Communicating Process Architectures – 2001
Alan Chalmers, Majid Mirmehdi and Henk Muller (Eds.)
IOS Press, 2001

Using Two-, Four- and Eight-Way Multiprocessors as Cluster Components

Brian Vinter[1,2], Otto J. Anshus[2], Tore Larsen[2], John M. Bjørndalen[2]
[1] *University of Southern Denmark,* [2] *Tromsø University*

Abstract. This work considers the pros and cons of different size SMPs as nodes in clusters. We investigate the Intel SMP architecture and consider the potential of and some problems with larger node-sizes in clusters of multiprocessors. Six applications that represent different classes of parallel applications are developed in versions that support both shared and distributed memory. Performance measurements are done on three different clusters of multiprocessors, with the purpose of identifying how the number of processors in each SMP node impacts the cluster performance. Our results show that clusters using higher order SMPs do not give a clear performance benefit compared to clusters using two-way SMPs. Off the benchmark-suite of six applications, the performance of two turn out to be independent of node-size, two show an advantage of larger node-sizes, as much as 34% improvement of eight-way nodes over a dual-system, while the remaining two show an advantage of dual-processor nodes as big as 11% over an eight-way cluster.

1 Introduction

There is currently an unprecedented proliferation of low cost, mass market; small-scale shared memory multiprocessors (SMPs). Dual processor PCs are used as engineering workstations, four processor systems are applied as departmental servers, and low-cost six- to eight-way systems using PC-type processors are used for database applications. It remains to be seen if mass-market, small-scale shared-memory multiprocessors will remain commercially and technically viable. Current speculations go both ways, using both technical and commercial arguments. For research on clusters, the mix of shared memory communication within SMPs and network-based communication among SMPs raises a set of interesting questions.

In this work, we characterize the performance of clusters using two-, four- and eight-way SMP nodes respectively. Six CPU-bound applications are implemented using a combination of shared memory and Message Passing Interface, MPI. We describe how the applications are parallelized and measure their relative performance on clusters of two-, four- and eight-way SMPs respectively. The purpose is to investigate and compare performance characteristics of clusters with a the same number of processors using two-way, four-way or eight-way SMP nodes.

1.1 Clusters of Multiprocessors

Loosely coupled clusters of computers are being investigated both in academia and in industry. Currently these "poor man's supercomputers" typically uses from eight to 64 CPU's, while some employ as many as 1000 CPUs[1]. The interest in Commercial Of The Shelf, COTS, clusters as alternative multicomputers has been stimulated by several developments in addition to the obvious price advantages. Most fundamental are the recent advancements

in high-speed, low-cost interconnects, i.e. 100Mb Ethernet at the low end, Myrinet[2] and SCI-bus[3] at the high-end, and, more recently, cLan[4] and Gigabit Ethernet[5] in between.

High-performance inter-process communication mechanisms for loosely coupled clusters, mainly the standardized Message Passing Interface, MPI[6], have come of age in stable and efficient implementations. The source-code compatibility of MPI applications between architectures has made it possible to use shrink-wrap software and public domain applications on clusters, independent of the specific cluster architecture. The latter has promoted clusters heavily in fields of science outside computer science. Computer scientists have been attracted by cluster architectures for a series of reasons, including application, operating-system and distributed shared memory research.

Most research on clusters has been on platforms using a high number of uniprocessor workstations, or very few servers, typically sharing secondary storage. Shared memory multiprocessors (SMPs), are currently at price points that make them very attractive as nodes in clusters, these are often refered to as CLUMPS, Cluster of Multiprocessors[7]. Dual processor workstations cost only marginally more than similar uniprocessor systems plus the cost of the extra CPU. Each dual processor workstation is typically cheaper than two similarly equipped uniprocessors, with the same amount of memory and disk capacity. Entry-level servers using four CPUs also add other attractive features like redundant power supplies and RAID-disks. Eight-way SMPs currently carry a price premium that is explained partly by expensive vendor-specific implementations and partly by lack of competition in the marketplace.

One interesting aspect of using multiprocessor cluster nodes is to determine how the traditional multicomputer programming models perform when running on clusters of SMPs rather than on clusters of uniprocessors. Traditionally shared memory application programming interfaces (API), and distributed memory APIs has been quite different in both syntax and semantics. Shared memory APIs, such as Sys V Interprocess Communication (Sys V IPC) and APIs found in thread packages such as POSIX threads (Pthreads), are kept simple to support both flexibility and good performance for most common cases. Distributed APIs, such as MPI and Remote Procedure Calls (RPC)[8], have more complex and powerful functionalities freeing the programmer from building his own operations using simpler ones. The question remains how the two classes of programming API will work together.

From here we go on to briefly investigate the Intel SMP architecture in section 2 and the MPI API in section 3, section 4 motivate and describe the applications we have chosen to test the impact of node-size on the scalability of the application. The results from our experiments are discussed in section 5. We take a brief look at related work in section 6 and finally we draw our conclusions in section 7.

2 SMP node architecture

The Intel multiprocessor architecture is one of the most widely available multiprocessors, and most of the control hardware that is needed to implement symmetrical-multi-processing is placed on the CPU itself, with the result that a dual-processor Intel machine is only marginally more expensive than the cost of a uni-processor and the additional CPU. As one can imagine this makes these machines attractive from a cluster point-of-view since a 32 CPU cluster made up of 16 dual-processor machines is notably less expensive than a similar cluster made from 32 standard PCs. Beyond two CPUs the Intel Standard High Volume, SHV, program has made four- and eight-way SMP machines fairly cheap too. However, these are marketed only for the server segment of the market and thus never becomes truly commodity and as a consequence they don't become quite as cheap, on a per CPU basis, as the

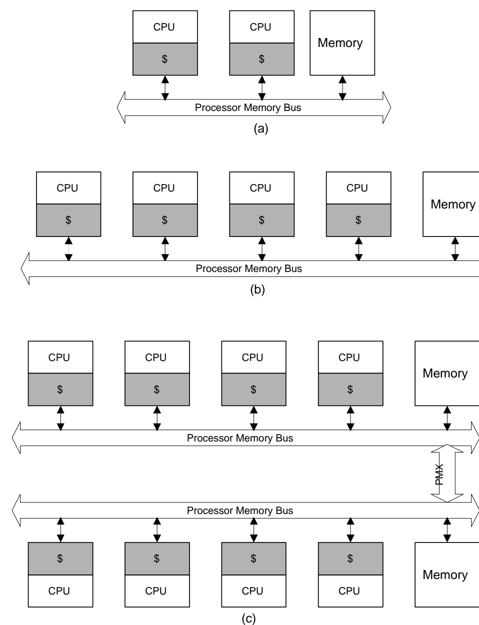

Figure 1: The Intel MP architecture in (a) two, (b) four and (c) eight CPU versions

dual-processor setup.

Intel's SMP architecture is a shared bus architecture, which is one reason why the machines are fairly cheap, but this is also the reason why the machines only scale to very few processors in each machine. Each CPU has two levels of cache and both levels are kept coherent in compliance with the MESI protocol. The Intel MP Specification as of revision 1.4[9] supports at most four CPU on the same bus. Thus to scale to eight CPUs, a system needs to use two busses and add extra hardware support to cache coherence of the two busses.

The purpose of using fewer nodes with more CPUs is to lower the average communication time between processors and limit the load on the cluster interconnect. These advantages should result in a better CPU utilization when running parallel applications on the cluster. On the downside, more processors on the same bus also means less available bandwidth per processor. Table 3 in section 5 shows the consequences of this for the three clusters used in this investigation.

If clusters of multiprocessors are intended to be 'poor-mans-supercomputers', a more serious problem is the premium cost that is associated with higher degree SMP servers. The higher cost of larger servers may be attributed to several issues; first and foremost is the lower volume in sales of these systems, which automatically increases the price per unit. In addition the higher degree servers require more hardware, larger casing, etc.

The fairly high price of four and eight way Intel based machines has resulted in a scenario where these architectures are only interesting to the segments of the market that heavily depend on the higher degree of multi-processing, e.g. the server market. Thus, four and eight CPU nodes are only available as servers with the associated high performance I/O subsystems and redundant power supplies, all of which add to the price of the machines. In addition the available four and eight way servers all seek to compensate for the limited memory bandwidth by using processors with large high-performance cache systems (Intel's Xenon class processors). The result is that it is very hard to compare the price of clusters based on different SMP sizes without ending up comparing apples to oranges. Figure 2 show the prices of a number of clusters all with 32 CPUs and 4GB memory, based on a variety of node types. While none of the two CPU per node clusters compares directly in configuration to the four and eight way versions, it is obvious that there is a significant premium associated

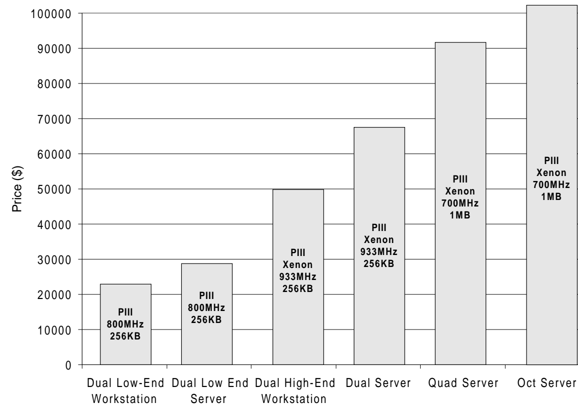

Figure 2: Price of a 32 CPU cluster using various machines as nodes

with using higher degree SMPs as cluster nodes.

It is clear that a 32 CPU cluster with 4GB RAM cluster made up of four eight-way machines is close to five times more costly than the cheapest dual-processor solution (700MHz / 1MB PIII Xenon vs. 800MHz / 256KB PIII) and 50% more expensive than the most expensive dual solution (700MHz / 1MB PIII Xenon vs. 933MHz / 256KB PIII Xenon). All the configurations in figure 2 are machines from the same brand-name producer. If the cheapest dual processor configuration were made up from home-assembled nodes, the price would be approximately 50% lower still.

The prices in figure 2 make it evident that for CLUMPS of node-size larger than two, the performance of these systems should out-perform the dual-systems in a ratio similar to the 50%-500% difference in price.

3 Message Passing Interface

The Message Passing Interface, MPI[6], is a controlled API standard for programming a wide array of parallel architectures. Though MPI was originally intended for classic distributed memory architectures, it is used on various architectures from networks of PCs via large shared memory systems, such as the SGI Origin 2000, to massive parallel architectures, such as Cray T3D and Intel paragon. The complete MPI API offers 186 operations, which makes this is a rather complex programming API. However, most MPI applications use only six to ten of the available operations.

MPI is intended for Single Program Multiple Data, SPMD programming paradigm, e.g. all nodes run the same application-code. The SPMD paradigm is efficient and easy to use for a large set of scientific applications with a regular execution pattern. Other, less regular, applications are far less suited to the MPI programming model and implementation in MPI is tedious.

MPI's point-to-point communication comes in four shapes, standard, ready, synchronous and buffered. A standard-send operation does not return until the send buffer has been copied, either to another buffer below the MPI layer or to the network interface, (NIC). The ready-send operations are not initiated until the addressed process has initiated a corresponding receive. The synchronous call sends the message, but does not return until the receiver has initiated a read of the message. The fourth model, the buffered send, copies the message to a buffer in the MPI-layer and then allows the application to continue. Each of the four models also comes in asynchronous, in MPI called non-blocking, modes. The non-blocking calls return immediately, and it is the programmer's responsibility to check that the send has com-

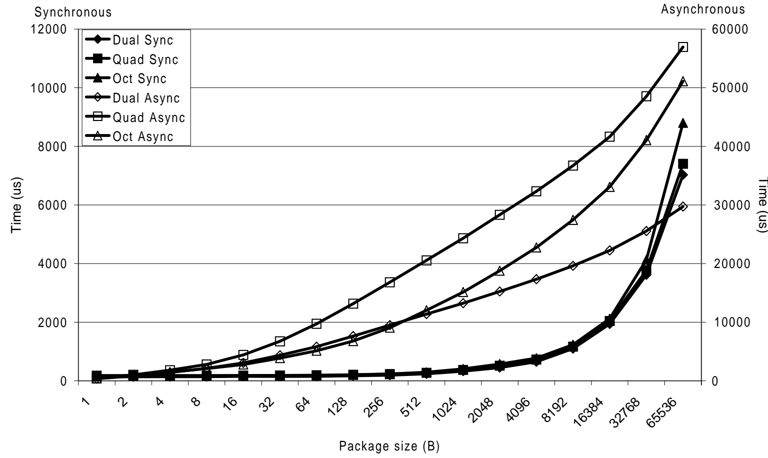

Figure 3: Point-to-point latency of synchronous and asynchronous messages in us

pleted before overwriting the buffer. Likewise a non-blocking receive exist, which returns immediately and the programmer needs to ensure that the receive operation has finished before using the data.

MPI supports both group broadcasting and global reductions. Being SPMD, all nodes have to meet at a group operation, i.e. a broadcast operation blocks until all the processes in the context have issued the broadcast operation. This is important because it turns all group-operations into synchronization points in the application. The MPI API also supports scatter-gather for easy exchange of large data-structures and virtual architecture topologies, which allow source-code compatible MPI applications to execute efficiently across different platforms.

3.1 Micro-benchmarks

Since the performance of the interconnect and the MPI layer dictates the scalability of the applications we will investigate in this work, we first identify the basic cost of synchronous (blocking) and asynchronous (non-blocking) point to point operations and of the inherently synchronous group operations, broadcast and global reductions. The MPI implementation we use here is the LAM-MPI distribution[10], version 6.5, which is commonly used in cluster-computing.

Figure 3[1] shows the latency of both synchronous and asynchronous point-to-point communication. In the applications we seek to use asynchronous messages as much as possible since these provides latency-hiding, in the cases where the application is able to do other work while the communication takes place.

It is obvious from figure 3 that the synchronous operation performance is dictated by the interconnect, while the performance of the asynchronous operations differ much more. The specifications of the clusters are listed in table 3, and it is evident that the performance of the asynchronous operations is heavily dependent on the processor speed. This is even clearer if we look at some of the raw numbers, listed in table 1. The specifications for the clusters are found in table 3 in section 5.

The broadcast latency of the dual-processor cluster with varying number of nodes and varying package size is shown in figure 4. The graph show that the latency of broadcasts depend heavily on both package size and the number of nodes. It is quite interesting that the

[1]Notice different scales for synchronous and asynchronous messages.

	Dual	Quad	Oct
1B	356	436	396
1KB	13236	24333	15136
64KB	29728	56949	51113

Table 1: Asynchronous message-latency in μs, on three different platforms

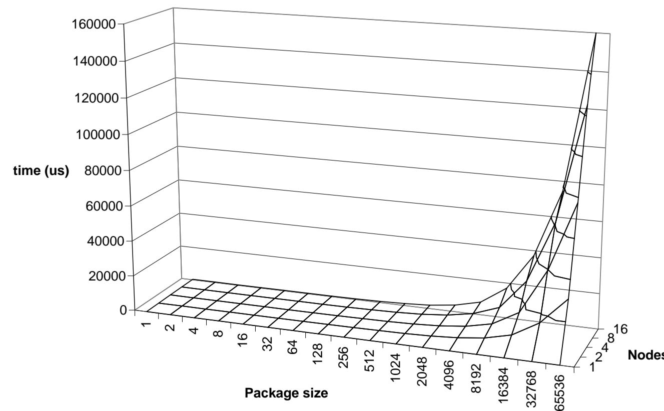

Figure 4: Broadcast latency of the dual-CPU cluster.

number of nodes has such a significant impact on performance, given that the interconnect[2] supports broadcast at the media layer.

Table 2 lists the broadcast latency for all three clusters in figure 3, and shows that while the eight-way cluster, with its faster CPUs, is faster than the four-way cluster at asynchronous operations, the four-way cluster is faster at synchronous operations. This is also visible in figure 3. Whether this is due to differences in the node architecture or due to the different 100Mb networks is unknown, though the different network-technology is the most likely reason.

Global reductions performance is very similar to the broadcast performance, though the latency is slightly higher. We chose not to present the graph and numbers here since the pattern is almost identical to that of the broadcast benchmark.

4 Applications

All the applications we present here are chosen to represent different classes of processor-bound hard-to-compute problems. The problems are chosen because they are well known and thus should allow the reader to focus on the performance characteristics that we observe, similar or close to identical performance behavior should be observed on other, more realistic, applications in the same classes.

The included applications are, an embarrassingly parallel Monte Carlo simulation; a global optimization problem, the Traveling Salesman Problem; a grid application, Successive Over Relaxation; a simple linear algebra problem, matrix multiplication; an automaton model, WATOR; and finally the classic super-computer benchmark, Gaussian elimination.

[2]Switched Ethernet

Nodes	*Dual*			*Quad*			*Oct*		
	1	**1k**	**64k**	**1**	**1k**	**64k**	**1**	**1k**	**64k**
1	0,5	0,5	0,5	1,2	1,2	1,2	1,0	1,0	1,0
2	151	676	30497	205	798	40987	202	835	44551
4	213	988	77504	280	1202	97817	260	1429	102916
8	302	1589	125511	546	1767	156712			
16	377	2028	159971						

Table 2: Broadcast-latency in μs with varying nodesize, number of nodes and packagesize

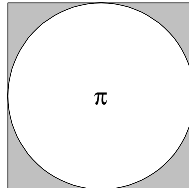

Figure 5: Monte Carlo Estimation of π

4.1 Monte Carlo Estimate of π

Monte Carlo methods are a widely used group of numerical methods, which involve sampling from random numbers. The Monte Carlo method can be used to solve otherwise intractable problems. Since Monte Carlo applications are based on random events a large number of such events must typically be processed to ensure a realistic result. The Monte Carlo method we use here, Monte Carlo π, is utterly uninteresting in and of itself but exhibits the same behavior as real world Monte Carlo applications as well as the related Las Vegas methods and random walks, all of which are frequently used in physics, biology and finance.

The Monte Carlo method of estimating π uses a unitary circle, inscribed inside a square. The application repeatedly picks a random point within the square, the fraction of the points that are inside the unitary circle then represents $\pi/4$, this is sketched in figure 5.

The Monte Carlo approach depends heavily on a good random number generator and a large number of guesses, still it will never provide a particular good estimate of π. The sequential solution picks all the numbers and estimates π using a single processor. The parallel solution distributes the required samples amongst the nodes. Each node spawns one thread per processor and each thread test a fraction of the tests that the node is responsible for.

The Monte Carlo estimation of π is embarrassingly parallel, and is included as a sanity-check, if we do not get perfect speedup on this application there is an error somewhere[3]. Since the problem is embarrassingly parallel it only include one protected function call amongst the threads in a single node to collect the total circle hits on the node and a single synchronous MPI operation to collect the global number of circle hits.

4.2 Traveling Salesman Problem

The Traveling Salesman Problem, TSP, is a classic representative for the class of global optimization problems. The TSP solution we use in this work is a depth-first branch-and-bound algorithm which makes the parallel version different from the other applications we use by the fact that a static division of the work would result in a highly unbalanced execution.

[3]The Monte Carlo π test actually detected various problem during the testing

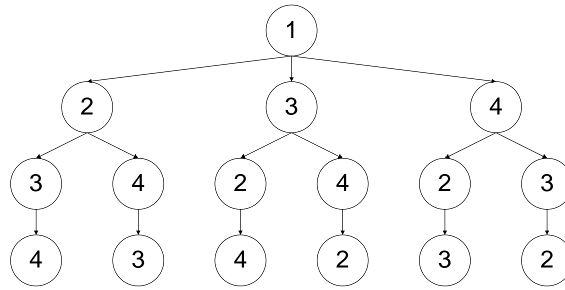

Figure 6: A complete TSP search-tree for a four city problem

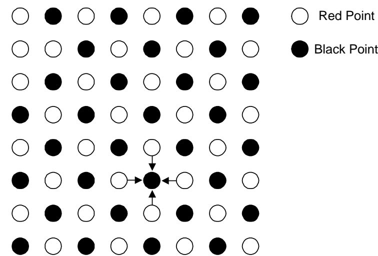

Figure 7: Red-Black Successive Over-Relaxation

Thus the parallel TSP is implemented as a bag-of-tasks application, a paradigm that does not come natural to the SPMD programming paradigm that MPI is designed around.

The parallel TSP is implemented as a global master process and set of worker-threads on each node, each thread communicates with the master to retrieve jobs and submit results. A job is represented as a set of cities that have already been placed and a set that need to be placed, i.e. a sub-tree. Each job that is sent from the master has the length of the shortest known route piggy-backed and each node maintains one shared instance of the bound value. Since the application is so highly unbalanced in the workload this application use synchronous messages for communication. A scheme for applying asynchronous messages could be developed, but two things talk against it, first of all an asynchronous scheme would place an extra job at each node, which is likely to increase the load-unbalance. The second reason why asynchronous messages are unfavorable is that the messages a quite small and the additional overhead of asynchronous messages, as show in figure 3, are very likely to be bigger that the potential gain from asynchronicity.

4.3 SOR

Successive Over-Relaxation, SOR, is a frequently used technique for solving very large systems of partial differential equations by successive approximations. The general idea is to approximate each element in a matrix to its neighbors until the sum of all changes within one iteration converges below a given value.

The Red-Black checker pointing version of SOR, shown in figure 7, returns identical results for the same system of equations; independent of the actual computing environment, while at the same time providing sufficient parallelism that real speedup can be achieved. With Red-Black checker pointing, the equation system is divided into alternating red and black points in a chess-board fashion. Updating a red point depends only on black neighboring points and vice versa. Using this, an algorithm is derived where each worker-process updates all its red points and then exchanges red border point values with its neighbors. Each worker then updates its black points and repeats the communication for the black points. At the end of each iteration the global change in the system is calculated and the process

```
do {
  if THREAD_ID=0 then {
    ASYNC_SEND(top and bottom black rows)
    ASYNC_RECV(top and bottom neighbor black rows)
  }
  UPDATE(red points)
  if THREAD_ID=0 then {
    ASYNC_SEND_FINISH
    ASYNC_RECV_FINISH
    UPDATE(top and bottom red points)
  }
  thread_barrier() //All threads must have finished red-update
  if THREAD_ID=0 then {
    ASYNC_SEND(top and bottom red rows)
    ASYNC_RECV(top and bottom neighbor red rows)
  }
  UPDATE(black points)
  if THREAD_ID=0 then {
    ASYNC_SEND_FINISH
    ASYNC_RECV_FINISH
    UPDATE(top and bottom black points)
  }
  thread_sum(change) //Collect the change from all threads
  if THREAD_ID=0 then MPI_SUM(change) //Find the global change
  thread_update(change) //Tell all threads the global change
} while ( change>epsilon )
```

Figure 8: CLUMPS version of Red-Black Successive Over-Relaxation

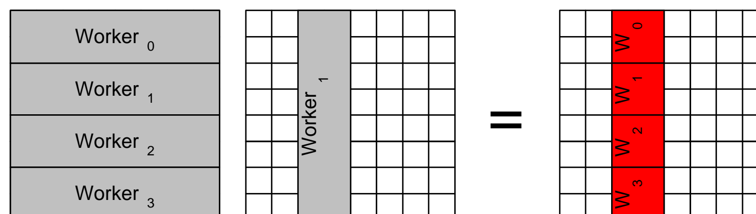

Figure 9: Matrix tiling

continues until the change in the system is below a given threshold.

The CLUMPS version divides the system into a set of static blocks which are divided amongst the nodes, each node subsequently divides its dataset between the CPUs on the node. Since LAM-MPI is not thread-safe one thread is responsible for communication and updating the upper-most and lower-most data-rows in the dataset, in addition to a portion of the dataset similar to the other threads, this thread also represents the node in the global reduction to find the total change in the system, see pseudo-code in figure 8. An iteration then includes two asynchronous send operations and two asynchronous receives as well as one synchronous global reduction operation.

4.4 Matrix Multiplication

Matrix multiplication is frequently used in scientific applications. Although several concurrent and parallel algorithms exist that requires extensive communication during the calculation, an alternative approach is to coarsely distribute the matrixes amongst the nodes and broadcast one matrix to all nodes one block at a time as shown figure 9. This approach is

```
in parallel C = 0
for i = 0;N-1{
  broadcast A*i within rows
  broadcast Bi* within columns
  in parallel C = C + (A*i)(Bi*)
}
```

Figure 10: PBLAS Matrix multiplication

similar to the one used in PBLAS [11]. The generic solution in [11] is based on the following matrix decomposition:

$$
\begin{aligned}
C = & \begin{pmatrix} A_{00} \\ A_{10} \\ \vdots \\ A_{(N-1)0} \end{pmatrix} \left(B_{00}B_{01}\dots B_{0(N-1)}\right) \\
+ & \begin{pmatrix} A_{01} \\ A_{11} \\ \vdots \\ A_{(N-1)1} \end{pmatrix} \left(B_{10}B_{11}\dots B_{1(N-1)}\right) \\
+ & \dots \\
+ & \begin{pmatrix} A_{0(N-1)} \\ A_{1(N-1)} \\ \vdots \\ A_{(N-1)(N-1)} \end{pmatrix} \left(B_{(N-1)0}B_{(N-1)1}\dots B_{(N-1)(N-1)}\right)
\end{aligned}
\tag{1}
$$

We have chosen a less generic, but more efficient algorithm where we maintain both A and B in distributed state and thus only need to broadcast one matrix amongst the other processes. In addition the matrix is not broadcast column by column, but the entire block at once. Each node uses an efficient sub-blocked matrix-multiplication to take advantage of cache-memory. The MPI solution results in one synchronous broadcast per node in the overall execution. The broadcasts are rather large however and will stress the MPI layer significantly.

4.5 WATOR

The WAter TORus world, WATOR, is a classic discrete event simulation[12], and while it provides valuable information in itself we chose to introduce it here for reasons similar to the Monte Carlo π example, namely that it is simple and easy to understand, while still being typical for the class of applications that it represents. Discrete event simulations are widely used to model everything from digital systems to financial forecasts, logistics and traffic simulations.

WATOR is the simulation of a very special world, first of all the planet is not a sphere as most planets we know, rather the planet is shaped as a doughnut, or a torus, which greatly simplifies mapping the world into discrete blocks. As the whole surface of WATOR is covered with water there are only two types of life, which we are interested in, fish and sharks.

Fish are simple organisms that move around at random, and at some point when a fish comes of age is will have two children and die itself. A fish can move into any of its neighboring eight squares, given that the square is empty, if all eight neighboring squares are occupied

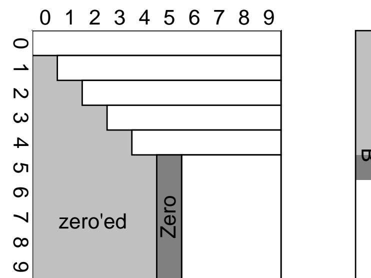

Figure 11: Gaussian Elimination

the fish remains in place. At any discrete time step a fish becomes one time-unit older, and once it has come of age it will split into two fish, one of which will go to a neighbor cell while the other stays in the cell where it was born, both new fish is age zero.

Sharks are similar simple creatures, however sharks also need to eat fish in order to survive. At each time-step a shark moves to a random neighboring cell which holds a fish, if there are no fish which neighbors the shark it moves to a random neighboring cell and increases its hunger index, if the hunger index reaches a starvation limit the shark dies, when the shark eats it resets the hunger index to zero. Similar to fish, sharks will at some point get old enough to breed and once this age is reached the shark is replaces by two new sharks, both with age and hunger index zero.

Thus the simulation of one time-step consists of two steps, first all fish are moved, then the sharks, each step issues four asynchronous MPI operations, two send and tworeceive. The randomness of the application allows us to ignore further synchronization issues.

4.6 Gaussian Elimination

Gaussian elimination seek to solve large systems of linear equations, by performing a LU (Lower Upper) restructuring of the system matrix and back-substitution of the parameters that were used for achieving the LU version of the system-matrix. Figure 11 show how each line is transformed into its LU shape by zeroing the left-most entry and dividing all lower rows by the row that was finished. There is a twist however, since the limited numerical representation in the processor can destroy the data if the divisor is close to zero. Because of this we need to perform a partial pivot on the remaining rows for each iteration such that the largest possible divisor is used.

To perform the LU decomposition in parallel we basically have two choices, either divide the matrix by rows or by columns. If we divide the system by columns then one processor alone can decide the pivot row and broadcast the pivot to all processors. After this an all-to-all communication phase is needed to create a copy of the active row at all nodes. If the matrix is distributed row-wise then an actual election of the best pivot value is needed. After the election the process that won the election broadcasts its row.

We chose the row-wise solution in order to reduce the large messages to a one-to-all broadcast. Including the partial pivot this means that each iteration use two synchronous group operations, one election of the best divisor and a broadcast of the 'winning' row.

5 Performance

To compare the performance of 2, 4, and 8 way SMPs, we have run the application from section 4 on three different clusters that all have a total of 32 CPUs, distributed over six-

	Dual	**Quad**	**Oct**
CPUs per node	2	4	8
Bus width	8B	8B	8B
Mem. Busses	1	1	2
CPU type	PIII	PPro	PPro
2' level cache	512KB	512KB	1MB
CPU speed	450MHz	166MHz	200MHz
Memory per node	256MB	128MB	2048MB
Bus Speed	100MHz	66MHz	66MHz
Eff. Bandwidth	0.89B/cycle	0.80B/cycle	See figure 12
Interconnect	Fast Ether	Fast Ether	100VG

Table 3: Architectural specifications of the three clusters used for the experiments

teen, eight and four nodes respectively. 100Mb networks interconnect the nodes in all three clusters.

Since our primary concern in this work is scalability we present performance as speedup, which will be presented as CPU utilization. The use of speedup as a measure of success is heavily debated but while the information that can be extracted from speedup numbers may be limited, it fits our purpose well. First of all, since the cluster-nodes differ in more ways than simply SMP-size other measures than relative scalability are unachievable. Secondly, since scalability is the primary topic of investigation in our work, achieved speedup is the parameter we are interested in.

5.1 Experiment Design

All experiments reported on in this paper were done using three clusters. The first cluster, *Dual*, is comprised of sixteen "no name" PCs, all equipped with two 450MHz Pentium III processors and the Intel BX chipset. Each PC has 256MB main memory, and a single 33MHz, 32-bit PCI bus. The second level cache-size is 512KB per processor.

The second cluster, *Quad*, is comprised of four HP LX-Pro Net-servers, each with four 166MHz Pentium Pro processors. Each server has 128MB main memory, and dual peer 33MHz, 32 bit PCI buses. The level 2 cache size is 512KB per processor. These machines belong to the first generation of Intel's SHV machines.

The third cluster, *Oct*, is comprised of four HP Netserver LXr Pro 8 servers, that all have eight 200MHz Pentium Pro processors. Each server has 2GB RAM and dual peer 33MHz, 32 bit PCI buses. The 2. level cache size is 1MB.

The *Dual* and *Quad* clusters used the Trendnet TE100-PCIA (DEC[4] Tulip 21143 chip set) 100 Mb/s network interface cards (NIC) connected to a switch. The network interface cards (NIC) were on PCI bus no. 0 on the *Quad* cluster. The *Oct* cluster is interconnected with a HP 100VG 100Mb/s network, also on PCI bus no. 0. The 100VG network is connected via a 100VG hub. The complete specifications for all three clusters are summarized in table 3.

The operating system on all clusters is Linux v. 2.2.14, and LAM-MPI 6.5 is used as the MPI layer. All the applications were compiled with gcc 2.96.2 and optimization-flag '-O3'. Time is measured with the Linux `gettimeofday` system call.

In order to test the performance under varying problem sizes, the applications have been run with three different datasets; tiny, medium and large. The datasets corresponds to 10, 100

[4]Now Intel, but most literature references work with the original DEC chipset

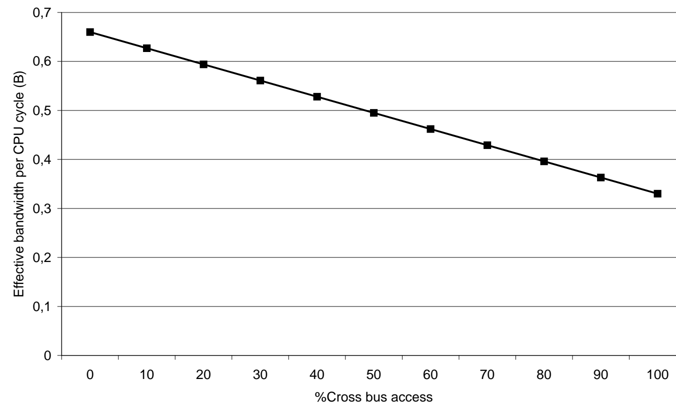

Figure 12: Effective available bandwidth per CPU per cycle in an eight CPU machine as a function of cross-bus accesses

	MC-π	**Mat. Mul.**	**SOR**	**TSP**	**WATOR**	**Gauss**
Tiny	25E6 darts	640x640	1000	14 cities	250x250, 100 itt	900
Medium	25E7 darts	1536x1536	3000	16 cities	800x800, 100 itt	1900
Large	25E8 darts	3072x3072	5000	17 cities	2600x2600, 100 itt	4000

Table 4: Applications and problem sizes used to measure the performance of 2-4-8 way SMPs

and 1000[5] seconds sequential runtime on the fastest CPUs, the 450MHz PIII processors in the dual-CPU cluster. Table 4 sums up the applications and the dataset parameters that have been chosen for each.

5.2 Monte Carlo π

The performance of the Monte Carlo application in figure 13 does not provide much information to us. All systems achieve close to perfect speedup even with the tiny problem-size. This was expected and the result validates that all the processors do in fact provide identical performance. The conclusion one may draw from the experiment is that if an application is embarrassingly parallel, there is no reason to invest in expensive SMP hardware; this conclusion is hardly surprising.

In order to achieve the expected result however, we had to replace the `libc` random function by a custom pseudo-random generator, since the use of the library version eliminated all parallelism within one node. In fact even a single threaded version is significantly slower than a standard sequential version, due to mutual-exclusion protection of the random function.

5.3 TSP

The TSP application was rather hard to fit into the timing parameters and required manual layout of the sequence of cities in the list to fit the desired runtime. While this dictates the execution time of the application it does not influence the parallelism in the execution.

Since TSP is both threaded and tasked, there are two features in the implementation that work against each other; on one hand more CPUs mean that the bound variable on any node is updated faster and thus reduces the work that need to be executed on that node. However,

[5]The large problem for SOR is not really 1000 seconds, but the largest we can do in 128MB

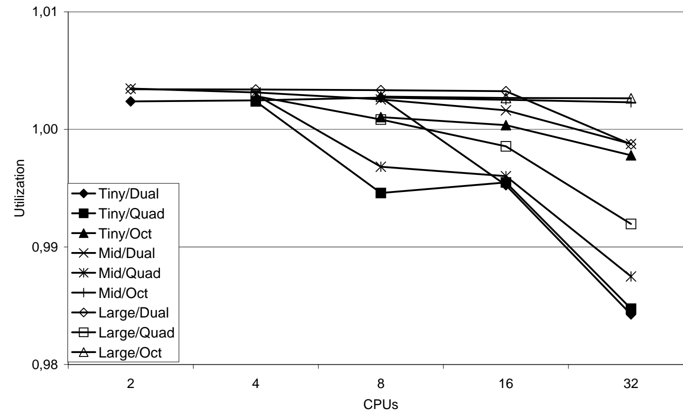

Figure 13: Performance of the Embarrassingly Parallel Application Monte Carlo π

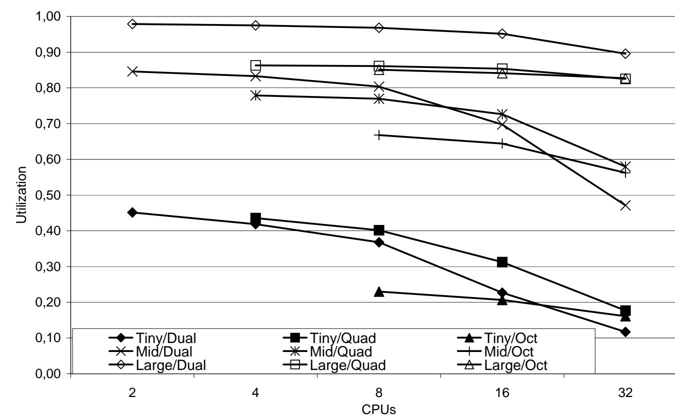

Figure 14: Performance of the Traveling Salesman Problem

more CPUs also increase the probability for contention on access to the MPI layer. The impact of the two elements is clearly seen in figure 14. The tiny portion benefits from faster updates of the bound variable, e.g. more CPUs per node, while the larger problems benefit more from lower contention on the network interface and favor the dual-processor cluster over the four- and eight-way systems.

5.4 SOR

The four-way servers are quite old and as a result have very little memory, which limits the size of the SOR application to a maximum of 5000x5000, floats in the large dataset. This results in a sequential execution time of only 250 seconds on the dual nodes, far from the target of 1000 seconds. Thus, this is not really a huge data-set per se.

SOR performance is quite similar on all three architectures as shown in figure 15. The dual-processor cluster has similar performance to the others up until 8 nodes while at 16 nodes it is 15% slower with the tiny test-set and 5% slower on the largest problem. The difference in effective bandwidth does not shine through because SOR is so regular that out-of-order execution, memory perfecting and cache locality is quite effective at hiding the memory latency. As figure 4 show the number of nodes is a significant parameter in the time a global reduction takes and this becomes obvious as the dual cluster falls behind the other two. The cross bus exchange of data in the eight-way nodes is visible too. The result is that the eight-way cluster is slightly slower than the four-way although the measurements in table

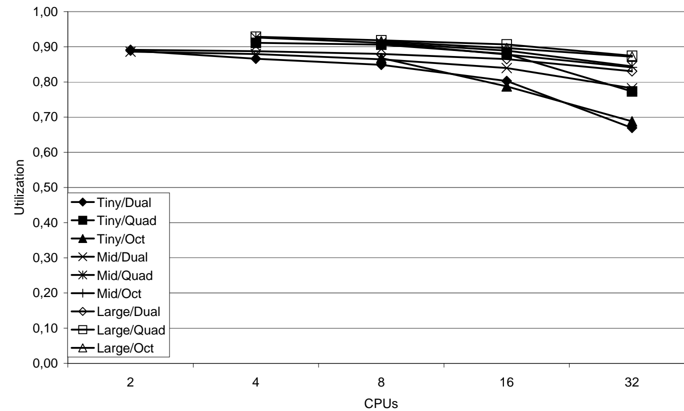

Figure 15: Performance of Successive Over Relaxation

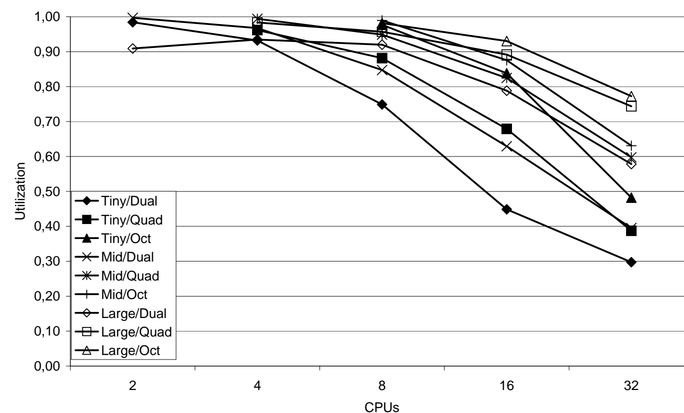

Figure 16: Performance of Matrix Multiplication

2 would indicate that it should be vice versa.

5.5 Matrix Multiplication

Since the matrix-multiplication algorithm we use is based on large broadcasts, it is not surprising that we see a clear advantage of larger node-sizes, that is, fewer nodes in the system. The pattern is clear over all problem-sizes and numbers of CPUs in the cluster. The result is that using 32 CPUs on the huge problem-set, as shown in figure 16, the eight-way CLUMPS is 34% faster than the two-way system.

The advantage of eight-way nodes over four-way nodes is not as large as table 2 would indicate. This is because once the data-block is received on the node, it still need to be communicated across the memory-bus connection, in effect making the cross bus rate close to 50%, which corresponds to an effective bandwidth of 0.5B/cycle. The success of the cache-efficient block-multiplication algorithm is evident in that the higher effective bandwidth of the two-way system does not outweigh the higher cost of broadcasting with more nodes.

5.6 WATOR

The WATOR application is the application that provides the highest level of asynchrony, since this application only uses non-blocking communication. The consequence of the asynchrony is that with the medium and large datasets the communication cost can be hidden quite well

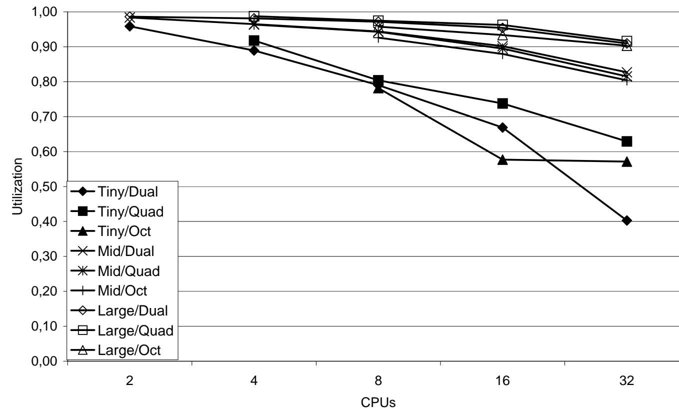

Figure 17: Performance of the WAter TORus simulation

and we achieve close to linear speedup and similar performance for all three systems. The small problem-set is not enough to hide the latency when we use more than a few nodes.

The performance of WATOR on our CLUMPS in figure 17 includes an interesting point on the choice of nodes size, since the four-way system consistently outperforms the eight-way CLUMP. While the difference is small, ranging from 10% on the small data-set to only 1% on the large set, it is consistent and except on the large data-set, it is also statistically significant. This behavior is the result of data-pushing vs. -pulling. Between nodes data is exchanged via MPI, which is slow but executes asynchronously, i.e. when the thread needs the data, it is likely to be ready. Within nodes the exchange scenario is vice versa; here the threads wait for the neighboring thread to finish, and only after the data is ready can the receiver start reading the data. In the eight-way system this means crossing the inter bus connection for two of the processors. While the waiting period is likely to be zero,[6] the data transfer still cannot begin before the data is needed.

A similar pattern is also seen in the SOR application, which is similar to the WATOR access pattern. However the global reduction in the SOR code favors the eight-way system and returns in less clear results.

5.7 Gausssian Elimination

Overall the Gaussian elimination provides much worse speedup than the other applications. This is no surprise since the parallel version is based on two synchronous MPI operations per iteration. It is interesting to see though that with 32 CPUs and solving the largest system, the higher available bandwidth provides better performance for the dual-processor based CLUMPS, which is 11% faster than the eight-way system and 33% faster then the four-way. Since there is little cross-bus communication the eight-way system does not suffer from lowered bandwidth and instead benefits from fewer nodes when performing the rather costly synchronous operations.

The implementation of the Gaussian elimination is simple and the performance could be improved by adding an extra thread, which performs the MPI operations while the other threads perform the calculations. This kind of hand-coded asynchrony would only increase the advantage of the two-processor CLUMPS since this is the architecture that suffers the most under the high cost of group operations.

[6]Almost certainly in fact, however the race-condition is handled in the code

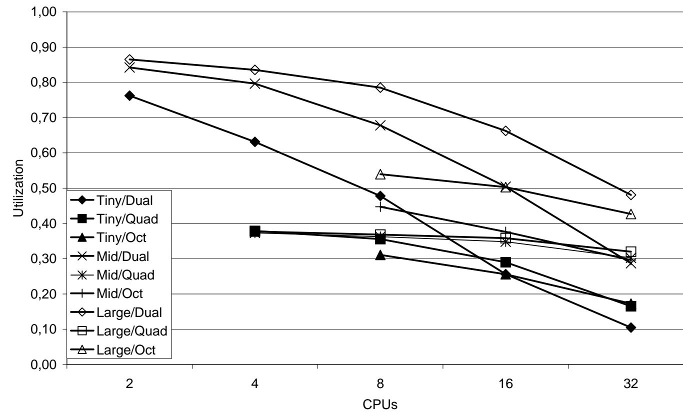

Figure 18: Performance of Gaussian Elimination

	Favors small nodes	Favors large nodes	Indifferent
MC-π			X
Mat. Mul.		X	
SOR		X	
TSP	X		
WATOR			X
Gauss	X		

Table 5: Summary of the applications and their favored node-size with 32 CPUs and the large data-set

5.8 Performance Summary

There is no one truth as to whether CLUMPS should use small or large nodes, as table 5 shows. It is clear, however, that there are applications that clearly favor larger nodes, such as the chosen matrix multiplication algorithm, and some that favor smaller nodes, most significantly the Gaussian elimination. Since the matrix multiplication is the strongest argument for large nodes, and given that the chosen algorithm is quite naive, the arguments for large nodes are quite weak.

6 Related Work

CLUMP architectures have been around for a while, but few in-depth investigations have been published.

In [13] Takahashi et al. design and implement a special CLUMP version of MPI, MPICH-PM/CLUMP. The MPI version uses Myrinet [2] and implements a full zero-copy protocol. The work compares the performance of the NAS benchmark suite on a uni-processor and a dual-processor cluster and concludes that the CLUMP machine is as much as 30% slower than, and never better than, the uni-processor version. Since the goal of this work is to build an efficient SMP MPI implementation, the dual-cluster is treated as uni-processors, e.g. the applications them-selves are not threaded.

Another angle is found in [14], where the authors investigate the performance of sparse matrix systems on shared memory, distributed memory and hybrid architectures. Here the authors find a small advantage of mixing shared memory and message passing programming.

More recently [15] compares Discrete Element Modelling code on a CLUMP using MPI

and OpenMP and concludes that MPI is favorable to OpenMP on a CLUMPS, but not on a single SMP node.

7 Conclusions

Comparing CLUMPS architectures is not straightforward, since we cannot compare systems that differ only in the number of CPUs per node. However, it is still evident that increased node-size comes at a price, and that this price is closer to 500% than to 50% extra cost per CPU, when comparing dual and eight-way nodes.

Initial micro-benchmarks show a huge difference between synchronous and asynchronous point-to-point operations. More surprisingly, a significant dependence on the number of nodes in the system is seen in group-operations such as broadcasts. This is interesting given that the nodes are interconnected by a media that supports broadcasts.

Our experiments show that the benefit of higher order nodes depends heavily on the application nature; embarrassingly parallel applications like the Monte Carlo π and highly asynchronous applications like WATOR shows no performance difference at all, while applications that depend on synchronous MPI operations such as SOR and Matrix Multiplication show an advantage of more CPUs per node, i.e. fewer nodes in the system. A global optimization problem, TSP, shows an expected advantage of larger node-size with small problems since these are not bandwidth dependent and more threads may lower the total work that has been performed. With large problem-sets the dual-processor system outperforms the four- and eight-way systems because of the lower contention on the network interface. The most significant advantage of smaller node-size is in applications that require more processor-memory bandwidth, such as Gaussian elimination, where we see as much as a 33% advantage of two-way nodes over four-way, and 11% over the eight-way system, with large problems. This is in spite of the fact that Gaussian elimination is based purely on synchronous MPI operations. Small equation-systems show an advantage of fewer nodes, with a 65% performance advantage of the eight-way system over the cluster based on dual-processors.

If we focus on the large problem sets, which after all are the main target for high-performance clusters, the results show two applications that favor large nodes, two that favors small nodes and two that are neutral towards node-size. Considering the price-premium on larger CLUMPS the straightforward conclusion is that larger nodes are not worthwhile, though there are advantages to be had on certain applications and for small problem-sets.

Acknowledgements

The authors would like to thank Dr. Joan Boyar for valuable input for the final version of this paper.

References

[1] Genetic-Programming.com. 1,000-pentium beowulf-style cluster computer for genetic programming. *http://www.genetic-programming.com/machine1000.html*, 1999.

[2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–38, Feb 1995.

[3] K. Alnaes, E.H. Kristiansen, D.B. Gustavson, and D.V. James. Scalable coherent interface. In *CompEuro 90*, 1990.

[4] Giganet. Giganet clan.

[5] G. Ethernliance. Gigabit ethernet: Accelerating the standard for speed, 1999.

[6] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, March 1994.

[7] Steven S. Lumetta, Alan M. Mainwaring, and David E. Culler. Multi-protocol active messages on a cluster of smp's. In *Proceedings of the 1997 ACM/IEEE SC97 Conference*, pages 15–21, San Jose California, USA., November 1997. ACM Press and IEEE.

[8] A. D. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, pages 39–59, Feb 1984.

[9] Multiprocessor specification version 1.4. Technical report, Intel Corp, 1997.

[10] G. Burns, R. Daoud, , and J. Vaigl. LAM: An Open Cluster Environment for MPI. www.lam-mpi.org, 1994.

[11] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of blas: General techniques for level 3 blas. Technical report, The University of Texas at Austin Austin, Texas 78712, 1995.

[12] A.K. Dewdney. Computer recreations. *Scientific American*, 250:22–34, 1984.

[13] Toshiyuki Takahashi, Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Horoshi Harada, Yutaka Ishikawa, and Peter H. Beckman. Implementation and evaluation of MPI on an SMP cluster. In *IPPS/SPDP Workshops*, pages 1178–1192, 1999.

[14] D. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems, 1997.

[15] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing*, 2000.