Using heterogeneity, we reduced the running time of WaterGAP down to a minimum of 99 seconds on the Kassel cluster, which corresponds to a speedup of 22 as compared to a sequential run on the fastest node. This speedup is obtained with a total of 32 processors: 6 processors of the powerful node for the largest basin, 2 processors of the powerful node for the second largest basin, 2 processors of another node for the third largest basins, and 22 single-threaded processes for all other basins and the master. In this run, the bottleneck was observed at the largest basin. Using 8 threads for this basin, though, the second largest basin became a bottleneck. Further speedups are therefore dependent on the availability of a second multiprocessor node with more than two processors.

V. Malleability with MPI-2

As stated before, another problem on our target cluster is idling of processors, because the batch system has to wait for the availability of all requested processors before application startup. We solved this problem by making the program malleable, i.e., the program starts with a small number of processes, and later incorporates more and more processes when the processors become available. The master/slave pattern is well-suited to malleability, as the master can easily distribute the tasks to an increasing number of slaves. When a slave dies, the master could also reassign its task, but this aspect is not relevant here. Instead of submitting a single job to the batch system, we submit several jobs, which request part of the resources each. For instance, we submit one job per MPI process.

We used MPI-2 for implementation, and therefore start this section with a brief introduction to the relevant parts of MPI-2. After that, we describe our implementation and discuss various design decisions and MPI-2 support. Briefly stated, we use a client/server scheme that complements the master/slave scheme already explained. The overall structure is depicted in Fig. 2, where edges marked 1 denote process startup, edges marked 2 represent master/slave communication for the distribution of basins, and edges marked 3 represent connection establishment by MPI-2 client/server routines. Note that master and server are different processes. The server is started in advance, while all other processes are started through the batch system. The first process takes the role of the master, and the others are slaves.

A. Dynamic Process Management in MPI-2

Whereas the original MPI-1 standard required the number of processes to be fixed at program startup, the more recent MPI-2 standard supports dynamic process management with two sets of functions:

First, `MPI_Comm_spawn` and related functions allow an MPI process to dynamically spawn new MPI processes. However, there is no obvious way to inform the existing MPI process about the fact that new processors have become available, and therefore we do not use these functions.

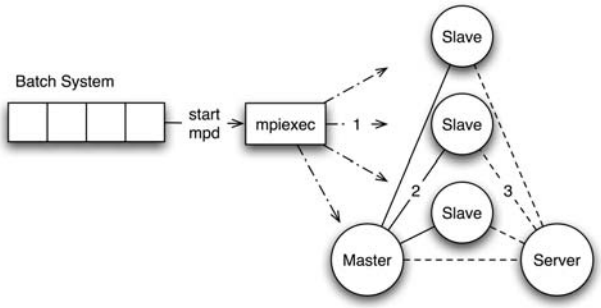

Fig. 2. Structure of malleable program

Second, MPI-2 defines a set of functions for client/server communication, somewhat similar to socket communication in networks. On the server side, functions include

- `MPI_Open_port` to establish a network address,
- `MPI_Comm_accept` to wait for a connection request from a client, and
- `MPI_Publish_name` to register port information on a name server

On the client side, functions include

- `MPI_Comm_connect` to establish communication with a server for which port information is known, and
- `MPI_Lookup_name` to retrieve port information from the name server.

After successful connection, the matching accept and connect calls each return an intercommunicator, through which server and client can send and receive messages in both directions.

The concept of communicators has already been introduced in MPI-1. Communicators are data structures to store all information required for communication between a group of processes. MPI distinguishes intracommunicators and intercommunicators. Intracommunicators number processes consecutively, for easy access. At program startup, the intracommunicator `MPI_COMM_WORLD` is predefined and comprises all processes started. Many MPI programs rely solely on `MPI_COMM_WORLD`.

The communicator returned after connection establishment is an intercommunicator, i.e., a bridge between two groups of processes. Note that both client and server may be composed of multiple processes. In this case, `MPI_Comm_accept` and `MPI_Comm_connect` have to be invoked by all processes of the respective group, and the calls return only after successful connection establishment (in MPI terms, the functions are collective and blocking).

Intercommunicators are more difficult to handle than intracommunicators, but MPI provides a function called `MPI_Intercomm_merge` to transform an intercommunicator into an intracommunicator. This function is collective and blocking over both groups. Communicators can be released using the function `MPI_Comm_disconnect`.

B. Master and Server Processes

We now return to the malleable WaterGAP version. From the running application's point of view, new processes become available at any point in time, and should enter computation as soon as possible. The master can not call `MPI_Comm_accept` itself, as the function is blocking and

would therefore prevent it from doing other work. Thus, the function must be called in a separate thread of control, the server (see Fig. 2). There are two opportunities for implementation:

- master and server are different threads of the same process, or
- master and server are separate processes.

In both cases, the master must be informed when a new slave becomes available, i.e., communication is required from server to master. This communication is rather difficult to accomplish in OpenMP, as it requires repeated inquiries and synchronization on the master's side. In MPI, communication is more natural: According to the master/slave scheme, the master runs a loop and receives a message from any slave in each loop iteration. The scheme can be easily extended to let the master receive a message from either a slave or from the server. With MPI communication between master and server, the only gain from running both in the same process would be the immediate availability of the intercommunicator returned by `MPI_Comm_accept` to the master. This gain is outweighed, however, by the opportunity to start the server in advance, which the second scheme provides (we will explain later why this is useful). Therefore, master and server are run in different processes.

We use a separate executable for the server, but the same executable for master and slaves. Thus, a job can dynamically decide to take over the role of master or slave. When an MPI process is started, it first connects to the server, who sends back the current value of a process counter. If this value is zero, the new process branches into the master code, otherwise into the slave code. In either case, the value is stored as a unique process number.

C. Communicators

In the non-malleable WaterGAP version, all communication has been accomplished through one intracommunicator: `MPI_COMM_WORLD`. In particular, the master used `MPI_Iprobe(...MPI_ANY_SOURCE...)` in its main loop, to wait for a message from any slave. MPI does not support a similar wait for a message from any communicator. To avoid major changes in the existing code, it would therefore be desirable to have a communicator for the group of all processes, as a replacement for `MPI_COMM_WORLD`. Although MPI provides constructor functions for communicators, these must be invoked in a collective and blocking call, i.e., all processes must carry out the functions at the same time, and the already existing slaves would have to interrupt their work for that. As a possible solution, the constructor may be invoked by a separate thread of each slave. In the hybrid MPI / OpenMP program, however, this thread is unrelated to the existing OpenMP threads, which would lead to an involved program structure with, e.g., nested parallelism.

Therefore, we have decided against a global communicator, despite the drawback that we had to go through the whole program and adapt all MPI functions to the new communicator structure. This structure uses as many communicators as slaves, each of them comprising the master and one slave; plus another communicator for master and server. All communicators are intracommunicators. We now describe how they are constructed.

As stated before, any new process first connects to the server and is assigned a process number. If the new process is the master, the returned intercommunicator is transformed into an intracommunicator and the former released.

If the new process is a slave, the server informs the master about its existence, by sending a message. In the main loop of the master, we have replaced the original `MPI_Iprobe(...MPI_ANY_SOURCE...)` by a loop that cycles through all communicators, thereby checking if a message from either the server or a slave is available. When the master receives a message from the server, it invokes accept to establish a separate connection with the slave and build a communicator that connects master and slave only.

D. MPICH2 in the Batch System

In addition to incorporating MPI-2 functions into the program, we had to deal with process startup through the batch system. The MPI-2 standard leaves process startup to the implementation.

The MPICH2 implementation [4] of MPI-2 includes a process management environment. Before an application is started with `mpiexec`, an MPI daemon, called mpd, must be invoked on each participating machine. Moreover, the set of daemons must form a ring. Daemons are started with the user commands `mpdboot` and `mpd`, where `mpdboot` starts a ring of daemons on a set of machines, and `mpd` starts a single daemon. As a parameter to `mpd`, one can specify the hostname / port of an existing mpd ring, which will connect the new daemon to this ring. Port information can be obtained by invoking the user command `mpdtrace -l`, on a machine that is already included in the ring.

As a first approach, we considered starting a cluster-wide mpd ring in advance, to avoid the need to start a new daemon whenever the batch system assigns a job. With this approach, many daemons are running idle, though (especially on a large cluster), and therefore, we decided to start an mpd only after a node has been assigned to WaterGAP by the batch system.

Daemons are started with `mpd`, and require hostname / port information to connect to the existing ring. This information must be provided in the job script, but it is not available before an initial mpd ring has been set up. For ease of implementation, we therefore decided to start the server (including mpd) in advance, using a cluster node with permission for interactive access.

The malleable MPI program is not bound to a particular size of the jobs, i.e., in the sequence of jobs that form a WaterGAP run, any job script may request any number of nodes. If a script requests several nodes, mpd's are started one at a time, and afterwards all processes are started with a single call to `mpiexec`.

E. Evaluation of MPI-2

In summary, our experiments have shown that it is possible to build malleable applications with MPI-2 and the MPICH2 implementation, and to run these applications through a batch system. Performance is difficult to measure as it depends on the cluster load. While our context only requires to increase the number of processors, we ob-

served in experiments that the program can to some extent also cope with the event of slaves being killed during program execution. In this case, the master proceeds with its main loop, communicating with existing and new processes despite the fact that `MPI_Iprobe` looks for messages from a dead slave.

To achieve malleability, we had to apply major changes to the original MPI communication structure. Instead of using the single communicator `MPI_COMM_WORLD`, we had to introduce a set of communicators (one per process), and therefore could not use the convenient `MPI_Recv(... MPI_ANY_SOURCE ...)` calls anymore. The alternative of constructing one communicator for all processes proved difficult, as communicator constructor functions are collective and blocking. Use of an additional thread would have complicated the hybrid MPI / OpenMP structure. In summary, we missed non-blocking communicator constructor and destructor functions, as well as a non-blocking variant of `MPI_Comm_accept`.

Another drawback of using a set of communicators instead of `MPI_COMM_WORLD` is lacking support for collective communication. In the WaterGAP program, we had to replace an initial broadcast of a parameter from the master to the slaves by pairwise communications.

Altogether, malleable programming requires a higher programming expense. Among the reasons for that (besides the already mentioned ones) is the need to manage the ring of mpd's. Moreover, debugging for errors in the communicator constructor and destructor calls (e.g. for a missing `MPI_Comm_disconnect`) is time-consuming.

VI. Related Work

Hybrid MPI / OpenMP programming has already been studied for several applications. Smith and Bull [12] identify situations in which hybrid programming is superior to a pure MPI approach, among them load balancing problems with MPI, and ease of implementation for a parallelization in multiple dimensions. Rabenseifner [11] classifies hybrid programs, WaterGAP belongs into the *hybrid masteronly* category. Spiegel and an Mey [13] dynamically vary the number of threads in different processes to improve load balancing on a shared-memory architecture. Numerical programs have been adapted to a heterogeneous cluster by Aliaga et al. [2]

The need for malleability in the context of grid computing was pointed out by, e.g., Mayes et al. [8], who designed a performance control system for applications that consist of malleable components. They implemented malleability by interrupting an MPI program, saving its state, and later restarting the program with a different number of processes. Hungershöfer [5] has shown that the availability of malleable jobs improves the performance of supercomputer schedulers. He referred to multithreaded programs.

VII. Conclusions

In this paper, we have described our experiences in parallelizing a simulation program for global water prognosis. The parallelization has been accomplished in three steps. First, in the base version of the program, we implemented a simple master/slave scheme in MPI. This program did not scale beyond 8 slaves, because of a bottleneck at the largest task. Therefore, in the second step, we mixed MPI and OpenMP to exploit two levels of parallelism. The hybrid scheme was most efficient when used with a varying number of threads depending on task size, yielding a maximum speedup of 22 with 32 processors. In the third step, we made the program malleable, as to better use the batch system and avoid wasting hardware resources.

The paper discussed our deployment of MPI-2 dynamic process management functions for malleability, and pointed out difficulties such as lack of non-blocking communicator constructor functions. The performance gain from malleability is difficult to quantify, as it depends on the cluster load. The investigation is therefore left for future research. In this context, it would also be interesting to dynamically adapt the master's task assignment to differences in computational power of the available and requested processors.

The WaterGAP program can be further improved, in particular with respect to I/O, and maybe intra-basin parallelism. Another venue for future research is evaluating MPI-2's support for malleability with other applications.

Acknowledgements

We thank the Center for Environmental Systems Research, especially Kerstin Schulze, Frank Kaspar and Lucas Menzel, for providing the WaterGAP code and answering our questions about WaterGAP. We are grateful to our colleague Björn Knafla for proofreading the paper. The University Computing Centers in Kassel and Frankfurt provided the computing facilities used for our experiments.

References

[1] J. Alcamo and other. Development and testing of the WaterGAP 2 global model of water use and availability. *Hydrological Science*, 48(3):317–337, 2003.

[2] J. I. Aliaga et al. Parallelization of the GNU scientific library on heterogeneous systems. In *Int. Symp. on Parallel and Distributed Computing, HeteroPar Workshop*, pages 338–345, 2004.

[3] W. Gropp et al. *MPI : The Complete Reference*. MIT Press, 1998.

[4] W. Gropp et al. *MPICH2 User's Guide, Version 1.0.3*, November 2005. Available at http://www-unix.mcs.anl.gov/mpi/mpich2.

[5] J. Hungershöfer. On the combined scheduling of malleable and rigid jobs. In *IEEE Symp. on Computer Architecture and High Performance Computing*, pages 206–213, 2004.

[6] F. Kaspar. *Entwicklung und Unsicherheitsanalyse eines globalen hydrologischen Modells*. PhD thesis, Universität Kassel, 2003.

[7] C. Leopold. *Parallel And Distributed Computing: A Survey of Models, Paradigms, and Approaches*. John Wiley & Sons, 2001.

[8] K. R. Mayes et al. Towards performance control on the grid. *Philosophical Transactions of the Royal Society*, 363(1833):1793–1805, 2005.

[9] MPI-2: Extensions to the message-passing interface, 1997. Available at http://www-unix.mcs.anl.gov/mpi.

[10] OpenMP Application Programming Interface Version 2.5, 2005. Available at http://www.openmp.org.

[11] R. Rabenseifner. Hybrid parallel programming on HPC platforms. In *European Workshop on OpenMP*, pages 185–194, 2003.

[12] L. Smith and M. Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2–3):83–98, 2001.

[13] A. Spiegel and D. an Mey. Hybrid parallelization with dynamic thread balancing on a ccNUMA system. In *European Workshop on OpenMP*, pages 77–82, 2004.

[14] M. Süß and C. Leopold. *Observations on the Publicity and Usage of Parallel Programming Systems and Languages: A Survey Approach*, 2006. In preparation.

[15] G. Utrera, J. Corbalán, and J. Labarta. Implementing malleability on MPI jobs. In *Proc. Parallel Architectures and Compilation Techniques*, pages 215–224, 2004.

[16] WaterGAP Home Page. http://www.usf.uni-kassel.de/usf/forschung/projekte/watergap.en.htm.