Combining Shared and Distributed Memory Programming Models on Clusters of Symmetric Multiprocessors: Some Basic Promising Experiments

L. Giraud*

July 2001
Revised with Alinka Itanium experiments - September 2001

CERFACS Tech. Rep.: WN/PA/01/19

Abstract

This note presents some experiments on different clusters of SMPs, where both distributed and shared memory parallel programming paradigms can be naturally combined. Although the platforms exhibit the same macroscopic memory organization, it appears that their individual overall performance is closely dependent on the ability of their hardware to efficiently exploit the local shared memory within the nodes. In that context, cache blocking strategy appear to be very important not only to get good performance out of each individual processor but mainly good performance out of the overall computing node since sharing memory locally might become a severe bottleneck. On a simple benchmark, representative of many large simulation codes, we show through numerical experiments that mixing the two programming models enables to get attractive speed-ups that compete with a pure distributed memory approach. This open promising perspectives for smoothly moving large insdustrial codes developed on distributed vector computers with moderate number of processors on these emerging platforms for intensive scientific computing that are the clusters of SMPs.
Keywords: Cluster of Pentium PCs, cluster Alinka Itanium, Compaq Alphaserver, shared memory, distributed memory, OpenMP, MPI, performance evaluation.

1 Introduction

In the recent years, new parallel computer architectures have appeared that combined disjoint memory address space between groups of processors and a global memory address space within each group of processors. This kind of computer architecture, promoted by the US-ASCI project, is usually called "Cluster of SMPs (Symmetric Multi-Processors)". This physical memory organization perfectly matches the requirements of parallel algorithms that can exploit two levels of parallelism. The outer/coarser is implemented between the SMPs and the inner/finer within each SMP. In that respect the parallel programming paradigms are message passing at the coarser level and loop level parallelism at the finer.

In this note we intend to investigate such programming combination through a very simple numerical algorithm, namely the explicit solution of the heat equation in a square.

*CERFACS, 42 av. Gaspard Coriolis, 31057 Toulouse Cedex, France.

For this purpose we consider different computing platforms that have such macroscopic memory organization. Those platforms are a cluster of bi-processors Pentium, a cluster of bi-processor Alinka Itanium and a Compaq Alphaserver cluster.

Through numerical experiments we show that mixing the two programming models enable to get attractive speed-ups that compete with a pure distributed memory approach. In addition these experiments reveal that the key to get good overall performance resides in a good exploitation of the local shared memory within the node.

2 Architecture overview of the target clusters

The experiments described in this note have been performed on three different clusters of SMPs namely

1. A cluster of eight bi-processors Pentium PC running Linux.
   The characteristics of the nodes on PC cluster are the following:

   - *Processor:* 2 Intel Pentium III, 933 MHz, 16 KB L1-cache, 256 KB L2-cache, peak performance 933 MFlops.
   - *Memory:* 1 GB of RAM shared by two processors. The processors share the memory access through one 64-bit 133 MHz memory path giving a bandwidth of around 1 GB/s.
   - *Network:* Myrinet network using DMA through a PCI card, latency 9 μsec, 250 MB/s peak bandwidth each way in full-duplex,
   - *Compiler:* Portland Group Fortran 90.

2. A cluster of two bi-processors Alinka Itanium PC running Linux.
   The characteristics of the nodes on PC cluster are the following:

   - *Processor:* 2 Itanium, 733 MHz, 16 KB L1-cache, 96 KB L2-cache, 2 MB L3-cache, peak performance 2.93 GFlops.
   - *Memory:* 1 GB of RAM shared by two processors. The interleaved memory is accessed through a sophisticated memory path enabling a transfer rate of 4.27 GB/s.
   - *Network:* 100 Mbit Ethernet,
   - *Compiler:* Intel or SGI Fortran 90 compiler.

3. A cluster of four ES40 Alphaserver Compaq.
   The characteristics of nodes on the Compaq computer are as follows:

   - *Processor:* 4 Alpha EV 6.7, 667 MHz, 64 KB 2-way associative L1-cache, 8 MB direct mapped L2-cache, peak performance 1.33 GFlops.
   - *Memory:* 4 GB of RAM shared by four processors through two 256-bit 88 MHz memory path giving a peak memory bandwidth of 5.2 GB/s.
   - *Network:* Fat tree Quadrics network with Elan card, 3 μsec latency and 200 MB/s peak bandwidth each way in full-duplex,
   - *Compiler:* Compaq Fortran 90.

If we compute

$$\sigma = \frac{\text{Peak processor (MFlop/s)}}{\text{Peak Memory bandwidth (Mw/s)}}$$

where the memory bandwidth is expressed in Mwords per second with a word being 64 bits, we obtain the figures displayed in Table 1. It can be seen that this ratio on the Pentium based platform is the lowest and that the Compaq is the computer that exhibits the best balance between memory bandwidth and processor speed.

Cluster	Peak processor (MFlop/s)	Peak Mem. Bandwidth (Mw/s)	σ processor	σ node
Pentium	933	128	0.137	0.069
Alinka Itanium	2930	546	0.187	0.094
Alphaserver Compaq	1330	665	0.5	0.125

Table 1: Ratio MFlops rate versus memory bandwidth per processor and per node on the three clusters.

The compilers used in our experiments support the OpenMP [2, 7] directives. For the experiments the codes were generated using the best found optimization options. On those platforms the message passing library is based on MPI-CH [5] optimized for the interconnecting network. We refer to the Appendix A for some additional experiments to measure the effective communication throughput using MPI on the Compaq-Quadrics and the Myrinet network.

3 The benchmark code

In order to investigate the parallel behaviour of these computing platforms we consider the numerical solution of the heat equation in two dimension using an explicit scheme. The main advantage of this algorithm is that its parallelization is straightforward and enables to easily mix the shared and distributed programming paradigms. Let us quickly present the algorithm and its parallelization.

The heat equation is usually written:

$$\left\{ \begin{array}{rcll} \dfrac{\partial v}{\partial t} - \Delta v & = & F(x,y) & \text{in} \quad \Omega, \\ v & = & 0 & \text{on} \quad \partial\Omega, \\ v(x,0) & = & v_0(x). & \end{array} \right.$$

This equation is discretized in time using an explicit scheme with time step Δt and in space by linear finite elements with mesh size h giving rise to the stiffness matrix $h^{-2}A$ for the discretized Laplace operator. With $u^m = v(x, t^m)$ at each time step we compute

$$u^{m+1} = u^m + \frac{\Delta t}{h^2} A u^m$$

that basically reduces to a sparse matrix multiplication per time step. Such algorithm is straightforward to parallelize but the policy to select depends on the architecture of the target parallel computer.

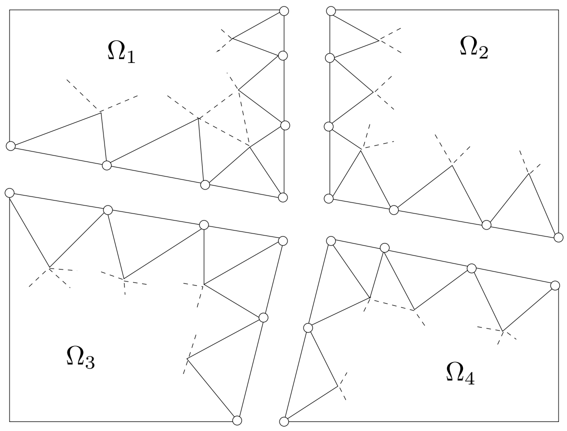


Figure 1: An example of mesh partitioning.

On shared memory platforms (also referred to as symmetric multiprocessor) a loop level parallelism is often efficiently implemented. On those platforms, OpenMP is the tool of choice for such a fine grain multi-threaded parallelization that can be implemented thanks to a few parallel directives inserted in the original sequential code. On distributed memory computers, a classical approach to implement the solution of partial differential equations is to split the mesh into subdomains, assign each subdomain to a different computing node and use message passing to exchange information along the artificial interfaces introduced by the mesh splitting. Such an approach enables to express a coarse grain parallelism suitable for distributed memory computers. An example of mesh splitting is given in Figure 1, where four subdomains are considered. With this approach the parallel code is very similar to the sequential one; the only two differences consist first in the message exchanges required along the interfaces to assemble the contribution of the subdomains to the matrix vector product at the shared nodes, second the reduction required to compute the 2-norm between two successive iterates that is the stopping criterion to detect the steady state. Due to the homogeneous nature of the computing nodes of a cluster, MPI [6] is the message passing library of choice.

On a cluster of SMPs, the two parallelization strategies can be combined. That is one subdomain is assigned to each node of the cluster, exchanges between the nodes are performed using message passing through the interconnection network. In addition, the fine grain parallelism is exploited within each subdomain on each node, where several processors share a global memory address space. We can notice that such mixed implementation perfectly matches the mixed memory management available on cluster of SMPs. The purpose of the next sections is to compare the performance of the pure message passing strategy with a mixed approach on two clusters of bi-processor PCs and on a Compaq ES40.

4 Experimental results

Because we are interested in the parallel behaviour of the cluster of SMPs on the algorithm described in the previous section, we choose to perform scaled experiments. That is, the number of the mesh points is increased linearly with the number of computing entities (threads and/or MPI processes) such that the amount of work per computing entity remains constant and independent of the number of processors used. Because the amount of computation for our explicit scheme is linear with respect to the number of mesh points, the ideal overall computing time should be constant and independent of the number of processors. For most of our numerical experiments we consider 512×512 grid points per computing entity so that the memory space requires is about 16 MB (i.e. larger than the L2 cache on the Compaq Alpha processor and the L3 cache of the Alinka Itanium) and 200 time steps.

4.1 Pure MPI implementation

In this section we report on numerical experiments where only the coarse grain parallelism is exploited; the global address space available on each node is not used by the algorithm even though it is actually used by the implementations of MPI on some platforms for the communication between processes running on the same node.

4.1.1 Experiments on the cluster of Pentium based PCs

# processors	2	4	8
Elapsed time (sec)	21.6	23.2	23.5

Table 2: Pentium cluster - Pure MPI implementation - 1 process per node.

# processors	2	4	8	16
Elapsed time (sec)	31.3	32.0	32.9	33.1

Table 3: Pentium cluster - Pure MPI implementation - 2 processes per node.

4.1.2 Experiments on the cluster of Alinka Itanium based PCs

The elapsed time reported in the table below have been obtained using the SGI compiler with the best found combination of optimization options. According to experiments not reported in this note the code generated by the SGI compiler was less efficient than the one generated by the Intel compiler. Unfortunately we were not able to use MPI with the Intel compiler available at that time.

# processors	1	2
Elapsed time (sec)	25.0	26.7

Table 4: Alinka Itanium cluster - Pure MPI implementation - 1 process per node.

# processors	2	4
Elapsed time (sec)	29.3	31.1

Table 5: Alinka Itanium cluster - Pure MPI implementation - 2 processes per node.

# processors	2	4
Elapsed time (sec)	13.9	14.0

Table 6: Compaq - Pure MPI implementation - 1 process per node.

# processors	2	4	8
Elapsed time (sec)	16.2	16.0	16.3

Table 7: Compaq - Pure MPI implementation - 2 processes per node.

# processors	4	8	16
Elapsed time (sec)	16.4	16.6	16.9

Table 8: Compaq - Pure MPI implementation - 4 processes per node.

4.1.3 Experiments on the Compaq

4.1.4 Observations

Looking at each table independently, it can be seen that in each configuration (i.e. same number of MPI processes per node) the performance of the three platforms scales almost perfectly. However, if we compare the performance in the different configurations we can observe that the performance deteriorates when the number of processes per node is increased. On the Compaq this deterioration of performance is 15 % when moving from one process per node to two processes per node, and less than 5 % when moving from two to four processes per node. This increase is less than 10 % on the Alinka Itanium platform when using the two processors versus only one. On the cluster of Pentium the situation is much worse, the performance decrease is about 40 % when using the two processors of the node compared to using only one.

At that stage we can ask whether this loss of performance is due to the fact that the processors share the memory or share the network interconnect device ? In order to further investigate this question we consider in the next section experiments where we either use only fine grain parallelism on one node or mix both fine grain parallelism via OpenMP and coarse grain parallelism with MPI. For all the experiments, we assign one MPI multi-threaded process per node. With such implementation the number of processors sharing the memory of each node can be varied, while only one performs the message exchanges and consequently uses the network interconnection. This will enable to distinguish whether the memory or the network access is the main bottleneck.

4.2 Mixed OpenMP and MPI implementation

For all the experiments, we select the best OpenMP parallel loop scheduling option, that was STATIC on the Compaq and on the Alinka Itanium and RUNTIME on the cluster of Pentium. The loop-level parallelism is illustrated in Algorithm 1 where the most time consuming part of the explicit scheme is given. It corresponds to a sparse matrix-vector product where the matrix is stored in the Compressed Sparse Row format. The two other computing kernels that have been parallelized similarly are a 2-norm calculation of vectors, via OpenMP reduction, and a vector copy.

```
!$OMP PARALLEL SHARED(a,x,y,ia,ja), PRIVATE(i,j)
!$OMP DO SCHEDULE(STATIC)
       do i=1,n
         y(i) = 0.0d0
         do j=ia(i), ia(i+1)-1
           y(i) = y(i) + a(j)*x(ja(j))
         enddo
       enddo
!$OMP END DO
!$OMP END PARALLEL
```

Algorithm 1: OpenMP directives to parallelize the sparse matrix-vector product.

In Table 9 and 11, we report experiments where only one multi-threaded MPI process is used. Actually in such a situation no message is exchanged, and then the possible network bottleneck is removed. In Table 10, we depict elapsed time observed on the Alinka Itanium platforms. Those results have been obtained with the Intel Compilers and the best found combination of optimization options. Since we were not able to link the resulting code with any of the two MPI libraries available at that time on that platform we just discarded the MPI calls. This does not affect those results where only the parallelism generated by OpenMP is exploited. Consequently those results are comparable with those depicted in Table 9 and 11.

# threads	1	2
Elapsed time (sec)	19.7	28.9

Table 9: Pentium Cluster - One MPI multi-threaded process.

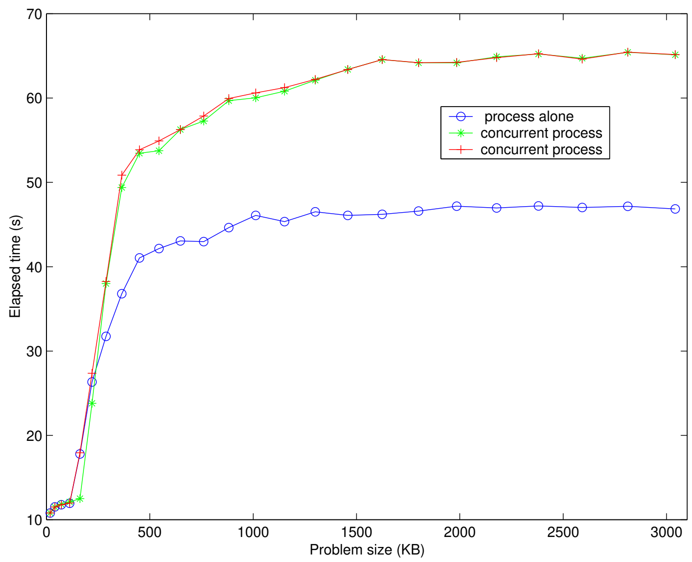
# threads	1	2
Elapsed time (sec)	12.4	14.5

Table 10: Alinka Itanium Cluster - One MPI multi-threaded process.

# threads	1	2	4
Elapsed time (sec)	12.9	13.7	14.9

Table 11: Compaq - One MPI multi-threaded process.

Similarly to what was observed with the pure MPI implementation when increasing the number of processes per node, the performance deteriorates when the number of threads is increased. The loss of performance is also about the same and fairly dramatic on the cluster of Pentium PC where almost half of the computing power of one CPU is lost when the two CPUs of a node are used. To fully assess that the loss of performance is due to the memory contention and not possibly to a poor implementation of the OpenMP threads, we plot in Figure 2, 3 and 4 the elapsed time required to perform a few time steps of the sequential explicit scheme on problems of increasing size while varying the number of independent processes running simultaneously on the same node. For those experiments the number of time steps is adjusted so that the amount of floating point operations per process remains almost constant when the size of the grid is increased (i.e. number of time steps times number of grid points is constant). In abscissae we give the amount of memory space needed by the processes for solving the problem.

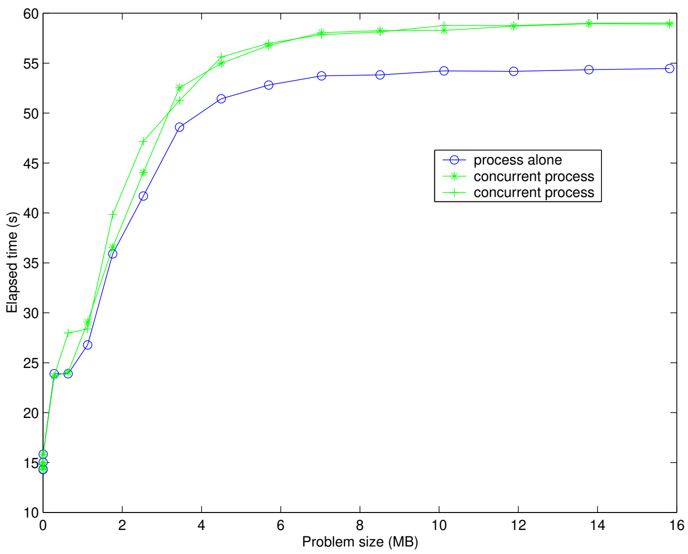


(a) 2 concurrent processes

Figure 2: Elapsed time varying the number of independent concurrent processes on the Pentium PC cluster.

Those experiments confirm that the main bottleneck for the performance on the cluster of Pentium PC is the shared memory. It can be seen in Figure 2 than when two independent concurrent processes run simultaneously the memory contention slows down very much the computation. On small problems that fit into the L2-cahe, the 3 curves perfectly overlap. On large enough problems, the elapsed time required by the two concurrent processes is about 45 % more than the time required by a process running alone for solving the same problem. Even though an increase is also observed on the Alinka Itanium cluster and on the Compaq, the gap are smaller. On the Compaq we observed about a 8 % increase for two processes up-to around 24 % with four processes, on the Alinka Itanium platform this increase is around 9 %.

It can be noticed that those plots also enable to determine some cache size. In Figure 2, the jump in the elapsed time occurs for problems of size around 256 KB, that corresponds to the size of the L2 cache. Even though it is not depicted a zoom close to the origin also reveals the size of the L1 cache. Similarly, in Figure 4 and 3 the jumps occur for problems of size around 2 MB on the Alinka Itanium platform that is the size of the L3-cache; and around 8 MB that is the size of the L2 cache of the Alpha processors. For problems small enough to fit into the caches, all the curves perfectly overlap. This is another clue that confirms that the loss of performance is due to contention occurring when the shared memory is accessed. To conclude with those experiments on the effect of the memory load on the performance, we should mention that our code does not implement any cache blocking strategy. In that respect it corresponds to the worse situation for memory contention; the shared memory is



(a) 2 concurrent processes

Figure 3: Elapsed time varying the number of independent concurrent processes on the Alinka Itanium cluster.

very much stressed. In Figure 5, 6 and 7, we report the results of similar experiments, where the explicit scheme is replaced by optimized BLAS 3 [3] Matrix-Matrix multiply DGEMM (for the Pentium cluster and Alinka Itanium cluster we used the ATLAS [8] BLAS and on the Compaq the vendor optimized library). These figures show that when the code makes the best possible use of the caches the memory contention decreases significantly. There is almost no loss on the Alinka Itanium platform and the Compaq (i.e. around 2 % increase of elapsed time) and it reduces to 16 % on the bi-processor Pentium (to be compared with around 45 % without any specific cache reuse policy).

The memory contention being now illustrated, we continue our experiments using multi-threaded MPI processes for the solution of the heat equation using the explicit scheme described in Section 3. None experiments are reported on the Alinka Itanium cluster since no Fortran compiler available at the time of these experiments was able to support simultaneously OpenMP and MPI. In Table 12 we report elapsed times observed on the cluster of Pentium PCs when each MPI process spawns two OpenMP threads. Those figures can be compared with those displayed in Table 3; it can be noticed that the pure MPI approach and the mixed MPI-OpenMP exhibit comparable performance.

# processors	4	8	16
Elapsed time (sec)	31.9	32.2	32.3

Table 12: Pentium PC cluster - 2 threads per MPI process (# processors = 2 $\times$ MPI processes).

In Table 13 and 14 we report similar experiments on the Compaq with respectively two threads per MPI process and four threads per MPI process. Those results can be compared with those in Table 7 and 8. They show that on the Compaq the MPI-OpenMP implementation is slightly less efficient than the pure MPI one, especially when four threads per MPI process are used. This difference is nevertheless not significant and more probably due to a worse cache reuse in multi-threading calculation than to a high multi-threading overhead.

# processors	4	8
Elapsed time (sec)	15.1	15.8

Table 13: Compaq - 2 threads per MPI process (# processors = 2 $\times$ MPI processes).

# processors	8	16
Elapsed time (sec)	17.3	19.1

Table 14: Compaq - 4 threads per MPI process (# processors = 4 $\times$ MPI processes).

5 Concluding remarks

In this work we have investigated the use of cluster of SMPs for parallel numerical simulations. Through numerical experiments using a simple benchmark code we show that combining message passing and loop level parallelism enable to get attractive efficiencies. Those basic experiments have revealed that the key issue to get good overall performance resides in the ability of the computer hardware or the implemented software to efficiently share the local memory between the processors within the node. In that context, cache blocking strategies appear to be very much important. Not only these cache reuse strategies enable to get the best performance out of each individual processor but also good performance out of the overall computing node. The cache blocking strategies contribute to reduce the shared memory contention that appears to potentially be a severe bottleneck.

Possible fields for application of this embedded parallelism are numerous and include the development of numerical librairies [4]. A very important one to enable the use of those computers in industry are some large industrial codes that have been developed over years on vector computers and then moved, with a significant manpower effort, on parallel distributed vector platforms with moderate number of processors. Using OpenMP, to parallelize most of the vectorial loops, in combination with MPI is a viable opportunity for smoothly moving those codes on this emerging and promising platforms for intensive scientific computing that are the clusters of SMPs. Another illustration can be owned to linear algebra where we can mention domain decomposition or more generally block preconditioning techniques for the solution of large sparse linear systems. For those techniques the numerical scalability is often related to the number of blocks or subdomains. In classical parallel distributed implementations one assigns one block per processor. Consequently increasing the number of processors for solving a given problem results in a less efficient numerical solver and then does not fully take advantage of the computing power of all the processors. Combining the two programming models enables to efficiently exploit some parallelism at a block level, through the use of parallel direct solvers for shared memory for instance. In that context the number of processors used to perform a given simulation can be increased without deteriorating the numerical property of the numerical algorithm since the number of blocks does not need to be increased. Such behaviour is shared with some multi-block compressible Navier Stokes solvers (see for instance [1]), where increasing the number of processors might imply to explicit more the numerical scheme and then deteriorates the convergence toward the steady state.

Acknowledgments

I would like to thank R. Clint Whaley for his email support installing ATLAS on the Alinka Itanium as well as the Alinka company for providing us with an access to their developer Alinka Itanium platforms. My sincere thanks to the people from the Computer Support Group at CERFACS. Especially Isabelle d'Ast and Gérard Déjean for their help in getting quickly started on the two computers just after their installation. I want also to thank Nicolas Monnier for providing me with many details on the computer architectures.

References

[1] J. Bohbot, G. Grondin, and D. Darracq. A parallel multigrid conservative patched/sliding mesh algorithm for turbulent flow computation of 3d complex aircraft configurations. In AIAA 2001-1006, editor, *39th AIAA Aerospace Sciences Meeting*, Reno, USA, 2001.

[2] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[3] J. J. Dongarra, J. J. Du Croz, I. S. Duff, , and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

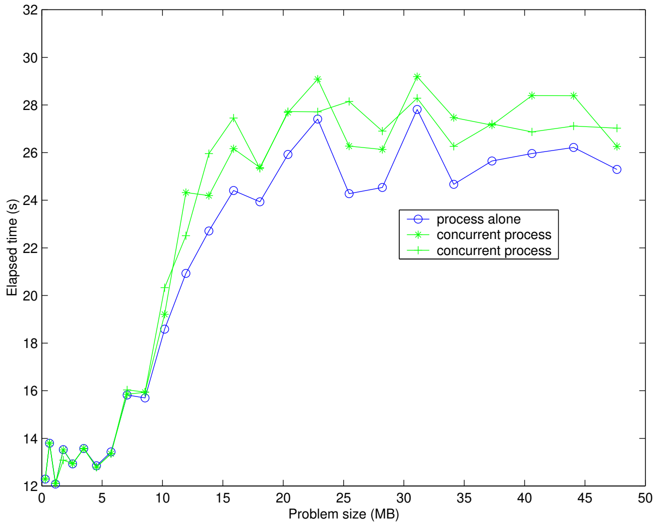
[4] J. J. Dongarra, S. Moore, and A. Trefethen. Numerical libraries and tools for scalable parallel cluster computing. *Int J. of High Performance Computing Applications*, 15(2):175–180, 2001.

[5] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

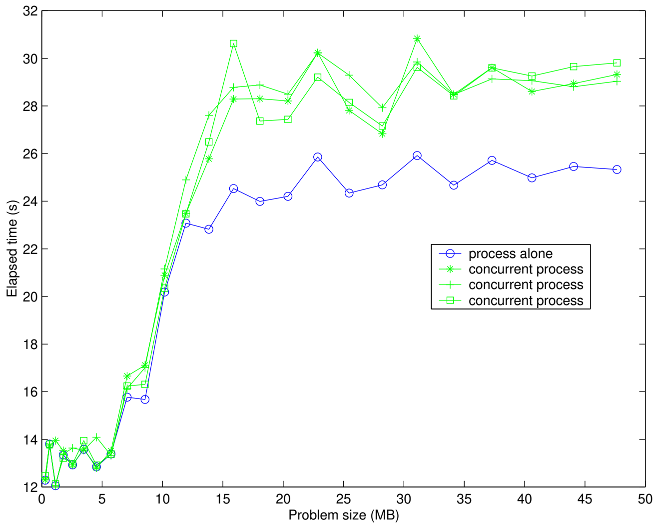
[6] Message Passing Interface Forum. MPI: A message-passing interface standard. Int. J. Supercomputer Applications and High Performance Computing, 8(3/4), 1994. Special issue on MPI.

[7] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface. Technical Report Versin 2.0, 2000.

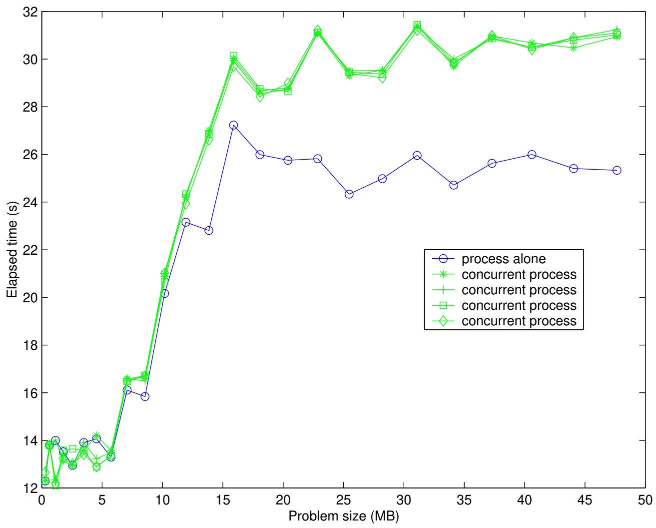
[8] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, submitted, 2000.



(a) 2 concurrent processes



(b) 3 concurrent processes



(c) 4 concurrent processes

Figure 4: Elapsed time varying the number of independent concurrent processes on the Compaq.

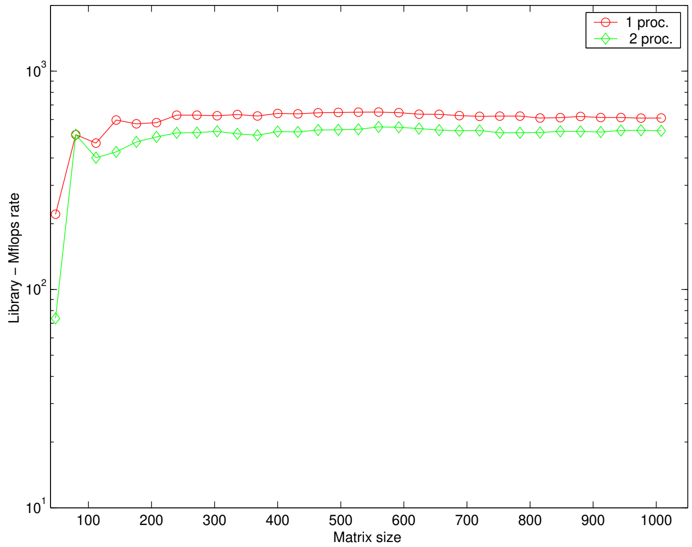


Figure 5: DGEMM MFlops rate varying the number of independent concurrent processes on the Pentium PC cluster.

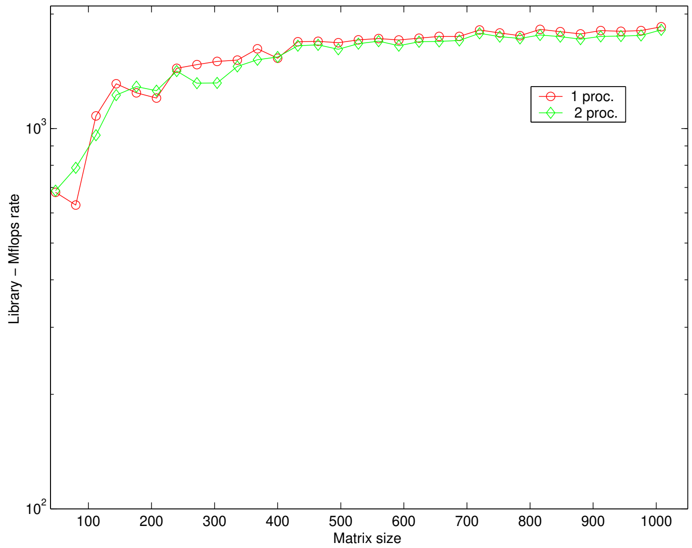


Figure 6: DGEMM MFlops rate varying the number of independent concurrent processes on the Alinka Itanium PC cluster.

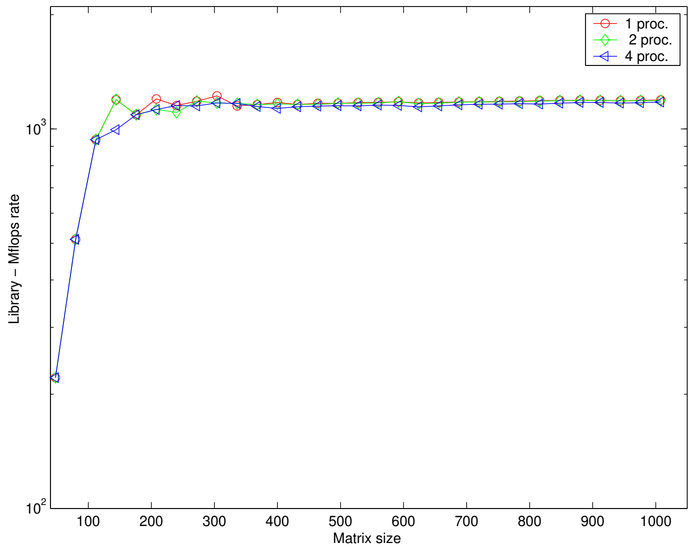


Figure 7: DGEMM MFlops rate varying the number of independent concurrent processes on the Compaq.

A Bandwidth benchmark

In this appendix we report some bandwidth measurements performed to evaluate the actual communication throughput. The code used intend to measure the bandwidth in full duplex mode and is basically composed by the following MPI calls:

```
MPI_Is end(buffer,size,MPI_DOUBLE,neigh,TAG1,MPI_COMA_WORLD,&request) ;
MPI_Reck(buffer,size,MPI_DOUBLE,neigh,TAG1,MPI_COMA_WORLD,&status) ;
MPI_Wait(&request,&status);
MPI_Is end(buffer,size,MPI_DOUBLE,neigh,TAG2,MPI_COMA_WORLD,&request) ;
MPI_Reck(buffer,size,MPI_DOUBLE,neigh,TAG2,MPI_COMA_WORLD,&status) ;
MPI_Wait(&request,&status);
```

A.1 Experiments on the PC cluster

In order to evaluate the communication bandwidth, the following experiments have been performed:

1. one pair of processes running on the same node, communication performed through the shared memory,
2. one pair of processes running on the same node, communication performed through Myrinet,
3. one pair of processes, each process runs on a different node, communication performed through Myrinet,
4. two independent pairs of processes running on two different nodes, that each each process of a given pair is on a different node, communication performed through the network.

The observed bandwidths are depicted in Figure 8.

A.2 Experiments on the Compaq

In order to evaluate the communication bandwidth, the following experiments have been performed:

1. one pair of processes running on the same node, communication performed through the shared memory,
2. one pair of processes, each process runs on a different node, communication performed through the network,
3. two independent pairs of processes running on two different nodes, communication performed through the shared memory,
4. two independent pairs of processes running on two different nodes, that each each process of a given pair is on a different node, communication performed through the network.

The observed bandwidths are depicted in Figure 9.

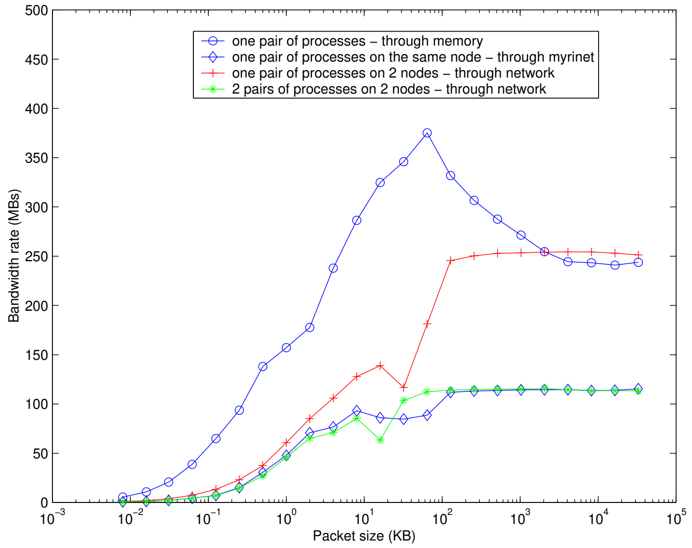


Figure 8: Measured bandwidth for a given pair of processes varying the size of the messages on the PC cluster.

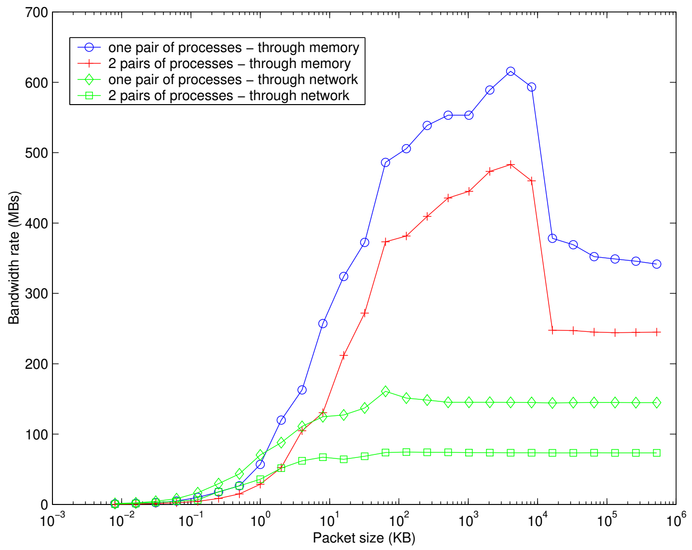


Figure 9: Measured bandwidth for a given pair of processes varying the size of the messages on the Compaq.