In Proceedings of the SCS Symp. on Performance Evaluation of Computer and Telecommunication Systems (SPECTS '98), Reno, USA, July, 1998, ©1998 SCS, reprinted with permission of the SCS

DISTRIBUTED SIMULATION OF MULTICOMPUTER ARCHITECTURES WITH MERMAID

A.D. Pimentel L.O. Hertzberger

Dept. of Computer Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{andy,bob}@wins.uva.nl

Keywords: Computer performance, simulators, distributed simulation, multicomputers

ABSTRACT

This paper describes the parallelization of the Mermaid multicomputer simulation environment. Due to our simulation methodology, this parallelization is reasonably straightforward as it does not require any measures to guarantee the causality within the simulated system. The resulting distributed simulator increases the simulation performance without any loss of simulation accuracy. Furthermore, the parallel simulation environment is also more scalable than its sequential counterpart with respect to the memory requirements. This gain in performance and scalability can be used for simulating larger target architectures and more realistic applications. Experiments with parallel Mermaid confirm that distributed simulation may lead to significant performance improvements compared to sequential simulation. In several cases, we even measured super-linear speedups.

1. INTRODUCTION

In order to explore the design space of multicomputer architectures, we developed the Mermaid simulation environment[Pimentel and Hertzberger, 1997b]. This environment allows the performance evaluation of a wide range of architectural design options by means of parameterization: from processor parameters, such as the cache specifics, to switching and routing techniques in the message-passing communication network.

Mermaid differs from other simulation environments in the way it addresses the tradeoff between accuracy and simulation efficiency. Presently, most multicomputer and multiprocessor simulators apply the *direct execution* technique to obtain high simulation performance [Rosenblum et al., 1995, Reinhardt et al., 1993, Boothe, 1993, Brewer et al., 1991, Davis et al., 1991, Covington et al., 1991]. In this technique, "uninteresting" instructions are not explicitly simulated but are directly executed by the simulating host computer. This requires, however, the instruction set of the host computer to be similar to that of the modelled architecture.

Besides direct execution, some simulators, like SimOS [Rosenblum et al., 1995], also provide multiple levels of simulation. This enables the architect to position the simulation at an interesting state using a fast and abstract level of simulation. Thereafter, the interesting section is studied using an accurate, and thus less efficient, mode of simulation.

Mermaid does not perform direct execution. Until now, it has applied two other techniques to address the accuracy-efficiency tradeoff. First, we offer the ability to simulate at different abstraction levels. But, unlike SimOS, the whole simulation takes place at one abstraction level only. So, if the research objective is fast prototyping, maximum accuracy is not required and simulation can be performed at a high level of abstraction. On the other hand, if accuracy is required, then simulation is performed at a lower and more computationally intensive abstraction level.

Second, at its lowest level of abstraction, Mermaid simulates abstract instructions rather than interpreting and simulating real machine instructions. For this purpose, we use some kind of trace-driven simulation. Compared to traditional instruction-level simulation, this approach typically results in a higher simulation performance at the cost of a small loss of accuracy. As a consequence, we obtain a simulation efficiency which is quite competitive with many direct execution simulators [Pimentel and Hertzberger, 1997a]. But, unlike the direct execution simulators, our simulation approach does not make demands upon the simulating host architecture and is therefore more flexible.

An alternative approach to boost simulation performance even more is followed by the Wisconsin Wind Tunnel (WWT) [Reinhardt et al., 1993]. Besides using direct execution, this simulator also exploits the inherent parallelism found in simulations of parallel computers. By performing *distributed* *simulation* of the parallel architecture, it tries to gain performance without sacrificing any simulation accuracy. The WWT has shown that distributed simulation may improve the simulation performance considerably. In fact, the WWT is one of the fastest, if not the fastest, multiprocessor simulator available. The performance gained by distributed simulation allows for the study of larger target architecture configurations and more realistic applications. As a side-effect, distributed simulation also improves the simulator's scalability with respect to the memory consumption. Because the memory requirements are spread over multiple host computers, it is possible to simulate workloads that consume large amounts of memory, which is common for multicomputer applications.

We strongly believe that the next logical step in improving Mermaid's efficiency and scalability is to extend it in order to support the distributed simulation of multicomputer architectures. For this reason, we implemented a prototype of *parallel Mermaid* which allows the simulation to be distributed and executed on a cluster of workstations.

In this paper, we describe the parallel Mermaid prototype. We show that the parallelization is quite straightforward due to our modelling methodology. For example, the causality constraint [Fujimoto, 1990], which is a common problem in the field of parallel (discrete-event) simulation, is automatically taken care of in our case. Moreover, we present performance results which indicate that substantial speedups can be obtained with our distributed simulator.

The next section gives a brief overview of the Mermaid simulation environment. Section 3 describes how the multicomputer simulations are parallelized to run on a cluster of workstations. In Section 4, the performance results of a set of benchmarks are presented. Finally, Section 5 concludes the paper and mentions possible future work.

2. THE SIMULATION ENVIRONMENT

The multi-layered simulation environment of Mermaid is shown in Figure 1. The lowest level, referred to as the architecture level, contains the architecture simulation models. These models are implemented in a highly modular fashion using the object-oriented simulation language Pearl [Muller, 1993], which allows flexible evaluation by means of parameterization. The simulators are driven by traces of abstract instructions, called *operations*, representing processor activity, memory I/O and message-passing communication.

Simulating at the level of operations has several consequences. As the operations abstract from the processors' instruction sets, the simulators do not have to be adapted each time a processor with a different instruction set is simulated. Furthermore, simulating operations rather than interpreting real instructions allows for only modelling the *timing consequences* of instruction execution. Most of the state transitions

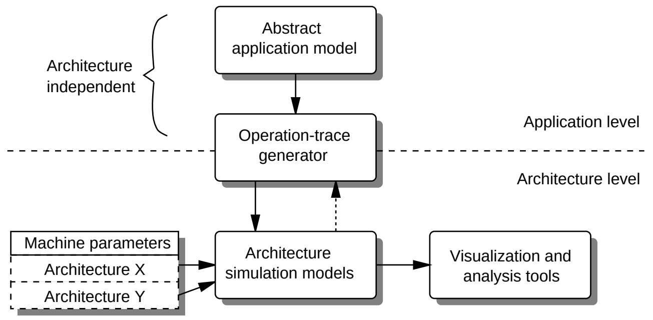

Architecture independent
Abstract application model
Operation-trace generator
Application level
Architecture level
Machine parameters
Architecture X
Architecture Y
Architecture simulation models
Visualization and analysis tools

Figure 1. The Mermaid simulation environment.

caused by the execution, such as the actual storing of a value in a register, do not need to be modelled. Consequently, it is not necessary to store large quantities of state information during simulation runs. For example, register and memory contents do not have to be modelled and simulated caches only need to hold addresses (tags), not data. As a result, our approach may yield higher simulation performance compared to the more traditional instruction-level simulation techniques. On the other hand, the strength of abstraction is also Mermaid's weakness. The loss of information at the level of operations (e.g. registers are not specified in operations) prohibits an *accurate* low-level simulation of, for example, the processor pipelines. In [Pimentel and Hertzberger, 1997a], we demonstrate that this small loss of accuracy still allows for accurately simulating multicomputers. The average errors we measured in that validation study do not exceed 5%.

To provide the architectural simulators with operation-traces representing actual application behaviour, an abstract application model and a trace generator reside on top of the architecture level (see Figure 1). The trace generator produces a separate trace for each processor within the multicomputer model. To produce these multiple operation-traces, the trace generator mimics concurrent execution by means of threads. Each thread accounts for the behaviour of one processor within the parallel machine. In this scheme, the validity of the operation-traces is guaranteed by the fact that the simulator can give *feedback* to the trace generator. Doing so, the simulator may stall and resume the trace-generating threads, thereby establishing execution-driven simulation [Pimentel and Hertzberger, 1997b].

The abstract application model specifies workloads in an architecture-independent manner. These workloads are either based on real applications or they are synthetic. Realistic workloads are obtained by tracing real programs, whereas the synthetic ones are generated from probabilistic descriptions of application behaviour [Pimentel and Hertzberger, 1997a]. Because the latter technique is flexible but only represents application behaviour with modest accuracy, it is typically used for fast prototyping. In this paper, we only focus on the realistic type of workloads generated by tracing real programs.

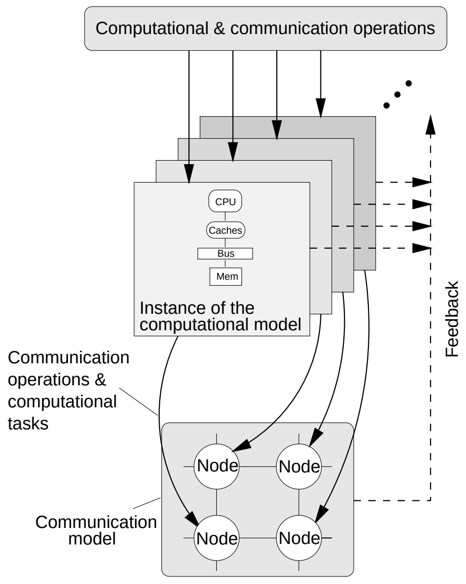

Computational & communication operations
CPU
Caches
Bus
Mem
Instance of the computational model
Feedback
Communication operations & computational tasks
Node
Node
Node
Node
Communication model

Figure 2. Multicomputer simulation using both the computational and the communication models.

2.1. A Hybrid Architecture Model

Many applications, and especially scientific applications, running on multicomputer platforms contain coarse-grained computations alternated with periods of communication. Because these computation and communication phases typically are distinct, Mermaid splits the simulation of multicomputers into two different models: a single-node *computational model* and a multi-node *communication model*. Each model operates at a different level of detail, and thus defines its own set of operations. The computational model simulates the application's computational behaviour. It models the incoming computational operations at a level of abstract machine instructions. Communication operations are not simulated by this model, but are directly forwarded to the communication model. The communication model accounts for the application's message-passing behaviour. To address the issues of synchronization and load-balancing properly, it models the computational delays found between communication requests at the task level. A parallel workload for this model therefore resembles a graph containing computational tasks and communication operations. The computational tasks are derived from the computational model, which constructs them by measuring the simulated time between two consecutive communication operations.

This approach results in a *hybrid model* which allows for simulation at different abstraction levels. If accuracy is required, then the complete hybrid model can be used. In this case, the single-node computational model is *replicated* for each simulated node. Each instance of the single-node model is then assigned to a node within the communication model in order to feed it with the computational tasks and the communication operations. This is illustrated in Figure 2.

If there is only the need for fast prototyping, then just using the communication model might be sufficient. In that case, the task-level operation-traces must be directly produced by the trace generator. In this study, however, we only consider the whole hybrid simulation model for parallelization. We believe that the exclusive parallelization of the communication model is not required since this model already simulates workloads almost as fast as the target machine executes them [Pimentel and Hertzberger, 1997a] and it does not consume large amounts of memory. Furthermore, the communication model only simulates communication in detail; computation is modelled by simply advancing the simulation clock. As the simulation of communication may exhibit poor locality and generally is not computationally intensive (i.e. it is not coarse-grained), parallelizing the communication model would introduce large communication overheads.

3. PARALLEL MERMAID

Mermaid's hybrid architecture model, as depicted in Figure 2, exhibits a lot of inherent parallelism, which simplifies the parallelization. The instances of the computational model perform computations which are local to a single node only. So, these instances are independent of each other (i.e. their synchronization and communication is simulated in the communication model only). As a consequence, they can easily be simulated in parallel on different hosts. Subsequently, when the communication model is not parallelized and is still executed on a single host computer, this scheme does not require any measures to guarantee the causality in the simulated system. In other words, using this approach, no communication events will ever take place out of order. This is because all communication is performed by the sequential communication model which has a global notion of simulation time. It therefore correctly sequentializes the incoming communication requests and, as a result, still supplies the trace generator with consistent feedback. Hence, due to our simulation methodology, there is no need for algorithms, like the ones discussed in [Fujimoto, 1990], to synchronize the distributed simulation clocks. This is in contrast to, for instance, the Wisconsin Wind Tunnel which uses a conservative algorithm to guarantee causality [Reinhardt et al., 1993].

When applying the above parallelization scheme, it seems that the (sequential) communication model could become a potential bottleneck. For two reasons, however, we think this is unlikely to occur. First, the number of communication requests typically is much smaller than the number of computational operations. This is especially true for the application domain in which we are interested, namely that of (sci-

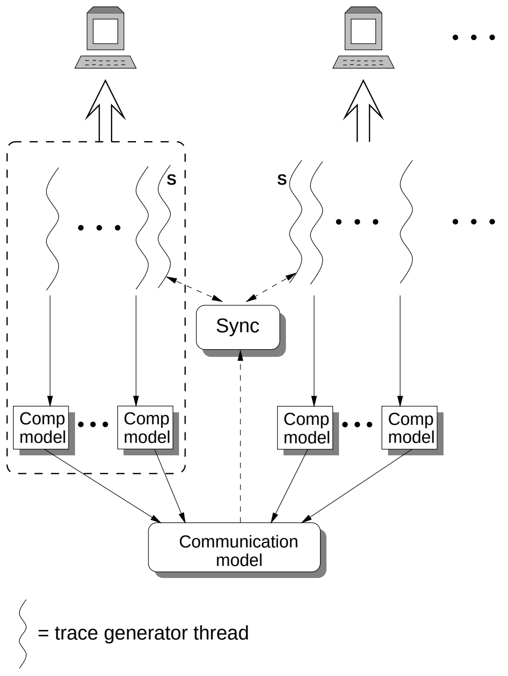


Figure 3. Distributed simulation with Mermaid.

entific) computationally intensive applications. For applications which are constrained by communication (which will probably not run very well on multicomputers anyway), the communication model may indeed limit the parallel simulation performance. Note that although communication is not extremely dominant in our applications, the communication model is still essential for correctly modelling the synchronization behaviour of the applications. Second, many multicomputer applications, and specifically the ones belonging to the popular class of SPMD programs, contain fixed communication patterns. So, their communication behaviour is not dependent on the underlying architecture. This type of applications allows for synchronizing the trace-generating threads at the application level rather than synchronizing them with simulator feedback, which will be discussed further on. As a result, the simulation of communication does not need to be execution-driven. Instead, the communication model can now operate in pure trace-driven mode, thereby not constraining the execution of the threaded trace generator.

The question that remains is whether or not to parallelize the trace generator. We decided to perform its parallelization for reasons of scalability. By dividing the generator threads over multiple workstations, the threads' memory requirements are spread over multiple machines as well. This makes it possible to scale the simulation to larger and probably more realistic applications.

The distributed version of the hybrid simulation model is shown in Figure 3. The trace-generating threads are, together with the instances of the computational model which they feed, spread over multiple workstations. This division of work can be performed according to any distribution scheme. But, in this paper, we assume that the work is evenly shared among all the hosts. As was mentioned before, the communication model is executed on a single host machine. When necessary, the distributed threads of the trace generator are synchronized by a process called SYNC. This synchronization can be performed in two ways. If execution-driven simulation of the communication model is required, then the SYNC process provides the threaded trace generator with the feedback from the architectural simulator. To send the simulator feedback to the appropriate thread, SYNC is connected to all participating workstations. On the other hand, if the communication model does not need execution-driven simulation, then SYNC can perform synchronizations between trace-generating threads directly. To do so, remote threads are able to receive and send messages from/to each other via SYNC to synchronize and to exchange data in order to keep their notion of local data consistent. For example, if a processor sends a message of size N to another processor, then the communication model simulates this by sending an "empty" message of size N. The actual data is transferred by SYNC. Of course, this is more efficient than explicitly simulating the data transmission. Note that if the workload execution is not dependent on the transferred data, then SYNC does not have to perform the data transfers (nor any synchronizations) at all. In that case, it does not matter when a thread's local data is inconsistent.

To coordinate these types of control, each distributed part of the trace generator has one extra thread, called the *S-thread*, which takes care of the communication between the trace generator and SYNC. The S-thread will, for instance, signal (and possibly wake up) a trace-generating thread when data for it has arrived from another, distant thread.

Typically, the SYNC process is placed onto the same host as the communication model. Communication between the different components within this distributed environment is performed by either Unix sockets or shared memory, dependent on the location of the communicating processes. For example, the threaded trace generator communicates via shared memory with the computational model, whereas each instance of the computational model uses a socket to talk to the communication model.

4. EXPERIMENTS

To evaluate the performance of parallel Mermaid, we use a model which represents a multicomputer consisting of T805 transputers [Inmos, 1992] connected in a two-dimensional mesh. For this architecture, we simulated the operation-traces of three SPMD-type benchmark applications: *gauss* (a solver of linear equations), *pdmm* (a matrix multiplication) and *sort* (an integer sort). A more detailed description of the bench-

Benchmark	Description	Data sizes
Gauss	A solver of linear equations using Gaussian elimination	Matrices of 64×64 and 128×128
Pdmm	A double-precision matrix multiplication	Matrices of 64×64 and 128×128
Sort	An integer odd-even transposition sort	32K and 64K of integers

Table 1. The benchmark applications used for the performance evaluation of parallel Mermaid.

marks can be found in Table 1. Of these three benchmarks, none requires execution-driven simulation (i.e. they all contain fixed communication patterns) and only the execution of *sort* is data dependent. The latter implies that, when simulating *sort*, the SYNC process transfers data between remote threads to keep their notion of local data consistent. So, in the case of the other two benchmarks, SYNC is simply not used.

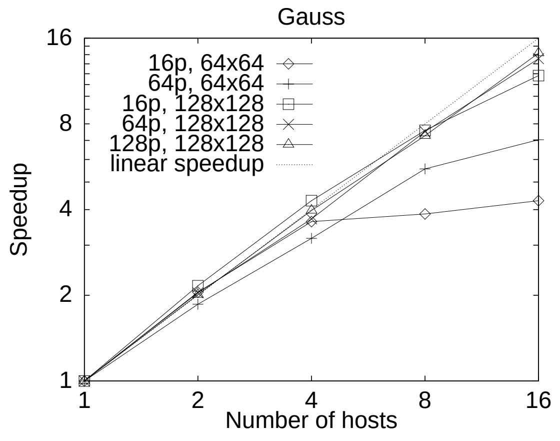
The experiments were performed using multicomputer configurations of 16 processors (*16p*), 64 processors (*64p*) and, where possible, 128 processors (*128p*). The cluster of host workstations over which the simulation is distributed consists of sixteen 110-Mhz Sun Sparc-4s connected by normal Ethernet. These are not particularly high-end machines, but they form one of the few lightly loaded, homogeneous clusters of workstations with acceptable performance at our department.

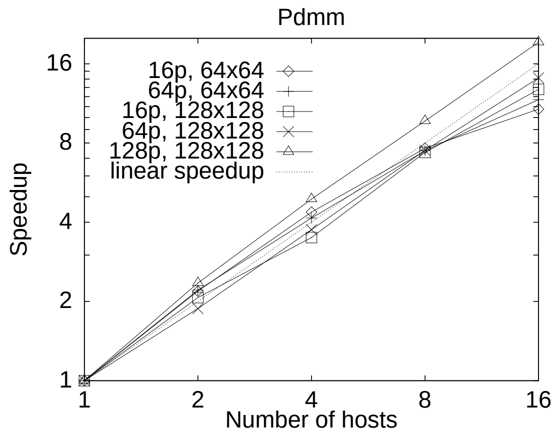
Figure 4 shows the speedups of the parallel simulation environment for the set of benchmarks. These measurements were performed using wallclock times. For the parallel simulations, this includes both the actual simulation time and the time it takes to distribute the processes over the multiple hosts. Note that both axis have a logarithmic scale.

Gauss

The graph at the top of Figure 4 presents the results for *gauss*. These results clearly indicate that most of the obtained speedups are substantial. For instance, the *128p* configuration with 128×128 matrices is simulated 14.2 times faster on 16 hosts than it is simulated on a single host. Moreover, for nearly all simulations using two hosts we measured a superlinear speedup. This is most probably due to caching effects. Only the simulations using 64×64 matrices fail to obtain significant speedups beyond 4 host platforms. Apparently, the grainsize of these workloads is too small.

The fact that the multicomputer configurations containing more processors perform better than those with fewer processors is caused by the increased overheads in the simulations of larger configurations. This will be elaborated upon when discussing the results of *pdmm*.





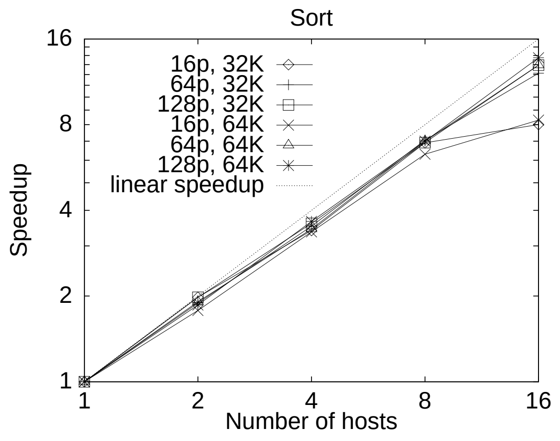


Figure 4. Performance results of parallel Mermaid.

Pdmm

For *pdmm*, of which the results are shown in the middle graph of Figure 4, the parallel performance is even better. All sim-

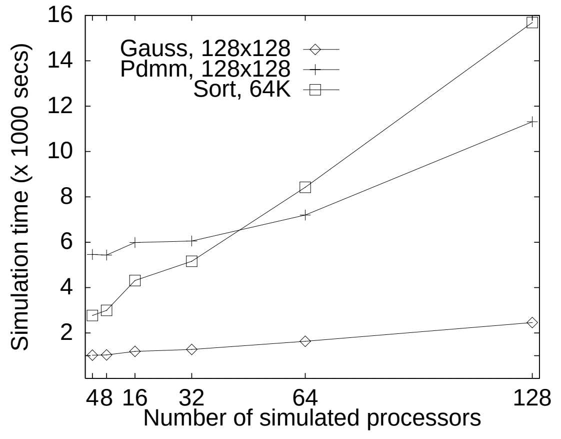


Figure 5. Simulation time v.s. the size of the simulated multicomputer on a single host.

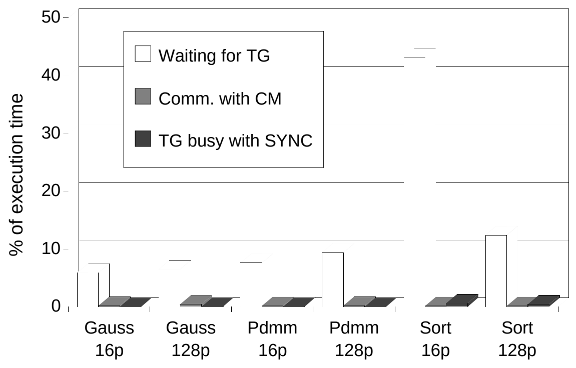


Figure 6. Breakdown of where the execution time goes due to certain parallel overheads. The white bar shows the average time the computational model waits for the Trace Generator (TG), the grey bar the average time the computational model communicates with the Communication Model (CM). Finally, the black bar represents the average time the TG is busy transferring data via the SYNC process. All numbers are percentages of the total execution time. These experiments are for 128x128 matrices and 64K of integers on a cluster of 16 hosts.

ulations properly scale with the increasing number of hosts. Using 16 hosts, for example, speedups of between 10.75 and 19.3 are obtained. Thus, again super-linear speedups were measured. In fact, all parallel *pdmm* simulations of the *128p* configuration with matrices of 128$\times$128 obtain super-linear speedups. To explain this, consider Figure 5. For each benchmark, this graph shows the simulation time as a function of the number of simulated processors on a single host. The graph demonstrates that the simulation time of all benchmarks is at least doubled when the multicomputer simulation is scaled up from 4 to 128 processors. This is caused by the increase of certain overheads in the sequential simulation, such as the thread overhead. Since the overheads per host within the distributed simulation are smaller (i.e. the overheads are parallelized), super-linear speedups might be obtained. In the case of *pdmm*, the simulation time for a single host starts to increase significantly after 64 processors. This corresponds with the super-linear speedup of *128p* in Figure 4.

Sort

The graph at the bottom of Figure 4 shows the results of *sort*. Again, substantial speedups are obtained. Although Figure 5 shows an impressive increase of simulation time for larger multicomputer configurations, no super-linear speedups were measured for *sort*. This is due to the synchronizations between the remote trace-generating threads, which cover up the gain of parallelizing the overheads. To illustrate this, consider Figure 6. This graph shows a breakdown of the average overheads in the parallel instances of the trace generator and the computational model. These results are for the *16p* and *128p* simulations of all three benchmarks on 16 hosts with the largest data sizes. Three types of parallel overheads are distinguished: the time the computational model waits for operations from the trace generator (white bars), the time the computational model communicates with the communication model (grey bars) and the time the trace generator is busy with transferring data via the SYNC process (black bars). The white bars in Figure 6 demonstrate that, in the case of *sort*, the computational model generally waits longer for the trace generator compared to the other two benchmarks. This can be explained by the fact that a trace-generating thread may be suspended (i.e. waiting for remote data) for a quite a while before being resumed by the remote thread. These synchronization overheads do not occur in the other two benchmarks, as they do not perform remote data transfers. Nevertheless, it may seem that the difference in overhead between the simulations of, for example, *pdmm*'s and *sort*'s *128p* configuration is only marginal. Note, however, that the results are averages and we measured a standard deviation (σ) of 3.4% for *sort*, while *pdmm*'s σ only equals to 0.2%. The large σ for *sort* is caused by a few parallel instances suffering from high synchronization overheads of which the largest equals to 22% of the execution time.

Figure 6 shows that this synchronization overhead is extremely high for the *16p* simulation of *sort*: the computational model waits on the average for more than 40% of the execution time for the trace generator. So, this must the reason for the poor parallel performance of the *16p* simulations on 16 hosts (see Figure 4). It is, of course, not surprising that this particular configuration suffers from these overheads as there resides only one simulated processor on each host. Thus, when suspending a thread, no other thread can take over and the simulation on the host is entirely stalled.

Due to the higher, parallelizable overheads of sequential simulation of larger configurations and the existence of multiple trace-generating threads per host, the performance of the *64p* and *128p* simulations does scale up after 8 host machines.

4.1. Discussion

For the simulated benchmarks, the results indicate that the sequential communication model does not constrain the parallel simulation performance. As Figure 6 shows (grey bars), the overhead of the communication between the computational model and the communication model is for all benchmarks exceptionally small. Figure 6 also demonstrates that the overhead of data transfers by SYNC is negligible (black bars). Naturally, this overhead only exists for *sort*. Note that, in this study, all necessary synchronizations within the trace generator are performed at the application level (by the SYNC process) rather than at the architecture level (by simulator feedback). We still need to investigate how architectural feedback affects the parallel simulation performance.

The performance improvements due to our extensions are very promising. Whereas the original simulators obtain a typical slowdown of between 60 and 750 per processor compared to the real machine [Pimentel and Hertzberger, 1997a], parallel Mermaid may reduce these slowdowns by an order of magnitude. Furthermore, the obtained super-linear speedups illustrate the improved scalability of parallel Mermaid with respect to its sequential counterpart. Simulating large multicomputer configurations on a single host machine may easily lead to performance degradations due to increased overheads or is simply impossible due to the excessive memory consumption of the workloads.

5. CONCLUSIONS

In this paper, we described the extension of the Mermaid simulation environment to support the distributed simulation of multicomputer architectures. Applying distributed simulation may boost the performance without any loss of accuracy. Furthermore, parallel simulations are more scalable than sequential ones as a single host platform may easily run out of memory when simulating large parallel applications.

We demonstrated that Mermaid can be parallelized in a straightforward manner. No measures were necessary to guarantee the causality within the simulated system. This is due to our simulation methodology, in which a sequential communication model has a global notion of time and thus correctly sequentializes all communication requests.

For a set of benchmarks, we showed that parallel Mermaid obtains significant speedups. The average speedup, for instance, for all performed simulations with 16 host computers is nearly 12. Moreover, in some cases, we even measured super-linear speedups caused by caching effects and the parallelization of overheads.

5.1. Future Work

In this study, we used a rather limited set of benchmarks. This benchmark suite should be extended in the future. With respect to this, we should also investigate the impact of workloads that require execution-driven simulation. Furthermore, the experiments were done using a homogeneous pool of 16 workstations. It would be interesting to investigate whether or not parallel Mermaid scales up beyond these 16 hosts. Moreover, in practice, heterogeneous workstations with different computational powers might be used. This would accentuate the importance of load-balancing strategies to optimally divide the computations over the hosts.

References

[Boothe, 1993] Boothe, B. 1993. Fast accurate simulation of large shared memory multiprocessors. Tech. Report CSD 92/682, Comp. Science Div., Univ. of California at Berkeley (June).

[Brewer et al., 1991] Brewer, E. A., Dellarocas, C. N., Colbrook, A., and Weihl, W. E. 1991. PROTEUS: A high-performance parallel-architecture simulator. Tech. Report MIT/LCS/TR-516, MIT Laboratory for Computer Science (Sept.).

[Covington et al., 1991] Covington, R. G., Dwarkadas, S., Jump, J. R., Sinclair, J. B., and Madala, S. 1991. The efficient simulation of parallel computer systems. *Int. Journal in Comp. Simulation*, 1:31–58.

[Davis et al., 1991] Davis, H., Goldschmidt, S. R., and Hennessy, J. 1991. Multiprocessor simulation and tracing using Tango. In *Proc. of the Int. Conf. in Parallel Processing*, pages 99–107 (Aug.).

[Fujimoto, 1990] Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53.

[Inmos, 1992] Inmos 1992. *The Transputer Databook*. Inmos Ltd.

[Muller, 1993] Muller, H. L. 1993. *Simulating computer architectures*. PhD thesis, Comp. Sys. Dept., Univ. of Amsterdam (Feb.).

[Pimentel and Hertzberger, 1997a]
Pimentel, A. D. and Hertzberger, L. O. 1997a. Abstract workload modelling in computer architecture simulation. In *Proceedings of the 3rd Workshop on Performance Analysis and its Impact on Design (in conjunction with ISCA'97)*, pages 6–14 (June).

[Pimentel and Hertzberger, 1997b]
Pimentel, A. D. and Hertzberger, L. O. 1997b. An architecture workbench for multicomputers. In *Proc. of the 11th Int. Parallel Processing Symposium*, pages 94–99 (April).

[Reinhardt et al., 1993] Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A. 1993. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. of the 1993 ACM SIGMETRICS Conf.*, pages 48–60 (May).

[Rosenblum et al., 1995] Rosenblum, M., Herrod, S. A., and Gupta, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 3(4):34–43.