PATHS - Integrating the Principles of Method-Combination and Remote Procedure Calls for Run-Time Configuration and Tuning of High-Performance Distributed Applications

John Markus Bjørndalen[1], Otto Anshus[1], Tore Larsen [1], Brian Vinter[2]

Department of Computer Science [1]
University of Tromsø

Department of Mathematics and Computer Science [2]
University of Southern Denmark

Abstract

A "path" based on the idea of method-combination and remote procedure calls to provide run-time configurable networks of computational communication paths between threads and data in distributed, high performance applications is proposed. An initial design is implemented, tested and analyzed.

We use a "wrapper" to provide a level of indirection to the actual run-time location of data by forwarding function calls to servers holding the target data. A wrapper specify where data is located, how to get there, and which protocols to use. Wrappers are also used to add or modify methods accessing data. Wrappers are specified dynamically. A "path" is comprised of one or more wrappers. Sections of a path can be shared among two or more paths. Establishing a path is a two-phase process of specifying the path, and then (recursively) setting up the path based on the specification.

A test system using the proposed architecture is implemented, demonstrated and performance measured using two benchmarks on three different clusters. We show that the proposed architecture can be used for mapping and improving the performance of applications on different topologies including a wide-area multi cluster configuration, and how wrappers can be used to distribute computational load off of heavily loaded target servers. We also show how thread distributions can be changed at run-time without altering application code.

We believe that the proposed semi-manual approach will prove useful in debugging and coarse-tuning distributed high-performance applications, and that it will provide valuable insights for developing later, more automated, middleware systems.

1 Introduction

A key challenge when running distributed high performance applications is to maintain thread-to-host mappings that achieve high performance or efficient execution. Attacking this challenge requires balancing the potentially conflicting goals of distributing threads for improved load balancing and for reduced communication overhead.

In reality, high scalability cannot be achieved unless the system is fine-tuned to balance computation, communication, and synchronization requirements. Unfortunately, high performance is often achieved

only after rigorous manual fine-tuning to obtain an efficient mapping of threads to hosts.

Efficient thread-to-host mappings may be achieved by directives in the application source-code, reflecting the topology of the host architecture in the application source-code. Alternatively, mappings may be obtained by communication libraries or by middleware. Static or dynamic mappings may be used depending on application dynamics, the homogeneity of the underlying architecture, or the regularity of the interconnection topology. Dynamic mappings accommodate changing needs over the application lifetime by use of costly thread migrations. In this case, the original placement problem is transformed into a problem of determining where and when any thread should be migrated.

One alternative to compile-time static placement and runtime dynamic placement is to implement a static placement on an irregular architecture by deciding on a mapping at load-time. A "manual" approach to the load-time solution is to allow the application programmer to instruct the middleware as to the distribution and communication patterns, specified either explicitly by the programmer or chosen from a library of algorithms. We have combined the static load-time and the dynamic run-time approaches. We allow for run-time placement, but we do this typically at the startup of the application by using a configuration map loaded with the application. However, the application can change this mapping at will if it so wishes.

This paper describes our initial work on a middleware extension inspired by method combination[8] and remote procedure calls[3] which allows the communication topology to be directed by specifying meta-code and meta-data, without introducing any modifications to the application code. The extended middleware also allows computations to be placed along the access paths to data. For now, we assume that the application under study is alone in using the underlying architecture; there are no other applications competing for resources. The goal of the mapping then, is to achieve high performance for one single application running alone.

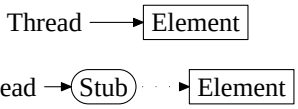
We show how one may experiment with different mappings of threads and their intercommunication, and demonstrates that this can identify flexible location policies which are independent of the application code and still supply effective placements.

Section 2 describes how wrappers are used and combined to specify and identify access paths. Section 3 describes our experiments using three different clusters located at the University of Tromsø, Norway and the University of Southern Denmark, Odense. Section 4 presents and analyzes the experiment results, section 5 presents related work, while section 6 presents our conclutions.

2 The configurable path framework

Our research platform uses the PastSet[1][13], a structured distributed shared memory system in the tradition of Linda[4]. A PastSet Element is a tuple space with tuples of the same or equivalent types, and is globally addressable with a name and the type of the tuples residing within it. PastSet also supports X-functions[14], which can be used to modify the behavior of the PastSet operations (to implement functionality such as, but not limited to, global reductions and caches).

A common way to implement remote access to shared objects such as a PastSet Element is to give the accessing thread a stub which forwards the operations to the remote server. The stubs interface is identical to the interface of the acessed element, providing transparent access to both local and remote elements.



A stub does not necessarily have to be limited to implement remote access though. It can just as easily be used to modify the semantics of an elements operations by implementing one of the PastSet X-functions (such as the global reduction sum). If stubs can be combined, we can easily set up a remote element which is used to compute global sums as follows:

We call the combination of the stubs a *path* to the remote element. The combination of all existing paths to an element forms a tree with threads as the leaves and the element as the root.

As long as the application only uses the reference to the topmost stub, it can be mapped to another cluster configuration without changing the application code. Fitting and optimizing the application to a particular configuration can instead be done by changing the path-building metadata and code.

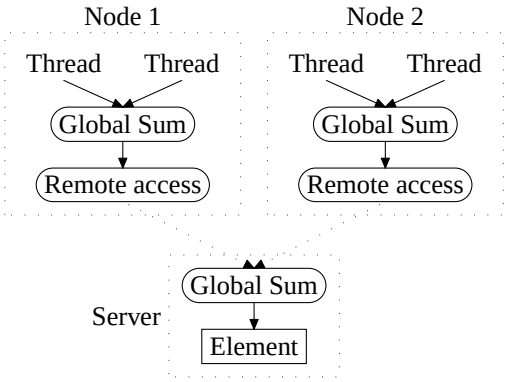


Figure 1: Threads in two nodes accessing a shared global sum element. Each node computes a partial sum before forwarding it to the global sum wrapper in the server.

2.1 Building and specifying paths

Setting up access from a thread to a PastSet element involves the following two stages:

1. Specify the the path. This involves examining information about the cluster topologies, the location of threads in the cluster and where the target element is located.

2. Build the path from the description. This involves creating and binding the wrappers with parameters specified in the path description.

To allow configurability of the wrappers, we include parameters for each wrapper in the path description. Some of these parameters are common for all wrapper types (such as whether the wrapper need to use thread synchronization mechanisms), or type specific (such as the protocol to use, remote address and service requirements in a remote access wrapper). Parameters not specified are assigned default values.

An example path description used by one of the nodes in Figure 1 is included in Figure 2. `build_path` builds the given path and returns a reference to the toplevel wrapper in the path.

Each thread creates (or is given) its own path description and calls `build_path` to get its own reference to the path. `Build_path` takes care of merging paths when the path descriptions allow for sharing parts of the path.

2.2 Current implementations

The current implementations use Common Lisp and Python for management of paths, while the PastSet applications and wrappers are implemented in C.

This allows us the flexibility of high level dynamic languages for experimenting with path building code, while keeping the high-level languages out of the loop when benchmarking the different configurations.

```
path = make_path(stage("reduce-sum", num_threads=2),
                 stage("remote", proto=TCP, host="p0"),
                 stage("reduce-sum", num_threads=2),
                 stage("core", name="PI-SUM1"))
elm = build_path(path)
```

Figure 2: Example path description

The work reported in this paper is based on experiments with the Python framework, which allows the path framework to be provided and extended either through embedded Python (by overloading two default functions for acquiring and releasing a path to an element) or, as we did, by handing path references to C algorithms written as Python extension modules. The latter method allows different Python scripts to experiment with path building and thread spawning using the same compiled C code for all experiments.

The wrappers can also be used directly from the high-level languages allowing, for instance, Python scripts direct access to tuples and elements.

Simple profiling of PastSet operations is provided with two trace wrappers. The "timestamp" trace wrapper simply forwards the operation to the next stage in the path and uses the Pentium timestamp counter to timestamp the start and completion time for each operation invoked. The timestamp data is logged to an array in memory and written to disk when the trace wrapper is deleted (reference counting is used to determine when wrappers should be deleted). The overhead of this wrapper is around 100-120 clock cycles.

The "operation" trace wrapper is an extension of the timestamp trace wrapper which additionally logs the contents of the tuples provided to or returned from the PastSet operations.

Any number of trace wrappers can be inserted anywhere in the path trees.

3 Experiments

To show how the framework can be used for mapping and optimizing an application to different topologies, we devised experiments to map two benchmarks to different path trees and two different thread allocation policies. We used these experiments to examine some choices which can be made when mapping an application to a cluster or multi-cluster environment.

We believe that changing the mapping will produce performance benefits because of the different emphasis put upon locality, load balancing and communication. Also, the potential mismatches between the collective data access patterns by all the threads, and each processor's individual data cache will be influenced by different mappings. Of course, the application will play a role in how successful a mapping is. For instance, frequent use of synchronization using locks, and especially global locks, will play a role in the resulting performance and the effect of a mapping.

Two basic benchmark codes were used:

- The **Global Sum benchmark** (GSum), which measures the average execution time of a global sum operation. The number of values to sum is equal to the number of threads used in the experiment.

- **Monte Carlo Pi** (MCPi), which computes an approximation of Pi by randomly throwing a number of darts and counting those hitting inside a circle.

  A total number of N darts are thrown by the threads, splitting the darts

evenly between the threads.

The time of throwing the N darts and running a global sum with the results is measured. N was 10 million for the cluster tests, and 100 million for multi-cluster tests. The problem is large enough that the communication latency should be masked by the time spent in the computation.

Figure 3 shows pseudocode for the benchmarks. The *TS()* macro samples the pentium timestamp counter, and stores the timestamp in an array. *gettimeofday()* samples the real-time clock on the host computer with microsecond resolution. The gsum benchmark was run with "iters" set to 1000. For both tests, the average of 5 benchmark runs are plotted in the graphs.

Based on the two benchmarks, we deviced the following experiments:

- Scaling on one node. Measure the execution time of a global reduction when we vary the number of threads from 1 to 16 on a single node.

- Thread placement and topology optimization. Two different thread distribution algorithms are used to assign threads to nodes in the clusters.

  The path trees were also varied to experiment with computing partial sums within partitions of the clusters to reduce the work and communication on the node hosting the target element.

- Monte Carlo Pi in cluster and multicluster configurations. Verify that the application can be mapped and scaled to the three clusters and when using the three clusters together in a multicluster configuration.

- Multicluster global sum. Measure the execution time of global sum using all three clusters with 3 to 96 threads. Threads are assigned evenly among the clusters.

The benchmark code was unchanged during the experiments, we only changed parameters and metadata for the Python framework code used to map threads and set up the paths.

Available for the experiments were 3 clusters with 32 processors in each, organized as follows:

- 2W cluster - 16 * 2-Way (Dual) Pentium III 450 MHz, 256MB RAM, Location: Odense, Denmark.

- 4W cluster - 8 * 4-Way (Quad) Pentium Pro 166 MHz, 128MB RAM, location: Tromsø, Norway

- 8W cluster - 4 * 8-Way Pentium Pro 200 Mhz, 1GB RAM, location: Tromsø, Norway

In addition, the root node for the multicluster experiments was a dual Pentium II 300 MHz machine with 256MB RAM located in Tromsø. One experiment was also run on a 650 Mhz Pentium III notebook (Dell Latitude CPx, 256MB RAM) to get results for a single-processor node.

For the current experiments, we only used TCP/IP over 100MBit ethernet for intra-cluster communication. The 4W cluster was connected to the root node through a HP 100 VG anylan switch, while the 8W cluster was connected to the root node using the departments local area network. The intra-cluster for the 8W cluster was a switch connected to the departments LAN.

The connection between Tromsø and Odense was the departments internet backbone.

4 Results

Figure 4 show the difference in execution time of the two different thread distribution algorithms. The *even distribution* algorithm distributes threads evenly among the nodes in the cluster. It starts with the first node and adds one thread to each node before it

```
barrier_sync();
gettimeofday();
TS(0);
for (i = 1; i= < iters; i++) {
   sum = gsum(i);
   TS(i);
}
gettimeofday();
```

(a) gsum

```
barrier_sync();
gettimeofday();
TS(0);
n_inside = mcpi(to_throw);
total = gsum(n_inside);
TS(1);
gettimeofday();
```

(b) mcpi

Figure 3: Pseudocode for the Global Sum and Monte Carlo Pi benchmarks. Only one of the threads runs the timestamp code. The others run the same code without the timestamp code in it.

goes back to the first node again. The *bucket* algorithm fills up one node with threads (number of threads equal to the number of CPUs in the node) before proceeding to the next.

As expected, once we reach 32 threads and the two distribution algorithms generate the same number of threads on all nodes, we end up with the same execution time with both algorithms.

For the 4-way and 2-way cluster, the *bucket* distribution algorithm performs better in the range between 1 to 32 threads. This is because we only need to pay the cost of bringing in a new node when the bucket algorithm has filled up the last node.

The 8-way cluster shows a different pattern though. After 4 threads, the bucket algorithm performs worse than the even distribution algorithm. The reason for this can be found in figure 5, in which the "Non-partitioned" graphs show the execution time of global sums with 1-16 threads on single nodes.

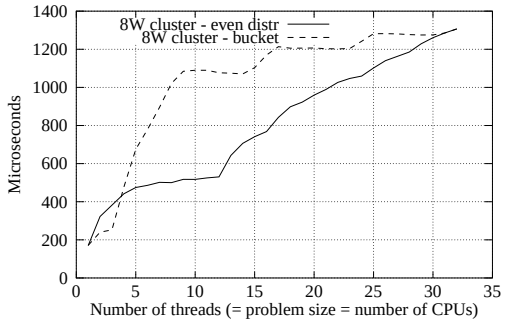
As the number of threads increase, we observe a sudden jump in execution time around 3-5 threads for the different SMP nodes. For the 8-way nodes, this execution time quickly grows over 800 microseconds when the number of threads equal the number of CPUs in the node. This shows that the internal synchronization cost for the global sum is higher than the node-to-node communication costs.

The curious jump in latency between 12 and 13 nodes for the *even distribution* algorithm on the 8-way nodes can also be explained from figure 5. At 12 threads, we have 3 threads running on each node. When we add one more thread, one of the nodes will increase to 4 threads, which corresponds to the point in the figure 5 where we get a sudden jump for the 8-way nodes. This jump is reflected in the cluster graphs since the execution time of the global sum is dictated by the slowest node.

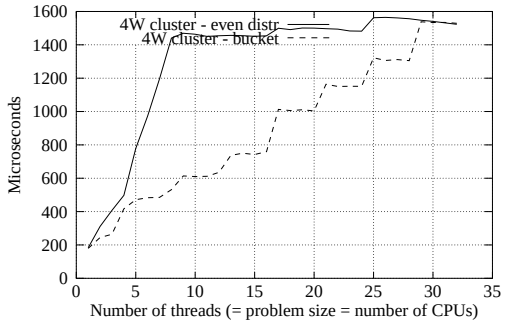
Since all the multiprocessor nodes show a distinct increase in latency once the node holds more that 3-4 threads, a natural assumption is that using partial sums might improve the execution time. The "partitioned" graphs in figure 5 shows an experiment where we limit the number of threads per sum wrapper to 4 by arranging the threads and wrappers in a hiearchial sum (see figure 6).

The graph shows that by limiting the number of threads to the range where the wrapper has the best performance, and at the same time increasing the potential parallelism, the execution time can be improved by roughly a factor two. The extra overhead of contributing through two layers of sums is less than the overhead reduced by partitioning the problem.

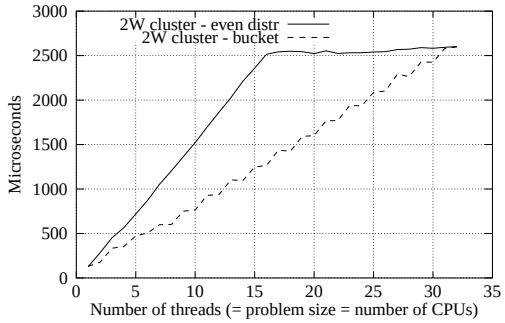
The above results suggest that partition-



(a) 8-Way cluster

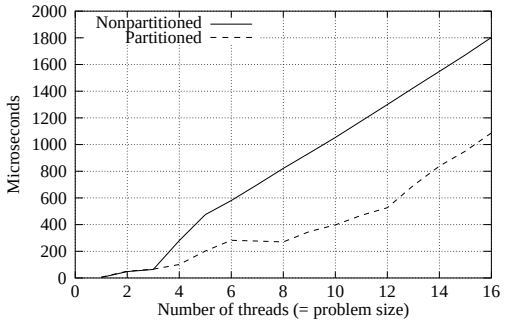


(b) 4-Way cluster

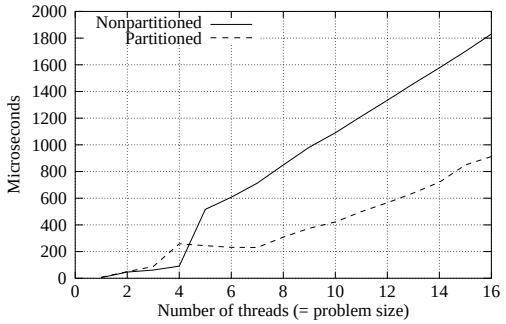


(c) 2-Way cluster

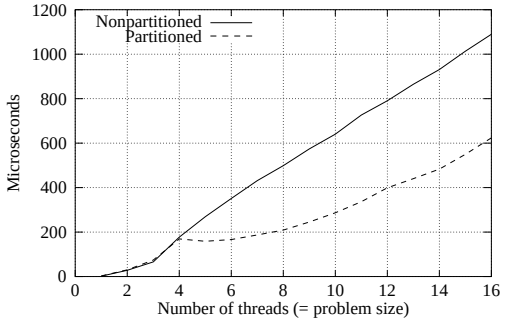
Figure 4: Execution time of global sum in each cluster using two different thread allocation algorithms



(a) Single 8-way node



(b) Single 4-way node



(c) Single 2-way node

Figure 5: Partitioning on single nodes. Single process, increasing the number of threads from 1 to 16. “Partitioned” uses a maximum of 4 threads per global sum, organizing the sum wrappers in a hierarchial partial sum tree.

ing the the clusters such that groups of nodes within the cluster contribute to a partial sum before the partial sums are added in a root node might improve the latency of the cluster, not only because of the higher level of parallelism in the cluster, but also due to a better resource usage in in the wrappers.

Figures 7 and 8 show experiments where we partitioned the path trees for the *even distribution* and *bucket* thread distribution algorithms. The path tree was first partitioned such that no sum wrapper had more than 4 contributing threads. The sum wrappers for the partitions were placed on nodes such that no node had more than one of the partial sum wrappers. This increased the network traffic from 16 to 20 roundtrip

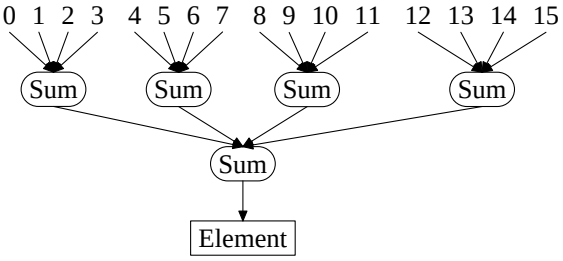
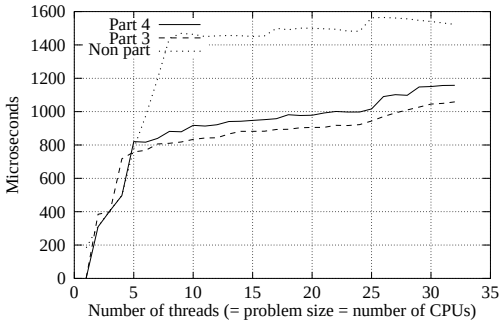
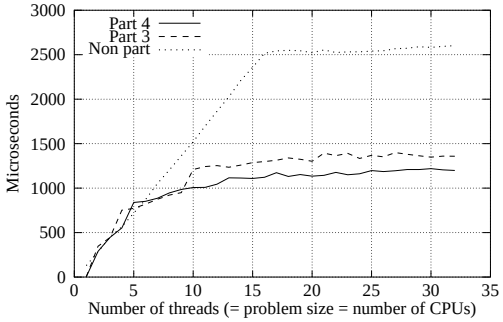


Figure 6: Hierarchial global reduction sum tree. The numbers represent threads. The upper layer of sum wrappers computes partial sums used in the lowermost sum wrapper.



(a) 4-Way cluster

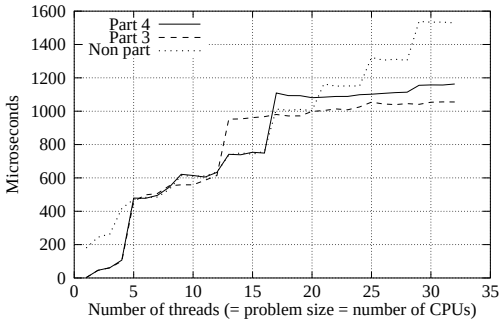


(b) 2-Way cluster

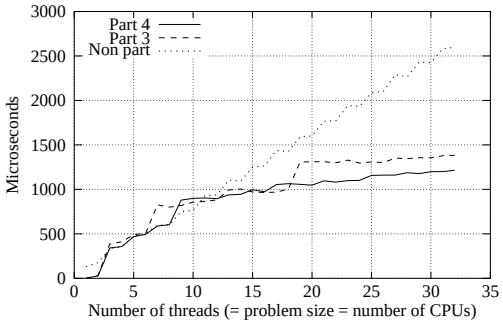
Figure 7: Cluster partition tests - even distribution

messages per sum in the 2-way cluster, and from 8 to 10 in the 4-way cluster.

Once the 4-split tests were made, we spent anoter 10-15 minutes making a map which reduced the number of threads per wrapper to 3. This added another level in the sum hiearchy and increased the number of roundtrip messages to 23 per sum for



(a) 4-Way cluster



(b) 2-Way cluster

Figure 8: Cluster partition tests - bucket

the 2-way cluster. The 4-way cluster didn't need another level, but increased the number of roundtrip messages to 11.

No tests for the 8-way cluster were made since the mechanism for partitioning the leaf threads within a node are not ready yet.

Both the 4-split and 3-split graphs show an improved operation execution time compared to the non-split graphs. For the *even distribution* graphs, we get a brakeoff at the point where the whole partial sum tree has been expanded, and only the number of threads at the toplevel is increased.

4.1 Multicluster results

Figure 9 shows the minimum, maximum and average operation execution time of a global reduction in a multi-cluster environment, going from 3 to 96 threads, at each step adding one thread to each cluster.

The figure show a signifiant variance in latency, ranging from 34 to 53 milliseconds,

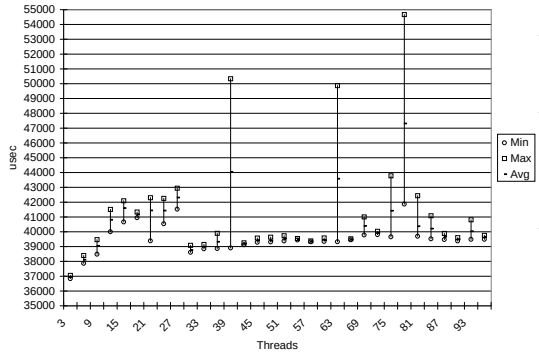


Figure 9: Multicluster global reduction

which is due to variation in the network latency between the Odense and Tromsø cluster.

The Path-framework allowed an easy hierarchical mapping of the threads in the global sum path trees, thus each reduction only generates one roundtrip message (contribute sum and retrieve result) between the Odense and Tromsø clusters, independent of the number of threads in the system. The intra-cluster latencies of the sums are so small, that they are not visible due to the high variation in latency in the Odense-Tromsø connection.

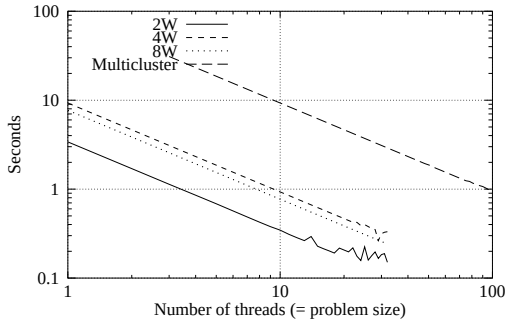


Figure 10: Monte Carlo Pi in clusters (10M darts) and multicluster (100M darts).

Figure 10 shows Monte Carlo Pi executed on the three clusters and on the multicluster. We observe the expected linear scaling for all instances. The multicluster problem is 10 times bigger than the one run on the individual clusters, and the work is divided evenly amongst the threads in the multicluster. Because the work is distributed evenly performance of the multicluster is dictated by the slowest CPUs in the system, which is the ones in the 4W cluster, as a result perfect speedup must be defined as the multicluster setup being 10 times slower than the 4W cluster, the graph clearly show this to be true.

5 Related work

Accurate and efficient performance prediction of existing distributed and parallel applications on target configurations with potentially thousands of processors is hard. Analytical solutions are difficult to develop, and many complex systems can be intractable. Simulation is a widely used tool, but its major limitation is its, often extremely, long execution time for large-scale systems. A number of simulators have been developed, including Parallel Proteus[9], LAPSE[5], SimOS[11], and Wisconsin Wind Tunnel[10]. These simulators typically are themselves parallel and use direct execution of portions of the code to reduce the cost. The slowdowns range from 2 to 100. Few simulators simulate both computation and I/O operations. In contrast to simulators, our approach execute the actual application code several times, each time with a different mapping. Of course, running an application, say, 10 times before deciding on a configuration to use, will give a slowdown of 10. However, the flexibility and simplicity is high.

In [7] it is shown that the three parallel computation models BSP, E-BSP and BPRAM in several situations do not precisely predict the actual runtime behaviour of an algorithm implementation. They report performance deviations between 25-200%. This is explained by the different approaches to communication and routing used by the models. Caching effects are also possible causes. Also, the efficiency of an implementation derived from the three

models did not match the performance possible by using hand tuned implementations. These results can be used to make a case for a system like ours where the programmer can try a few configurations and select the one giving the best performance. This can prove to be much simpler than hand coding an algorithm to utilize the hardware platform. The resulting performance will most likely not be optimal, but it can be better than not doing anything.

In [6] both processor and memory load balancing are used to support low contention and good scaling to hundreds of processors. Gang-scheduling is used to avoid wasting cycles spinning for a lock held by a descheduled process (actually, a virtual CPU). In contrast, our system is much simpler and provides for much less or no automatic support at the present time.

In [12] it is shown that there is a communication and load bralance trade-off when partitioning and scheduling sparse matrix factorization on distributed memory systems. Block based methods result in lower communication costs and worse load balancing, whereas a "round robin"-based scheme where all threads are distributed over the processors gives better load balance but higher communication costs.

In [16] an approach to load balancing for general-purpose simulations is reported in with little modification is needed to the user's code. Their approach uses runtime measurements and demonstrates better load-balancing than approaches without such measurements. Three different load-balancing mapping algorithms are used. This approach is similar to ours in that little modification of the user's code is needed. As they do, we also use different mappings and leave it to the application to control them. Our approach differ in that we can both try different mappings and add arbitrary code along the access path to data. Also, we differ in that we do a prerun of a few mappings, and then we choose a single one and we let the application use the selected mapping without incurring further overhead. Of course, we take all the overhead when choosing a mapping. For clusters where they can be dedicated to applications running often, this configuration hunting overhead will be amortizised over time.

In [2] three categories of useful tools were found when tuning the performance of NOW-Sort, a parallel disk-to-disk sorting algorithm on a cluster system: tools that help set expectations and configure the application to different hardware parameters, visualization tools that animate performance counters, and search tools that track down performance anomalies.

We believe that our system can, by simple means presently controlled by the programmer, improve performance by finding a configuration where the resource usage better avoids hot spots, bottlenecks, and expensive waiting times for processor, memory, cache, and I/O by compromising between load sharing and communication. The flexibility of using maps, paths and wrappers also make it possible to monitor the application and provide data for visualization of both behaviour and performance. At the present time we have not investigated approaches to sharing clusters among several concurrent computations.

6 Conclusion

Fine-tuning the performance of high-performance distributed applications through analytical means or simulation is hard, requiring detailed insights into the tradeoffs and effects of caching, synchronization, locality, load balancing, and communication demands.

We have proposed an approach and developed a middleware extention where different mappings of an applications communication and computations can quickly be tried out without changing the application code.

Experiments showed how we used this system to discover some of the factors con-

tributing negatively to the application performance, and then remapped the application to avoid configurations where components in the application did not scale well. We also showed how the application could be remapped to a multicluster environment without changing the application code.

The results from this work was used in [14] to benchmark PastSet using the path framework against MPI[15] (LAM-MPI), where we showed that PastSet was 83% faster than LAM-MPI on global reductions.

We believe the framework can be useful both for developing analytical models by providing information on factors relevant for analysis, and for tuning of an application where a fine-grained analysis can be difficult to attain.

By analyzing the performance results using different mappings, we have also exposed some bugs in the implementation of the application. Our approach can be useful both when debugging an application as well as finding configurations that offers improved performance.

References

[1] ANSHUS, O. J., AND LARSEN, T. Macroscope: The abstractions of a distributed operating system. *Norsk Informatikk Konferanse* (October 1992).

[2] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., CULLER, D. E., HELLERSTEIN, J. M., AND PATTERSON, D. A. Searching for the sorting record: Experiences in tuning NOW-Sort. *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools (SPDT 98), USA* (1998), pp. 124–133.

[3] BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. In *Proceedings of the ninth ACM Symposium on Operating Systems Principles* (1983).

[4] CARRIERO, N., AND GELERNTER, D. Linda in context. *Commun. ACM 32*, 4 (April 1989), 444–458.

[5] DICKENS, P., HEIDELBERGER, P., AND NICOL, D. Parallel direct execution simulation of message-passing parallel programs. *IEEE Transactions on Parallel and Distributed System* (1996).

[6] GOVIL, K., TEODOSIU, D., AND YONGQIANG HUANG AND, M. R. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Symposium on Operating Systems Principles (SOSP'99), published in Operating Systems Review 34(5)* (December 1999), pp 154–169.

[7] JUURLINK, B. H., AND WIJSHOFF, H. A. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems Vol. 16*, No. 3 (August 1998), pp. 271–318.

[8] KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[9] LUO, Y. Mpi performance study on the sgi origin 2000. *Pacific Rim Conference on Communications, Computers and Signal Processing* (1997), pp 269–272.

[10] REINHARDT, S., HILL, M. D., LARUS, J., LEBECK, A., J.C, LEWIS, AND WOOD, D. The wisconsin wind tunnel: Virtual prototyping of parallel computers. *Proceedings of the 1993 ACM SIGMETRICS Conference* (May 1993).

[11] ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. Using the simos machine simulator to study complex computer systems. *ACM*

Trans. On Modeling and Computer Simulation Vol. 7, No. 1 (January 1997), pp. 78–103.

[12] VENUGOPAL, S., AND NAIK, V. K. Effects of partitioning and scheduling sparse matrix factorization on communication and load balance. *Proceedings of the 1991 conference on Supercomputing* (1991), pp. 866–875.

[13] VINTER, B. *PastSet a Structured Distributed Shared Memory System*. PhD thesis, Tromsø University, 1999.

[14] VINTER, B., ANSHUS, O. J., LARSEN, T., AND BJØRNDALEN, J. M. Extending the applicability of software dsm by adding user redefinable memory semantics. *Parallel Computing (ParCo) 2001, Naples, Italy* (September 2001).

[15] WALKER, D. W. The design of a standard message-passing interface for distributed memory concurrent computers. In *Parallel Computing, Vol. 20*. April 1994, pp. 657–673.

[16] WILSON, L. F., AND NICOL, D. M. Experiments in automated load balancing. *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS '96)* (1996).