Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments

Claudia Leopold and Michael Süß

University of Kassel, Research Group Programming Languages/Methodologies
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
{leopold, msuess}@uni-kassel.de

Abstract. Large-scale MPI programs must work with dynamic and heterogeneous resources. While many of the involved issues can be handled by the MPI implementation, some must be dealt with by the application program. This paper considers a master/slave application, in which MPI processes internally use a different number of threads created by OpenMP. We modify the standard master/slave pattern to allow for dynamic addition and withdrawal of slaves. Moreover, the application dynamically adapts to use processors for either processes or threads. The paper evaluates the support that MPI-2 provides for implementing the scheme, partly referring to experiments with the MPICH2 implementation. We found that most requirements can be met if optional parts of the standard are used, but slave crashes require additional functionality.
Keywords: dynamic process management, malleability, adaptivity, hybrid MPI/OpenMP, master/slave pattern

1 Introduction

Traditionally, MPI programs have used a fixed number of homogeneous processes. Modern architectures and especially grids, in contrast, are characterized by dynamic and heterogeneous resources: Nodes can crash, be withdrawn by the scheduler in favor of higher-priority jobs, or join a running computation after having finished a previous task. Moreover, different nodes may comprise a different number of processors.

The ability of applications to dynamically adapt to a changing number of processors is often denoted as *malleability.* This term goes back to Feitelson and Rudolph [1], who classify jobs as rigid, moldable, evolving, or malleable. Both evolving and malleable jobs change the number of processors during execution, evolving jobs for internal reasons such as requesting additional processors for a complicated subcomputation, and malleable jobs in reaction to changes caused by the environment.

Many architectures combine shared-memory within the nodes and distributed-memory in-between the nodes. They can be programmed in a hybrid style, using MPI processes that are composed of threads. Whether or not the processors of a node are more profitably used for processes or threads, depends on the application. It may be useful to change this assignment dynamically. We call this feature *process-thread adaptivity.*

This paper evaluates the support for malleability and process-thread adaptivity that is provided in MPI-2, mainly through the dynamic process management functions. We base our discussion on a hybrid MPI/OpenMP application from the simulation domain, which is described in Sect. 2. The application uses a master/slave scheme, in which slaves correspond to MPI processes that internally deploy a different number of OpenMP-threads.

Previous work by the same authors has shown that additional processes can be dynamically incorporated into this application [2]. The present paper adds the aspect of process-thread adaptivity, and discusses the case of slaves leaving the computation prematurely. We show that MPI-2 provides sufficient support for process-thread adaptivity if the implementation covers some optional parts of the standard. Evolving programs are supported as well, but the case of a slave leaving the computation abruptly can not be handled appropriately, and we discuss possible workarounds.

Sect. 2 of the paper starts with an outline of the application, including parallelization and deployment of hybrid processes. Then, Sect. 3 explains at an algorithmic level our modifications of the master/slave scheme to handle dynamic and heterogeneous resources. The realization of this scheme in hybrid MPI-2/OpenMP is the topic of Sects. 4–6: Sect. 4 recalls the program structure for incorporating additional processes, Sect. 5 discusses process-thread adaptivity, and Sect. 6 is devoted to node withdrawals. Related work is reviewed in Sect. 7, and the paper finishes with conclusions in Sect. 8.

2 Application and Experimental Setting

The example program, called WaterGAP, computes current and future water availability worldwide [2]. WaterGAP partitions the surface area of continents into equally-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), over a period of several years. The program has been written in C++.

WaterGAP uses two levels of parallelism: a master/slave scheme implemented with MPI at the outer level, and data parallelism implemented with OpenMP-threads at the inner level [2]. The master/slave scheme relies on the observation that the set of grid cells is naturally partitioned into basins that do not exchange water with other basins. Thus, the overall computation is divided into independent tasks that correspond to one basin each. Task sizes are known in advance, but range from a few very large tasks to many small ones.

Scalability of the master/slave scheme is limited, since the program can not run faster than the time needed to compute the largest basin. Therefore, data parallelism is used to speed up the computation of large basins internally. Data parallelism yields lower speedups than master/slave parallelism [2], i.e., if a multi-processor node is assigned one large basin, it finishes earliest when using a multi-threaded process. If the same node is assigned several small basins, it finishes earlier when using several single-threaded processes. Therefore, we use a different number of threads for different processes.

Experiments were carried out on the compute cluster of the University of Kassel, a Linux cluster that comprises a large number of double-processor nodes, and one eight-processor node. On this architecture, a maximum speedup of 22 was achieved with 32 processors [2]. Here, the largest basin was computed by a multi-threaded process, other large basins were computed by double-threaded processes, and the small basins were computed by single-threaded processes.

In all experiments, we used the Portland Compiler, and the MPICH2 [3] implementation of MPI-2 (release 1.0.3, process manager `mpd`, compiled with Portland compiler). Experiments were carried out both interactively and through the batch system. We experimented with both C and C++ bindings of the MPI functions.

3 Dynamic and Heterogeneous Master/Slave Scheme

The standard master/slave scheme uses one master and several slaves. The master starts computation by sending a task to each slave. Whenever a slave has finished its task, it reports the result back to the master and gets the next task, until all tasks have been processed. We modify the scheme to incorporate:

– dynamic arrival of slaves,
– arrival of more powerful slaves that can take over expensive tasks, and
– sudden or announced withdrawal of slaves.

The first case is easy to handle at an algorithmic level: the master adds the slave to its pool of communication partners, and sends a task. The other two cases require task reassignment. While one can think of very sophisticated and efficient schemes, we restrict our considerations to a simple scheme here that is sufficient to identify and study essential requirements for MPI support:

After creation, a new process connects to the master and requests work. The master assigns the tasks by size, starting with the largest task. To keep track of the state of computation, it stores for each process: task currently assigned to, size of this task (in grid cells), and number of processors. The latter is sent to the master with the slave's work request.

Although tasks are assigned strictly in order of decreasing size, an assignment may be a better or worse fit. A good fit maps a large basin to a process with many processors, or a small basin to a process with a single processor. Architecture-specific thresholds specify the meaning of terms large etc. In our setting, basins are classified as large, medium, or small, depending on the number of grid cells; slaves are classified as powerful (8 processors), normal (2 processors), or weak (1 processor). We speak of a good fit for combinations large-powerful, medium-normal, and small-weak.

One case of a bad fit assigns a large basin to a weak slave (combinations large-weak, large-normal, and medium-weak). Here, the slave starts computing, but when a more powerful slave arrives later on, the master reassigns the basin. As MPI-2 provides no means for the master to signal this event to the first slave, the slave occasionally asks whether there was a reassignment. If so, it abandons its work and requests a next task from the master.

The reverse case that a small or medium basin is assigned to a powerful slave (combinations small-powerful, small-normal, and medium-powerful), occurs only when all larger basins have already been assigned before. Hence, after receiving the basin, the slave splits itself up into multiple processes. One process computes the basin, and the others request more work from the master, i.e., become separate slaves. The splitting generates weak processes when the assigned basin is small, and normal processes when it is medium.

Moreover, tasks are reassigned when the master learns that a slave has died, and will therefore not finish its task, or when the task pool is empty, but some results have not been received yet. When several slaves are computing the same basin and one has found the result, the others are abandoned as soon as they report back.

One case needs particular consideration: reassignment of a large basin (from a dead slave) after powerful slaves have been split up into groups of weak ones. The master stores the grouping of processes, keeping the original process as a leader. To assign a task to the group, it requests all processes except the leader to exit (when they report having finished their present task). Then, it assigns the task to the leader, who spawns new threads.

4 Dynamic Integration of Processes

A program version that allows for dynamic integration of slaves has been described by us [2]. It uses an additional process, called server, that invokes the accept function and helps in communicator construction. All communication is accomplished through intracommunicators that connect two processes each: master and slave, or master and server. The construction of a single communicator for all processes proved difficult, since communicator constructor functions are blocking and collective. Busy slaves can not call these functions, except in a separate thread, which would interfere, however, with the internal OpenMP structure for data parallelism.

All occurrences of `MPI::COMM_WORLD` had to be replaced by pairwise communicators. Since there is no `MPI::ANY_COMM`, the master waits for a message from any communicator with loop

```
while (!isMessage) {
    rank = (rank + 1) % total;
    isMessage = comms[rank].Iprobe(...);
}
```

where `comms` is an array of all intracommunicators.

5 Process-Thread Adaptivity

The modified master/slave scheme poses two requirements:

- A slave that runs on a multi-processor node must be able to dynamically spawn either processes or threads on the same node.
- A slave must be able to exit computation after receiving a termination request from the master.

For the first requirement, a slave must know how many processors it owns. Although the OpenMP function `omp_get_num_procs` yields the number of physical processors, it is possible that only part of them are available to the application. Thus, information must be passed from the resource manager (e.g. batch system) to the MPI application. MPI-2 defines a constant `MPI::UNIVERSE_SIZE` for that purpose, but leaves it to the implementation to set its value or not. The MPICH2 implementation sets the value to parameter `usize` of `mpiexec`. We use this parameter to provide to each slave the number of processors it owns.

Spawning the corresponding number of threads is a simple call to the OpenMP function `omp_set_num_threads`. Processes are spawned with `MPI::Comm::Spawn`. The new processes can not rely on `MPI::UNIVERSE_SIZE`, but get the number of processors from their parent, through an argument of the spawn function. These processes also differ from the processes started with `mpiexec` in that they are connected to their parent with a communicator. We close this communicator immediately, and then handle all processes the same way.

Threads are always spawned on the same node. Processes, in contrast, may be spawned on any node that is available to the MPI system. This placement may be inappropriate as the system can not take the existence of threads into account (especially if they have not been created yet). To keep track of the available resources, we always spawn processes on the same node as their parent. MPI-2 supports that with the reserved `info` key `host`, which is an optional part of the standard again.

The second issue (slaves exit computation) is easy to resolve. As will be further discussed in the next section, the slave first disconnects from the rest of the program, by closing the master-slave intracommunicator, and then calls Finalize. Since this function is collective over the set of connected processes only, the slave returns immediately.

6 Termination of Processes

For evolving processes, i.e., program-initiated termination, the exit of slaves is easy. The principle has already been explained in the last paragraph. It relies on the fact that a slave is connected to the rest of the program through a single communicator between master and slave only. All other communicators are closed immediately after their creation. Thus, the slave can disconnect, without enforcing any other process to participate in this blocking and collective operation. Note that a process is connected to all processes in `MPI::COMM_WORLD`, but we start each process with a separate call to `mpiexec`, and so `MPI::COMM_WORLD` is a singleton.

After termination, the master must exclude the slave from its pool of communication partners, since Iprobe does not work with a null communicator. Also, the basin must be reassigned to another slave. With this scheme, a slave may leave computation at any time, either in reaction to a termination request by the master, or on request of the resource manager (provided that the resource manager can pass the request to the slave).

Implementation of malleability, in contrast, is problematic. When a slave suddenly dies, it is not able to call Disconnect nor Finalize. The MPI standard states that "'if a process terminates without calling Finalize, ... the effect on connected processes is not defined"'. Thus, it may happen that a single faulty process brings the whole application down.

According to our experiments, the MPICH2 implementation is more robust. When a slave dies, the master's message-waiting loop (see Sect. 4) continues without any problem, just not receiving messages from the dead slave anymore. We tested this feature by running each process in a separate window, killing one with `Ctrl-C` (during a computation phase), and observing the output. The behavior was the same in the batch system, with an exit call in one slave's code.

Using the reassignment scheme described in Sect. 3, the program manages to compute all tasks and generate the complete output. Nevertheless, we did not find a correct way to finish the program. The MPI standard requires that each process calls Finalize, which is a collective and blocking operation over connected processes. While a slave can disconnect from the rest of the program and terminate as described above, the master can not disconnect from a dead slave. Consequently, its call to Finalize does not return. The standard defines the function `MPI::Abort` to kill processes, but the behavior of this function is not specified in detail. In our experiments, this function did not return either. The only way we found to let the program return, was to omit the Finalize call from the master. Then termination works fine, except for an error message, but this workaround of course conflicts with the standard.

The termination problem can probably be solved by clarifying the behavior of `MPI::Abort`. An alternative solution relies on a communicator clean function that eliminates all dead processes from the communicator, i.e., live processes are disconnected from dead ones, and dead processes do not need to take part in any future collective operation. Such a function may either be provided by the MPI API, or be invoked implicitly by the MPI implementation. The implicit variant is already provided by Fault Tolerant MPI or FT-MPI [4]. It comfortably solves our termination problem since after cleaning, the master can call Finalize. In FT-MPI, communication functions return an error code after a communication partner has crashed. This mechanism solves a second problem: notification of the master in the event of slave death. As the master regularly contacts all slaves in the message-waiting loop, it learns about the crash soon and can reassign the basin immediately. Unfortunately, FT-MPI supports only part of MPI-2.

7 Related Work

The process termination aspect of malleability has been discussed under the heading of fault tolerance, e.g. in a survey paper by Gropp and Lusk [5], and in FT-MPI [4].

Much work on malleability was carried out in the scheduling community, where it was shown that malleability significantly improves system throughput in both supercomputers and grids [6, 7]. Two approaches for making MPI programs malleable have been followed: 1) checkpointing, i.e., interrupting the program, saving its state, and later restarting it with a different number of processes [7], and 2) folding, i.e., using a fixed number of processes, and coping with changes in the number of processors by varying the number of processes per processor [8]. We are not aware of other experience reports on making an application malleable with the MPI-2 dynamic process management routines.

Outside MPI, research on handling node crashes with the master/slave scheme has been done with PVM [9] and Java [10]. The more general divide-and-conquer pattern is considered by Wrzesińska et al., in a Java-based framework [11]. They suggest a scheme to avoid redoing work that another process already did before crashing. None of this work considers multi-threaded processes or process-thread adaptivity.

Except for malleability, hybrid MPI/OpenMP programming is well understood [12, 13], including dynamic variations in the number of threads per process for better load balancing [14].

8 Conclusions

This paper has discussed MPI-2 support for dynamic and heterogeneous processes, on the basis of a hybrid master/slave application. The master/slave scheme was modified to dynamically add processes, and reassign tasks when powerful slaves arrive or slaves exit. We observed that MPI-2 supports integration of slaves and process-thread adaptivity, provided that the implementation covers optional parts of the standard: the constant `MPI::UNIVERSE_SIZE` and the `info` key `host`. Termination, in contrast, requires active participation of a slave, or functionality beyond the MPI standard to eliminate dead slaves from a communicator, and to notify the master after slave crashes.

In experiments, the malleable program performed better than the original one, mainly because it started before all desired resources were available. Malleability and process-thread adaptivity come at the price of higher programming overhead and a performance penalty. For the master/slave example, the programming overhead was reasonably low, but this may be different for applications that require algorithmic changes such as data redistribution. The performance penalty is due to the overhead for additions and withdrawals of nodes, the need to use pairwise communicators instead of `MPI_COMM_WORLD`, the bookkeeping overhead at the master, and task reassignment costs. Our application has a high computation-to-communication ratio, and thus the overhead was not an issue.

Future research may address improvements of the simple master/slave scheme referred to in this paper. For instance, the master may restrict use of multi-threaded slaves to basins that would otherwise delay the overall computation. It may also cooperate with the resource manager to get a forecast of resources. Notification of slaves after reassignment may use one-sided communication instead of pairwise communication-based polling. Furthermore, checkpointing may be integrated. Finally, the scheme may be refined to handle the case that the master dies. Sophisticated master/slave patterns may be implemented in a skeleton library, which may extend to other malleable patterns.

References

1. Feitelson, D.G., Rudolph, L.: Toward convergence in job schedulers for parallel supercomputers. In: Job Scheduling Strategies for Parallel Processing, Springer LNCS 1162 (1996) 1–26
2. Leopold, C., Süß, M., Breitbart, J.: Programming for malleability with hybrid MPI-2 and OpenMP: Experiences with a simulation program for global water prognosis. In: High Performance Computing & Simulation Conference. (2006) 665–670.
3. Gropp, W., et al.: MPICH2 User's Guide, Version 1.0.3. (November 2005) Available at http://www-unix.mcs.anl.gov/mpi/mpich2.
4. Fagg, G.E., et al.: Process fault-tolerance: Semantics, design and applications for high performance computing. Int. Journal of High Performance Computing Applications **19**(4) (2005) 465–478
5. Gropp, W., Lusk, E.: Fault tolerance in message passing interface programs. Int. Journal of High Performance Computing Applications **18**(3) (2004) 363–372
6. Kalé, L.V., Kumar, S., DeSouza, J.: A malleable-job system for timeshared parallel machines. In: IEEE/ACM Int. Symp. on Cluster Computing and the Grid. (2002) 230–237
7. Vadhiyar, S.S., Dongarra, J.J.: SRS: A framework for developing malleable and migratable parallel applications for distributed systems. Parallel Processing Letters **13**(2) (2003) 291–312
8. Utrera, G., Corbalán, J., Labarta, J.: Implementing malleability on MPI jobs. In: Proc. Parallel Architectures and Compilation Techniques. (2004) 215–224
9. Goux, J.P., et al.: An enabling framework for master-worker applications on the computational grid. In: IEEE Int. Symp. on High Performance Distributed Computing. (2000) 43–50
10. Baratloo, A., et al.: Charlotte: Metacomputing on the web. In: Int. Conf. on Parallel and Distributed Computing Systems. (1996) 181–188
11. Wrzesińska, G., et al.: Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In: IEEE Int. Parallel and Distributed Processing Symposium. (2005)
12. Smith, L., Bull, M.: Development of mixed mode MPI/OpenMP applications. Scientific Programming **9**(2–3) (2001) 83–98
13. Rabenseifner, R.: Hybrid parallel programming on HPC platforms. In: European Workshop on OpenMP. (2003) 185–194
14. Spiegel, A., an Mey, D.: Hybrid parallelization with dynamic thread balancing on a ccNUMA system. In: European Workshop on OpenMP. (2004) 77–82