Intra node parallelization of MPI programs with OpenMP

Franck Cappello and Olivier Richard
LRI, Université Paris-Sud, 91405
Orsay, France
Email: fci@lri.fr.

September 28, 1998

Abstract

The availability of multiprocessors and high performance networks offer the opportunity to construct CLUMPs (Cluster of Multiprocessors) and use them as parallel computing platforms. The main distinctive feature of the CLUMP architecture over the usual parallel computers is its hybrid memory model (message passing between the nodes and shared memory inside the nodes). Some of the primary issues to address for the CLUMP are: 1) to be able to execute the existing programs with few modifications 2) to provide some programming models coherent with the performance hierarchy of the data movements inside the CLUMP 3) to limit the effort of the programmer while ensuring the portability of the codes on a wide variety of CLUMP configurations. We investigate an approach based on the MPI and OpenMP standards. The approach consists in the intra-node parallelization of the MPI programs with an OpenMP directive based parallel compiler. The paper presents a detailed study of the approach in the context of the biprocessor PC CLUMPs. It provides three contributions. First, it evaluates the ability of biprocessor PCs to effectively provide a speed up over single processor PCs in the context of shared memory parallel programs. Second, it investigates the method to transform MPI parallel programs in order to execute them on a CLUMP. Third, it presents the performance evaluation of this method applied on the NAS parallel benchmarks executed on a cluster of biprocessor PCs.

1 Introduction

Many computing centers are now equipped with parallel platforms using PCs as computing nodes and a high performance network like Myrinet as the interconnection network. Today, most of the microprocessors (and especially the Pentium family) and their chip-set are designed to easily build multiprocessors. One direct effect of this design strategy is that biprocessor PCs come to the market roughly at the same time as single processor PCs for a given generation of microprocessor. One other effect is the relatively low ratio of multiprocessor PC cost / single-processor PC cost. Since the cost of the network connection is a large portion of the cost of a PC node in a parallel platform, it becomes appealing to use multiprocessor PCs as the nodes for a parallel platform.

Networks of multiprocessors PC present an hybrid memory model: message passing between nodes and shared memory inside each node and conform to the term of CLUMP (CLUster of MultiProcessors). It exists several ways to program the CLUMPs. [1] and [2] presents respectively a taxonomy and a classification of the programming model for the CLUMPs. We may classify two main approaches by distinguishing the programmer view of the memory in a CLUMP:

- a single memory model (SMM) or
- an hydride memory model (HMM)

In the SMM approach the programmer only see one memory model. A mechanism is added to the platform to unify the view of the memory model. The main drawback of the SMM approach is the performance cost of the unifying mechanisms.

1.1 Message passing SMM

[3], [4] and [2] have developed a version of their message passing libraries to work in SMP platforms. We also have ported a version of the BIP [5] library for SMP PC. The BIP library had been initially developed for the Myrinet communication network. The cost of message passing in shared memory has three components: First, the system bus will be shared for intra-node communication. So, colliding communications must be sequentialized. This is a main disadvantage over a high performance network with point to point communications. Second, a message passing requires at least one buffer copy from the sender buffer to the receiver buffer even when it is performed in the shared memory. This will add some substantial extra memory movements over the ideal situation where the shared memory execution model is used inside the SMP nodes. Third, the message passing implementation in shared memory (i.e. the communications occur within the shared memory by buffer copies) may reach communication performance lower inside the SMP than between separate nodes. For example, in our implementation of BIP in shared memory, we obtain approximatively half the bandwidth of inter-node BIP performance. The bandwidth is reduced due to the PC system bus performance (500 MB/s with 66 Mhz system bus and 750 MB/s for 100 Mhz system bus) and the need to cross the bus two times for each communication: buffer read and buffer write. Despite the use of optimized memory copy or DMA copy mechanisms, the communication bandwidth reaches a maximum of 500 Mb/s. Inter-node BIP performance reaches 1 Gb/s.

[6] presents a model to program the CLUMP through a small kernel of collective communication and computation primitives. The communication primitives are implemented on top of a message passing library and a SMP Library. Algorithms implemented with this model are programmed as a succession of collective computations and collective communications.

1.2 Shared Memory SMM

Shared Virtual Memory environments provides the opportunity to program the CLUMP with the shared memory model. Several projects have already published some design and performance results about the CLUMP:[7], Shasta [8], Cashmere-2L [9] and SoftFLASH [10]. As for the mono-processor platforms, the performance of DVSM mainly relies on the protocol efficiency.

Other programming paradigms, initially developed in the context of the single-processor node parallel computers, may also unify the memory model for the CLUMP.

Split C [11] is a parallel extension to the C programming language that support access to a global address space on distributed memory architectures. Split-C allows the programmer to use two types of pointers: standard pointers and global pointers. Global pointers reference the entire address space. Standard pointers reference only a memory region local to the processor. By distinguishing local and remote memory accesses, Split-c gives the programmer a way to manage the locality of the data references.

Recently, OpenMP [12] has been implemented on a network of workstations on top of the Treadmark DSM system. It provides a very convenient way to program the distributed memory architectures. Its performances relies on the shared virtual memory software.

1.3 HMM

HMM has been the first approach used to program the CLUMPs. Original works include programming the PVPs (Parallel Vector Processor) where Vector Supercomputers are interconnected with a high speed network to form a parallel architecture. HMM programming has been the theme of a workshop in 1995 [13]. Several contributions concerned the porting of applications from the parallel computers to the CLUMPs. Recently the NAS parallel benchmarks repository has published the performance of a Power Challenge Array on a selection of the NPB2 benchmark programs [14].

[15] presents a hybrid shared memory/distributed memory programming model for the CLUMP. Intra-node computation utilizes a multi-threaded programming style. Inter-node programming is based on message passing and remote memory operations. The hybrid programming model is based on the SPMD programming style. The programmer has to manage the data set partition and distribution. Data sets are first partitioned between the nodes. Each distributed data set portion is then partitioned among the threads within each node.

1.4 Portability requirement

In both cases SMM and HMM, a main issue for the programmer is the portability of his code. Moving from traditional supercomputers (vector machine) to shared memory or message passing parallel computers has already forced the users to reconsider their application programming. Some programming standards have emerged for parallel computers: HPF for data parallel computing and MPI for message passing. Shared memory parallel computers are programmed using automatic parallelizers, directive based parallelizing compilers or directly using a process or thread API. Moving from single processor nodes to multiprocessor nodes in the parallel architectures may also require an effort. However, a methodology to program the CLUMPs should seriously consider the portability criterion and provide an approach compliant with a wide variety of CLUMP configurations.

1.5 A method based on MPI and OpenMP

In this paper, we investigate a methodology which primary aims to provide portable codes with a reduced effort in the context of the HMM approach. The approach uses OpenMP for shared memory parallelism inside the nodes and MPI for message passing between nodes.

OpenMP derives from the ANSI X3H5 standards effort. It is a set of compiler directives and runtime library routines that extend a sequential programming language to express shared memory parallelism. The language is extended by a collection of compiler directives, library routines, and environment variables. The directives include parallel region constructs (OMP PARALLEL), work-sharing constructs (OMP DO) that apply on DO loops to specify how iterations should be split among the threads, and synchronization constructs. OpenMP conforms to the SPMD programming paradigm. The OpenMP API uses the fork-join model of parallel execution. Program execution begins as a single process called the Master thread of execution. The master thread executes sequentially outside the parallel regions. A parallel region is enclose between a pair PARALLEL and a END PARALLEL directives. When the master thread enters a parallel region it creates a thread team. Then, all threads in the team execute the statements of the parallel region. The threads are synchronized upon the completion of the parallel region. The work sharing construct DO allow to distribute the iterations of a do loop among the threads already existing in the parallel region. By default the scope for variables in a parallel region is SHARED. A clause section in the format of the PARALLEL and DO directives allows the user to control the scope attributes of the variables within the directives. The clauses define a data environment for each member of the thread team. In particular, the PRIVATE clause can be used to create dedicated copies of some variables for each thread of the team.

MPI is one of the post popular library for message passing for multi-PC parallel platforms. A lot of applications have been written or ported for the message passing paradigm. A methodology proposed to program the CLUMPs from MPI should also: A) provide a way to execute the existing the MPI programs written for mono-processor nodes, B) ensure that the programs implemented with this methodology will work with mono-processor nodes.

OpenMP is used in the methodology to express the intra-node parallelism from the Fortran code of the MPI programs. This two steps framework corresponds to the hierarchy of granularities of the data movements a) between SMP nodes and b) inside a SMP node.

When programming in the message passing style, the application is constructed as large communicating processes. Data and works are distributed among the nodes in order to 1) minimize the number of communications, 2) pack the data to communicate in long messages and 3) overlap the communications by computations. These optimizations are required due to the impact of the communication cost on the global performance.

In shared memory parallel programming, data movements are also managed in order to limit their contribution on the total execution time. The sequence of memory references are optimized by the compiler or/and the programmer in order to exhibit temporal and spatial localities. However, the basic memory movements in shared memory is usually a cache block. Moving cache blocs across the network of a parallel architecture and especially across a network designed for message passing has two main drawbacks: a) the time to make a remote reference (few microseconds) is huge for the processor point of view and it has to spin a long time (thousands of instructions). b) the bandwidth provided by a high speed network may be wasted with messages as small as a cache bloc. The non blocking cache and the data prefetch mechanisms are

used to reduce the penalty of the remote memory references. The block transfer engines in the distributed shared memory machines provide a way to reach the bandwidth of a high speed network by transfering large memory blocs across the network. An other way to reduce the remote memory penalty is to force the shared memory data references to stay inside the node.

Before presenting the method for executing MPI-OpenMP programs on the CLUMP, we evaluate the ability of the biprocessor PCs to effectively provide a speed up over the single processor PCs for numerical applications. We also compare biprocessor versus mono-processor nodes in the domain of the global platform cost.

2 Biprocessor versus mono-processor nodes

We have measured the performance of a multi-threaded version of the SPLASH2 [16] benchmark on multi-processor PCs to evaluate the interest of using them as nodes for a parallel platform. The data sets used for the measurements are presented in the table 1.

	LU	FFT	Cholesky	radix	Ocean	Water nsqured	fmm	Water spq	Radiosity
data set	2048	20/22	tk29.0	$2^2 4$	514	inuput2 4096	16384	4096	default

Table 1: Data sets for the SPASH2 benchmark programs used for the performance measurements

This study has been performed on three successive generations of biprocessor PCs (Pentium PRO 200, Pentium II 300 and Pentium II 400). The main differences between the Pentium PRO 200 nodes and the Pentium II 300 nodes are the CPU cycle time, the cache size (L1:2*8kB vs 2*16kB, L2: 256kB vs 512kB) and the memory technology and pipelining (EDO vs SDRAM). The differences between the Pentium II 300 nodes and the Pentium II 400 nodes are the CPU cycle time and the system bus cycle time. The figure 1 presents the speed-up of biprocessor nodes versus single processor nodes for the three generations.

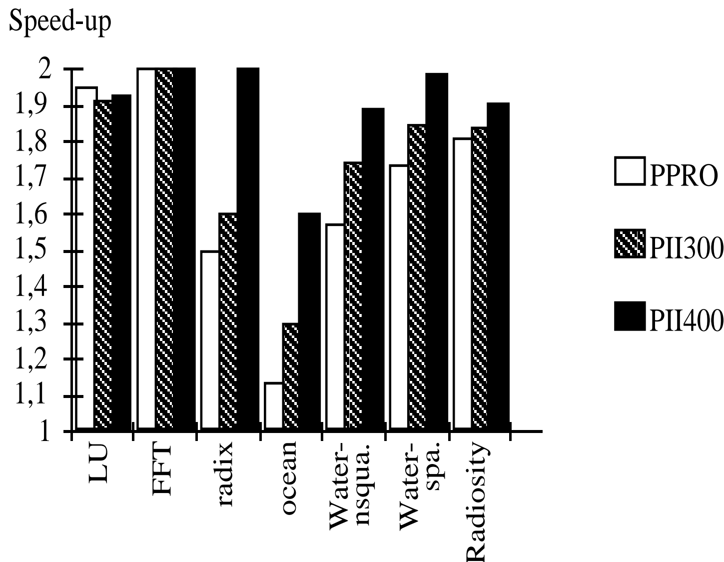


Figure 1: *Speed-up of biprocessor versus single processor for three generations of microprocessors*

Recent biprocessors provide remarkable speedup over single processor PC for these benchmark programs. For most of the benchmark programs, the last generation provide a higher speedup than the previous ones.

Two main generation evolutions explains this results : larger processor caches (2*8kB and 256kB for Pentium Pro versus 2* 16kB and 512kB for Pentium II) and a lower [processor frequency / bus frequency] ratio (4,5 for the Pentium II 300 and 4 for the Pentium II 400). For these programs, larger caches and a lower [processor frequency / bus frequency] ratio reduce the memory penalty and lower the impact of the bus accesses for each processor.

The figure 2 presents the speed-up among the three generations for single processor.

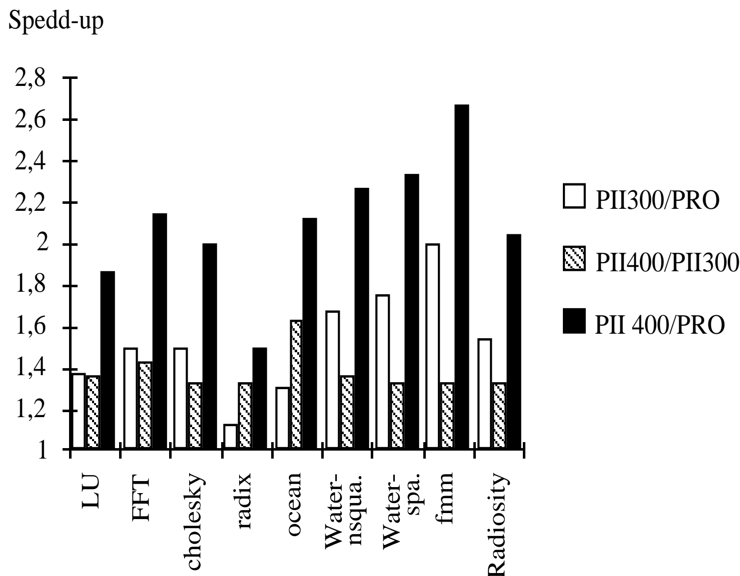


Figure 2: *Speed-up among three generations of mono-processors*

For most of the benchmark programs, the increase of the CPU frequency and the cache size (Pentium PRO versus Pentium II 300) allows to provide a speed-up about 1.3. This speed-up evolves with the locality properties of the data references. The programs exhibiting a high locality of data references do not take benefice of the higher bus frequency of the Pentium II 400. So for these programs the speed-up is close to the ratio of the processor frequency (400/300 = 1.33). The higher speed-ups for the Pentium II 400 over the Pentium II 300 occur for the programs with the less data reference localities.

Biprocessors seems to be good candidates as nodes for parallel platform because: 1) they offer the opportunity of a high speedup over single processor nodes and 2) their speedup over single processor remains stable across the processor generations.

2.1 Cost of biprocessor

Using biprocessor PCs as nodes for a message passing parallel platform may reduce their speedup over single processor nodes due to the difficulty to efficiently program a parallel computer with an hybrid memory model.

Multiprocessors nodes offer two potential interests for message passing parallel platforms. First, according to their speedup over single processor nodes they may longer resist to the next generation mono-processor. The Figure 3 present the speed-up of biprocessors PC of a given generation (g) over single processor PC of the next generation (g+1). For the SPLASH2 benchmark, the PII 300 biprocessor keep a substantial speed-up over the single processor PII 400 except for the OCEAN program.

Using biprocessors instead of single processors as the nodes increases the parallel platform cost. As for the other potential applications of the parallelism, we have to compare the cost/performance ratios of the biprocessor and the single processor nodes.

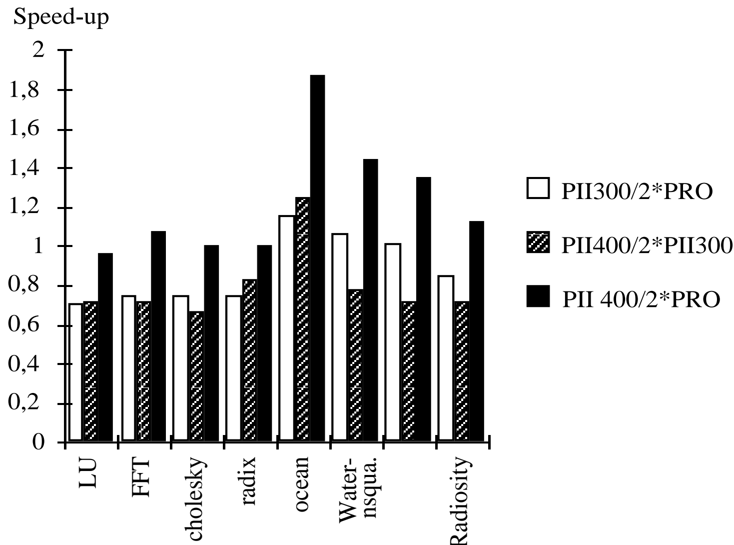


Figure 3: *Speed-up of the generation g mono-processor over the generation g-1 and g-2 biprocessors*

The second potential interest of multiprocessor nodes resides in the reduced number of the network connections for a given number of processors in a parallel platform. The Speed- up of biprocessor nodes over single processor nodes seems to promise identical performance for the parallel platforms composed of n single processors nodes or of n/2 biprocessor nodes.

This section has shown that the biprocessors are very promising as the nodes of parallel platforms. The next section proposes an approach to execute MPI programs on the CLUMPs. Although they provide non-negligible speed-ups for some programs, the result section will moderate our opinion about the performance of the biprocessor used as the nodes of a parallel platform.

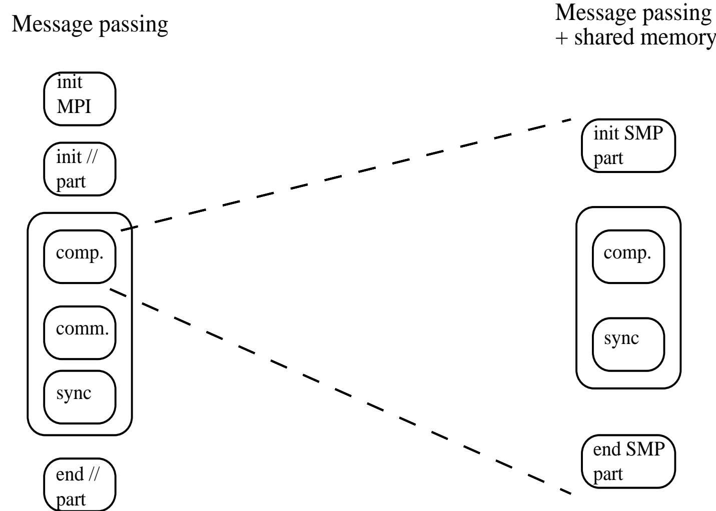
3 Intra MPI processes parallelization with OpenMP

Although the methodology described in this paper can be applied in the context of MIMD applications, we describe it only in the context of applications programmed for the SPMD execution model.

3.1 The basic methodology

Parallelizing an application for the message passing following the SPMD paradigm often produces a program with the typical layout presented in figure 4. The program starts initializing the communication system. Then it performs some local computations and calls some communication subroutines to split the data sets among the nodes participating to the application. The program continues executing a main block typically containing a loop nest. The main block is designed to be executed in parallel. The body of the loop nest can be described as a succession of three sections. The first section executes the node local computations. The second section communicates some partial results to the other nodes. The third section synchronizes the nodes before the next loop nest iteration. The final part of the program gathers the individual partial results and computes the final result.

The applications written from MPI programs come in the form of one executable file that has to be ran on each node of the parallel platform. Within each node, the program is executed inside a process. As previously mentioned, this process computes on local data and communicates with the processes on the other nodes. Parallelizing the program executed on each node lead to parallelize the main block. This block often encompass an iterative calculus with inter-iteration dependencies. So intra-process parallelization could not be attempted at this level in the general case. For most of the NAS programs, we have parallelized




```
program cg
…
call initialize_mpi
…
call setup_proc_info( num_procs, …)
call setup_submatrix_info( l2npcols,…)
…
do it = 1, niter
         …
         call conj_grad ( colidx,…)
              …
         do i = 1, l2npcols
           call mpi_irecv( norm_temp2,…)
           call mpi_send(  norm_temp1,…)

           call mpi_wait( request,…)
              …
        enddo
        …
endo
…
call mpi_finalize(ierr)
…
end
```

```
subroutine conj_grad( colidx,…)
      do i = 1, l2npcols
         call mpi_irecv( rho,…)
         call mpi_send(  sum,…)
         call mpi_wait( request,…)
      enddo
       …
!$OMP PARALLEL PRIVATE(k,sum)
!$OMP DO
         do j=1,lastrow-firstrow+1
            sum = 0.d0
            do k=rowstr(j),rowstr(j+1)-1
               sum = sum +
a(k)*p(colidx(k))
            enddo
            w(j) = sum
         enddo
!$OMP END DO
!$OMP END PARALLEL
       …
         do i = l2npcols, 1, -1
            call mpi_irecv(…)
            call mpi_send(…)
            call mpi_wait( request,…)
         enddo
       …
      return
      end
```

Figure 4: *Parallelizing the MPI code. The main loop nest of the CG code calls the conj-grad subroutines and contains some communication calls. The computation loop nest of conj-grad is parallelized for intra-node execution using the shared memory paradigm.*

the computational section of the main block. Figure 4 presents this hierarchy of parallelism: message passing between MPI processes and intra-node parallelism within each MPI process. The intra-node parallel execution is performed by a group of threads within the same process. The parallelization directives are simply applied on the parallelizable loop nests of the original MPI code. From the other nodes point of view, a biprocessor node with an intra node multi-threaded execution of a fraction of the MPI code behaves like a mono-processor node.

The PGI fortran compiler used for our experiment is a subset of the OpenMP Fortran Application Program Interface. The PGI thread implementation is based on the Pthread which is a de-facto standard.

When there is no inter-iteration dependencies in the program main loop nest, the parallelization can be attempted at the main bloc level. In such a case, all intra-node parallel tasks will execute the communication

calls. The program correctness may require a total order for the communication performed by each node. In that case, it is the responsibility of the programmer to add the necessary code to control the operation sequence.

3.2 Selecting the loop to parallelize

The application may exhibit a large number of loop nests and subroutine calls. It can be difficult to discover manually (reading the text source) which loop nests are worth while to parallelize.

In general, three parameters distinguish the loop nests to parallelize to the others: 1) of course, it must exist a parallelization of the loop nest that respect the semantic of the sequential version, 2) the loop nest must have a subtential contribution to the total execution time, 3) the body of the loop nests must be long enough to make the parallelization overhead negligible.

The first parameter should be examined looking for the dependencies. It may also require some inter-procedural analysis when the candidate loop nests contains some subroutine calls in their body. The second parameter is obtained using a profiled execution of the application. It provides the respective cost of each subroutine and each loop nest. The third parameter must be estimated according to the cost of the parallel operations in OpenMP. Table 2 gives the cost of the main parallel operations for several nodes of our platform.

	Pentium Pro 200	Pentium II 300	Pentium II 400
Fork-Join (Parallel DO)	5 us	3.5 us	3 us
Lock (Critical)	1.66 us	1.45 us	1.4 us
Barrier	1.36 us	1.33 us	1 us

Table 2: The cost of the parallel operations of OpenMP on our platform

The figure 5 presents a framework for selecting the loop nests to parallelize.

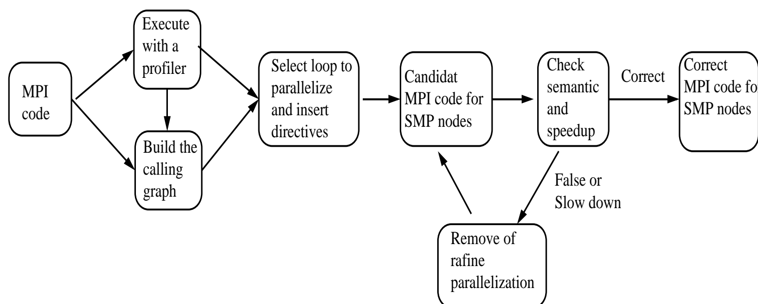


Figure 5: *The parallelization framework*

The framework begins with the MPI source file. The profiled execution allow to discover the most expensive loop nests. It is not always judicious to parallelize all expensive loop nests. Assuming that the calling graph is a tree, the rule to select the loop nests should be to investigate the loop parallelization from the root of the calling graph to the leaf. Parallelizing at the highest level in the calling graph allows to reduce the number of fork and join operations. It also allows to distribute more job to each thread. The figure 6 presents a situation where the loop nest parallelization may be used at two different levels in the calling hierarchy.

Assume that the three loop nests are parallelizable. The loop nest 1 is at a higher level on the calling graph. Parallelizing at this level allow to use a single fork-join operation and include b2 in the parallel execution. Parallelizing at the loop nest 2 and the loop nest 3 level requires 2 n fork join operations (PARALLEL DO), reduces the amount of operations attributed to each thread and leads the execute b2 sequentially.

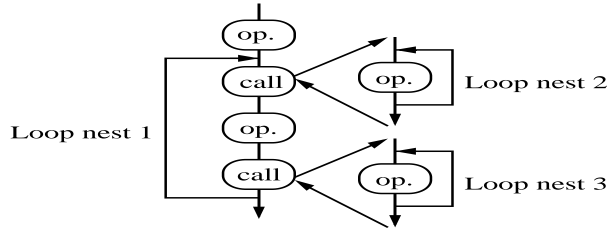


Figure 6: *A situation where the loop nest are parallelizable at two different levels of the calling hierarchy*

The next step after selecting the loop nests to parallelize is to add the OpenMP directives in the code in order to obtain a candidate MPI code for SMP nodes. Then the program is compiled and ran on the platform. At the end of the execution, the result correctness and the speed-up must be checked. If the result is falls or if the speed-up is less than one, the parallelization must be refined or removed. This process may require several iterations in order to provide an efficient and correct code.

3.3 Main conceptual advantage and limit of the method

3.3.1 Advantage

The intra-node parallelization with this methodology lead to an interesting property: all nodes still only see a single entity on each other node i.e. a single MPI process. This property may be very useful for a platform with heterogeneous nodes. For example, a parallel platform with a variety of multiprocessor nodes and some mono-processor nodes will be able to execute all MPI programs. Such a platform could be useful in the context the SPMD model assuming a balance of the individual performance of all node. For example, according to the results of our SPLASH 2 experiment, a platform using Pentium II 400Mhz mono-processor nodes and Pentium Pro 200 Mhz biprocessor nodes will be balanced. So the methodology provides a way to add up to date nodes in a parallel platform while continuing to use older nodes.

3.3.2 Limit

A limit of the approach comes from the way the shared memory parallelization is applied to the message passing programs. With this method, we should not expect a local speed up close to the speedup one can obtain by directly parallelize a sequential program. As shown on the figure 7, the intra node parallelization only concerns the computation part of the main bloc of the original MPI program.

In contrast with a shared memory program directly derived from a sequential program, there is a lot of substantial work that can not be parallelized. More precisely, the speed up is not only bounded by the local sequential part contribution to the local execution time (Amdahl's law) but also by the communication and synchronization contributions to the local execution time. Unfortunately the Gustafson low does not apply here as it does for usual parallel programs because the communication time may evolve as a function of the data set size.

4 Intra node parallelization of the NAS NPB 2.3 Benchmarks

The methodology gives a general framework for the program parallelization. In this section, we will discuss the parallelization of the programs of the NAS NPB 2.3 [17]benchmark suite. Despite several programs require some specific optimizations to reach a useful speedup, we try to limit our investment to a "reasonable effort": the rule was to obtain a reasonable speed-up (in term of the cost/performance ratio). In particular we do not investigate the memory hierarchy nor the I/O optimization effects. Also, we do not intent to parallelize the loops with a cost less than 1% of the total execution time. The parallelization of all the programs takes about one month.

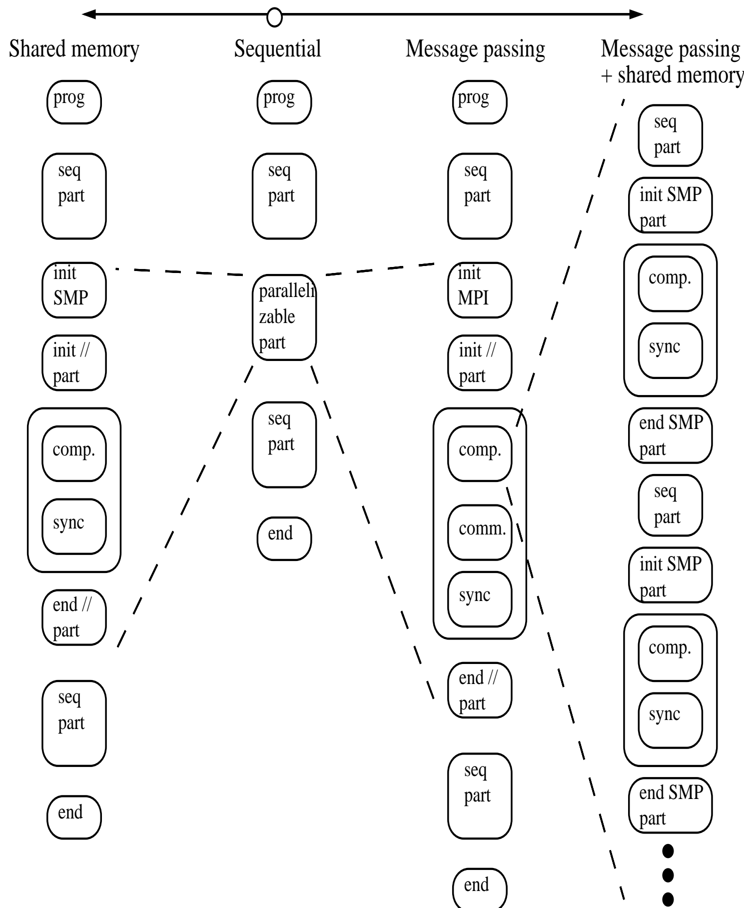


Figure 7: *Expansion of the parallelization overhead. Parallelizing for the shared memory or the message passing paradigm requires to add some code to the original program. Parallelizing simultaneously with both paradigms leads to add the initialization and the ending sequences of the both*

For each benchmark, we have profiled a single node execution. This approach minimizes the communication cost and help the programmer to select the loop to parallelize. In other hand, the cost of all loops changes when the program is executed in a multi-node version. The communication cost and the distribution of the work among the nodes reduce the cost of the selected loops.

We also choose to profile the execution of the class W benchmark despite we intent to parallelize for the class A version. Profiling the execution of the class A programs would give more accurate figures for each loop cost. However profiling the execution of a program with a wide data set, a very long program or a large set of programs may be not practical in general. Since this approach is attempted to be useful with a reduced effort, we parallelize the class A version from the profiled execution of the class W version.

The result of the profiled execution is presented by a calling graph with the cost of the main subroutines. Two costs are reported. The local cost do not includes the cost of the subroutines called by the current subroutine. The total cost includes the cost of the routines called by the current subroutine. When the parallelization is not trivial, we detail the structure of the subroutines and its loops.

LU, FT and CG are very easy to parallelize following our methodology. MG, EP, SP and BT require more efforts and some dedicated optimizations.

4.1 LU

LU factors a dense matrix into the product of a lower triangular and a upper triangular matrix. The figure 8 presents the calling hierarchy and the cost of each function as returned by the profiled execution of the original MPI version. The initialization sequence counts for less than 1% of the total execution time.

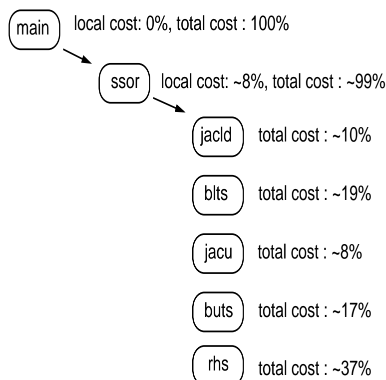


Figure 8: *The calling graph of LU*

ssor contains four main loops. Two of these loops are parallelizable. The two other loops are not directly parallelizable at least because they call some communication subroutines. rsh contains a sequence of 6 loop nests. All the 6 loop nests have been parallelized. There are 2 calls to a communication procedure inside the sequence between the loop nests. blts and buts begins and ends with a communication between nodes.They also contains two loop nests of which only one (with the smallest body) is parallelizable. Finally, jacld and jacu contain one loop nest with a long body which is parallelizable.

4.2 FT

FT implements a 3-D FFT to solve a partial differential equation. The 3-D FFT performs 1-D FFTs in each dimension by calling the cffts(1,2,3) subroutines. The figure 9 presents the hierarchy of the main function calls and their respective weights. The initialization are not presented in the figure. Their execution times are included in the computation of the benchmark performance. Two subroutines (compute-indexmap and compute-initial-conditions) total 5% of the execution time. We do not investigate their parallelization.

The fft routine contains a loop which cannot be parallelized at least because it contains some calls to the transpose routine which, in turn, contains some communications calls. So the parallelization should be applied only from the next call levels i.e. the cffts(1,2,3) routines. The cffts(1,2,3) have the same structure. There is a single loop nest containing a call to the cfftz routine. As presented in [18], an inter-procedural analysis shows that the subroutine calls are independents. So the outermost loop of the cffts(1,2,3) loop nest is parallelizable providing the opportunity to get substantial job for each thread.

4.3 CG

GC Solves an unstructured sparse linear system by the conjugate gradient method. It uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros. As shown in figure 10, most of the execution time is spent within the conj-grad subroutine. However, initialization procedure for the CG program has a higher cost than for the previous ones: about 7%. The percentage in the figure are not normalized. The first loop nest and the second one are responsible respectively of 92% and 4% of the conj-grad procedure execution time. conj-grad contains some other loop nests. All of them count for less than 1% of the procedure execution time. Nevertheless, together these loop nests cost 4% of the procedure execution time.

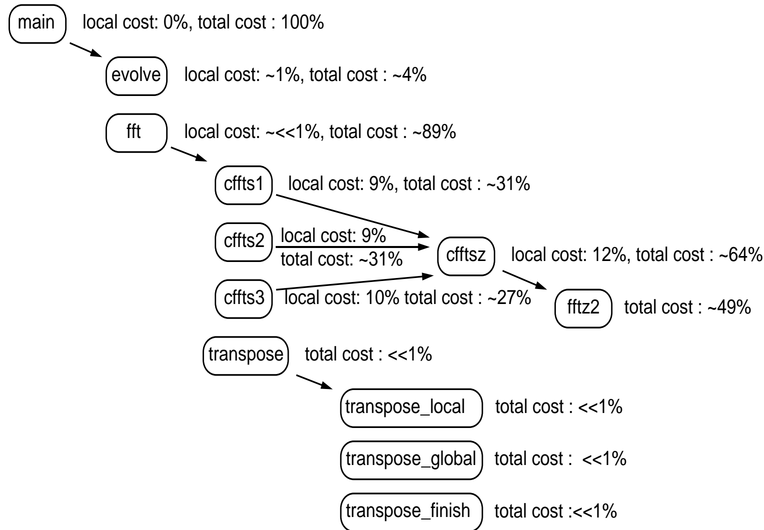


Figure 9: *The calling graph of FT*

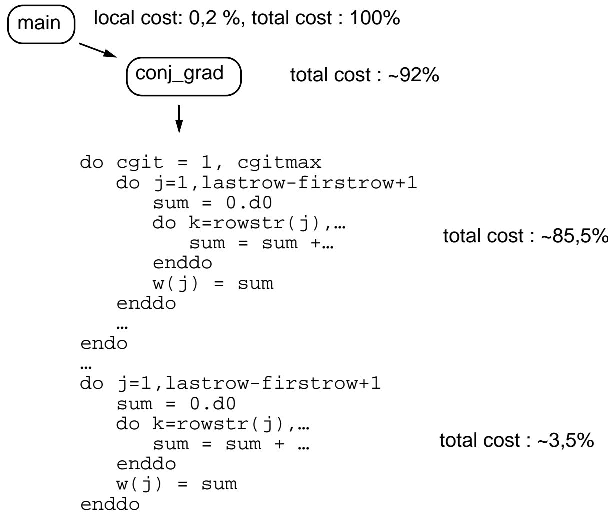


Figure 10: *The calling graph and main loop nests of CG*

We have parallelized the two main loop nests (j index).

4.4 MG

MG uses a multi-grid algorithm to obtain an approximate solution of a three-dimensional scalar Poisson equation. MG is more complicated to parallelize essentially because its execution time is split among more than ten subroutines. As for the previous figures, the figure 11 presents the calling hierarchy of the parallelized subroutines. The figures (% of the total execution time) must be understood taking into account a very long

initialization procedure. The initialization time in W class is approximatively 66% of the total execution time. Some time consuming subroutines can not be parallelized (zran3, comm1p, norm2u3, comm3) because they contain communication calls or inter-iteration dependencies. Their combined execution time exceeds 30% of the total execution time.

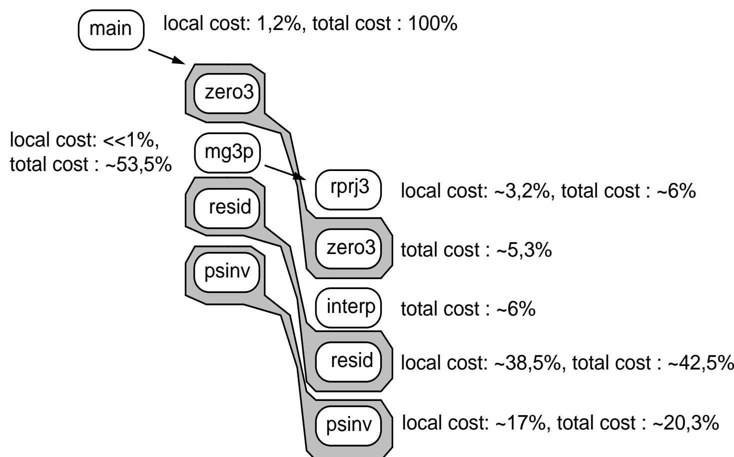


Figure 11: *The calling graph of MG*

mg3p is not parallelizable because it calls some communication subroutines. zero3, rprj3, interp, resid and psinv contains loop nests that are parallelizable at the outermost loop.

4.5 EP

EP generates pairs of Gaussian random deviates according to a specific scheme and tabulates the number of pairs in successive square annuli. With our methodology EP does not provide the property promised by its name. It is a typical case where the programmer has to add some code and carefully manage the threads. As shown in figure 12, EP contains one main loop nest. The initialization delay is negligible for EP. For each MPI parallel process, most of the execution time (95%) is spent in the two innermost loops.

The loop nests do not execute any communication call. The communications occur at the end of the program by the way of three reduction operations. The local computation provides a q array and two scalars (sx and sy) in each node. The reduction operations add the value of the distributed q arrays and (sx, sy) scalars. The parallelization of the outermost or the innermost loop leads to create dedicated storage of sx, sy and q for each local thread. In both cases, some code must be added before the reduction operations to combine the partial results computed by the threads within each node. We parallelize the outermost loop. sx, sy and q becomes arrays indexed by the thread number. They are reduced locally to scalars (sx and sy) or to one dimensional array (q) before the reduction operations.

4.6 SP

The SP solves three sets of uncoupled systems of equations in the x, y, and z direction. These systems are scalar pentadiagonal. The execution time of the SP benchmark is split among ten main subroutines. As for EP, the initialization cost can be neglected. So percentages of the total execution time can be considered as is. The figure 13 presents the calling graph of SP.

The local cost of copy-face is negligible. It calls the compute-rhs subroutine and some MPI communication routines. compute-rhs contains a long loop nest parallelizable at its outermost loop. It requires the dedication of fifteen private scalar variables for each thread. (x,y,z)-solve contain a succession of loop nests. Most of them are parallelizable at the outermost loop. The other generate or read messages to or from the communication

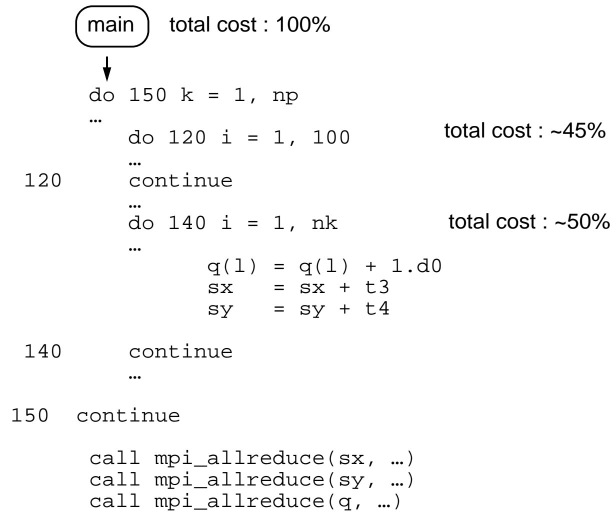


Figure 12: *The main loop nest of EP*

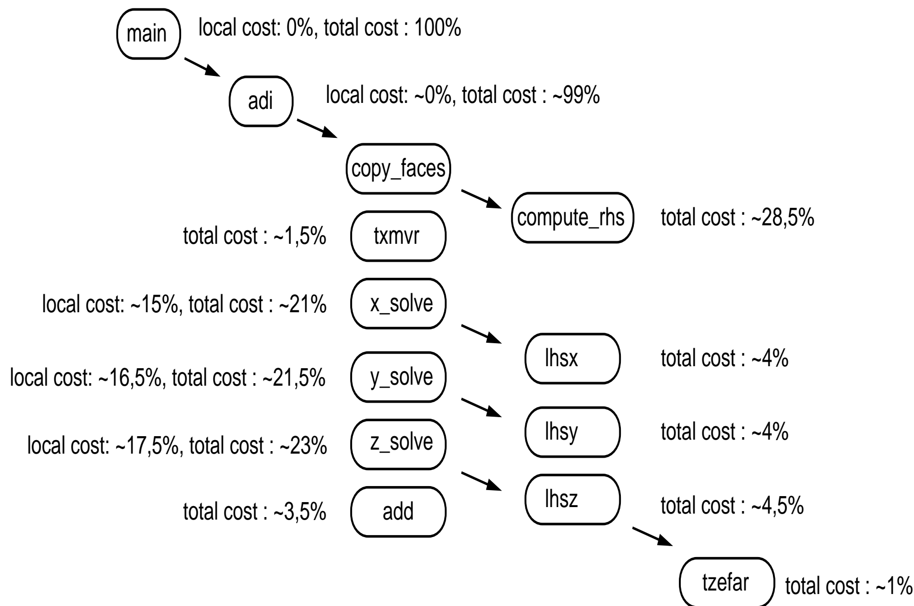


Figure 13: *The calling graph of SP*

buffers. These loop nests are not parallelizable due to an inter-iteration dependency: increment of the buffer pointer. One loop nests of z-solve requires a loop exchange to provide the good result when parallelized. lhs(x,y,z) contain five loop nest fully parallelizable. Finally, add, tzefar and txinvr contain one loop nest also fully parallelizable.

4.7 BT

BT has the same structure as SP. The three sets of uncoupled systems of equations are block tridiagonal with 5x5 blocks in the BT code. The figure 14 presents the calling hierarchy of BT. Like for EP the percentages of the total execution time can be considered as is.

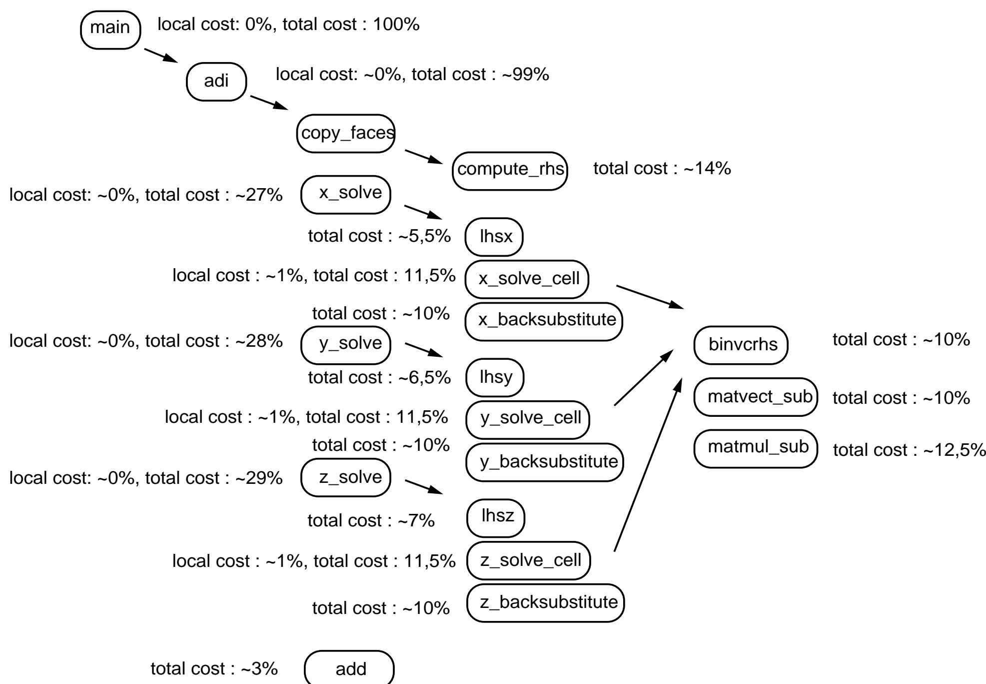


Figure 14: *The calling graph of BT*

copy-faces fills a communication buffer, executes some communication calls, reads the communication buffer and finally calls compute-rhs. The buffer manipulation loop nests are not parallelizables due to the buffer pointer increment within the innermost loop. compute-rhs contains one long loop nest fully parallelizable at the outermost loop. (x,y,z)-solve make subroutine and communication calls. lhs(x,y,z) contain two loop nests fully parallelizable. (x,y,z)-solve-cell call matvec-sub, matmul-sub and binvcrhs within a loop nest. The inter-procedural analysis does not reveal any inter-iteration dependencies for these subroutine calls. The loop nest can be parallelized at the outermost loop. (x,y,z)-backsubstitute contain one main loop nest fully parallelizable assuming a loop exchange of the two outermost loops for z-backsubstitute.

5 Performance

In this section, we start presenting the platform used for the performance evaluation. Then, we compare biprocessor versus single-processor configurations on the NAS 2.3 parallel benchmark. Finally, the CLUMP performances are compared against the performances of some high end supercomputers with the same benchmark.

5.1 Platform hardware and software

The platform contains a Myrinet network with four ports. We use three types of biprocessor nodes: Pentium Pro 200 Mhz, Pentium II 300 Mhz and Pentium II 400 Mhz. Each myrinet PCI interface has a 1MB local memory.

The software environment includes Linux 2.0.33, the BIP 0.94c version of the MPI library, the F77 PGI 1.7 programming environment and the Linux Pthread library. BIP raw performances on Myrinet connected PC is a latency of 5us and a bandwidth of 1 Gbit/s. MPI BIP reach 20 us (latency) and 1 Gb/s (bandwidth). All benchmarks have been compiled with the o2, unroll and P6 options.

With this platform we are not able to provide global speedup scalability results due to the modest number of nodes in our largest configuration. However, the global speedup scalability is not a main issue here. The NAS Benchmark repository already provides some scalability results of some message passing architectures (IBM SP2, NOW, P6 PC mono-processor connected by myrinet) executing the NAS benchmarks. From the global speedup point of view, each biprocessor node behaves like a fast mono-processor node. So the global speed-up versus number of nodes curve with biprocessor nodes are likely to follow the one of mono-processor nodes.

The relevant issue here is to evaluate the change of the local speed-up (within the biprocessors) with the number of nodes. As the result section will show, the local speedup evolve with the number of nodes depending of the application and the data set size. As the next section will show, some general trends can be derived for the local speedup even with a modest number of nodes.

5.2 Mono-processor PC versus biprocessor PC

In this section, we compare a parallel platform based on mono-processor nodes against a parallel platform based on biprocessor nodes. We compare these platforms for three different configurations: 1 node, 2 nodes and 4 nodes. All the measurements are made with Pentium II 300 Mhz nodes.

The figure 15 presents the speedup of the biprocessors based CLUMP over the single processor based platform for a constant number of nodes.

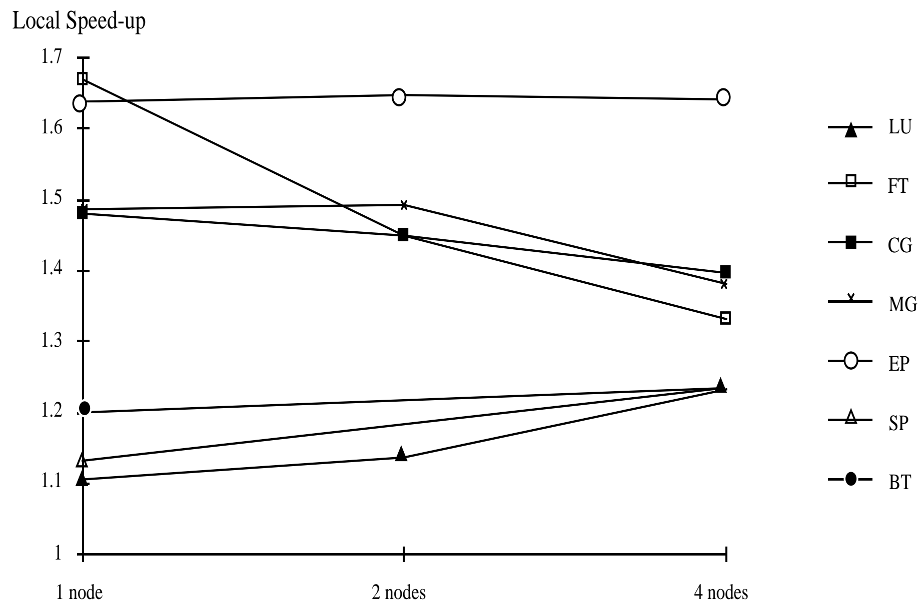


Figure 15: *Intra-node speed-up of the biprocessors over mono-processors for the NPB 2.3 Benchmarks and for the same number of nodes*

The speed-up evolves with the number of node in the CLUMPs following one of three trends:

- it remains constant (the inter node communications and synchronization have negligible impact on the global performance. Typically the application data sets are large and the main loops exhibit substantial parallelism),

- it increases (this is a case analogue to the previous one but reducing the data set attributed to each node may increase the memory hierarchy effectiveness by reducing the block conflict penalty).
- it decreases (the inter node communications significantly contribute to the global performance. With the increase of the number of nodes in the platform, each node has less data to compute for a given data set size. The communication time is marginally reduced by the increase of the number of nodes. So the communication time contribution to the local execution time increases with the number of nodes. The consequence is the reduction of the local speedup.

The figure 16 shows that Pentium II 400 biprocessors provide a more constant speed-ups across the different benchmarks. Their lower CPU frequency/bus frequency ratio help them to reach a higher speed up for the applications with the less data reference localities. They are also more sensitive to the communication cost.

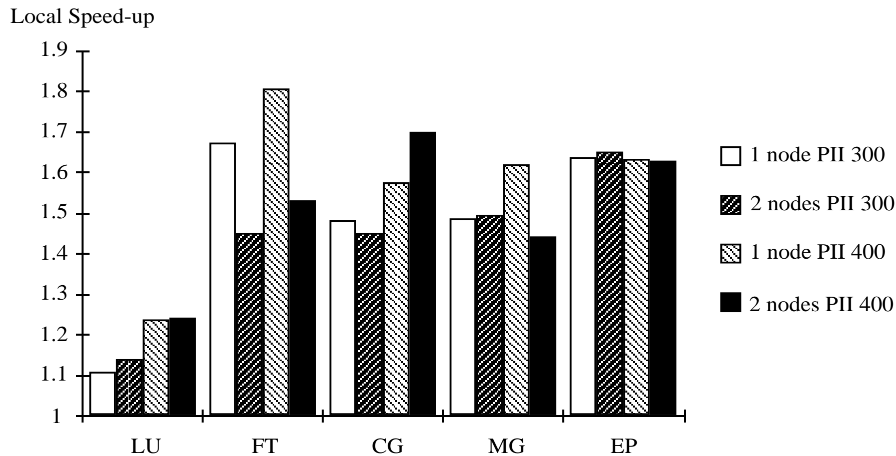


Figure 16: *Local speed-up of the biprocessors Pentium II 400 Mhz nodes and of the biprocessors Pentium II 300 Mhz nodes for the NPB 2.3 Benchmarks*

The figure 17 presents the performance of the biprocessors based CLUMP against the single processor based platform for a constant number of processors.

We should consider the cost/performance ratio of the biprocessor nodes against the mono-processor nodes. As we have previously mentioned, using multiprocessors instead of mono-processors as the nodes of a parallel platform allows to significantly reduce the number of network connections in the platform. A biprocessor based platform requires half the connections of a mono-processor based platform. This is a significant issue for the PC based parallel platforms because the cost of the network connections is a significant part of the global node cost. The PC based parallel platforms typically associate high performance PCs (but relatively low cost) with very high performance network (Myrinet, SCI, etc.). In a typical mono-processor PC based parallel platform, the network cost is the half of the total platform cost. Biprocessors are about 1.5 times more expensive than mono-processor for a given microprocessor and memory size. Assuming these ratio, for a given number of nodes, a biprocessor based platform is 1.25 times more expensive than a mono-processor based platform. According to the speed-up in the figure 15, biprocessor nodes justify their extra cost (up to 4 nodes, at least). For a constant number of processors, a biprocessors based platform is about 1.6 times less expensive than mono-processor based platform. The figure 17 shows that the global speed-up of the biprocessor based platform is higher than 0.6 except for LU. So biprocessor based platforms justify their usefulness by providing an alternative to the mono-processor based platforms.

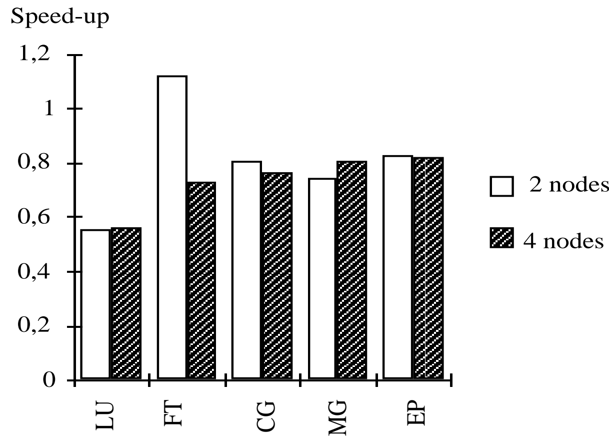


Figure 17: *Global speed-up of the biprocessors over mono-processors for the NPB 2.3 Benchmarks and for the same number of processors*

5.3 Network of biprocessor PC versus high end supercomputers

We compare a parallel platform based on biprocessor PC nodes against some of the most powerful parallel computers. The comparison is made for three different configurations: 1 node, 2 nodes and 4 nodes. The measurements for the two first configurations are made with Pentium II 400 Mhz biprocessor nodes. The figures for the 4 node performances are estimations (to the reviewer: the final version of the paper will include the actual measurements. The figures are obtained with the equation:

$$\text{4 Pentium II 400 Mhz biprocessor node performance} = \frac{\text{Single node Pentium II 400 Mhz biprocessor performance}}{\text{Single node Pentium II 300 Mhz biprocessor performance}} \times \text{4 Pentium II 300 Mhz biprocessor node performance}$$

Figure 18: *The equation used to compute the performance of the 4 nodes Pentium II 400 biprocessors*

The resulting figures should be understood as optimistics. The communications may have a greater influence for the Pentium II 400 nodes and they may reduce the speed-up of the Pentium II 400 nodes over the Pentium II 300 nodes for these programs.)

The figures 19,20 and 21 presents the performance of the biprocessors PC CLUMP against the SGI/CRAY T3E 900, T3E 1200, the SGI Origin 2k with 195 Mhz processors, the IBM SP2 with 66Mhz Power 2, the HP/Convex Exemplar SPP2000 and the SUN Ultra Entreprise 4000 for a constant number of nodes. The figures come from the NAS NPB2.3 repository.

The benchmark sources have been modified for the parallelization on the CLUMP with the OpenMP directives. However, less than 5% of the lines of the source code differs of the original code. For a same number of nodes (2 processors per node on the CLUMP), our PC based CLUMP approximatively reaches the performance of the supercomputers.

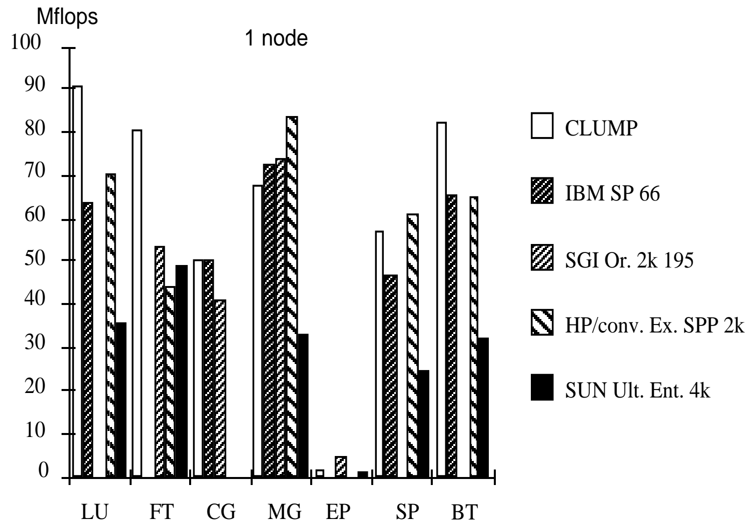


Figure 19: *Performance of the CLUMP and some parallel supercomputers on the NAS NPB 2.3 Benchmark suite for the single node configuration*

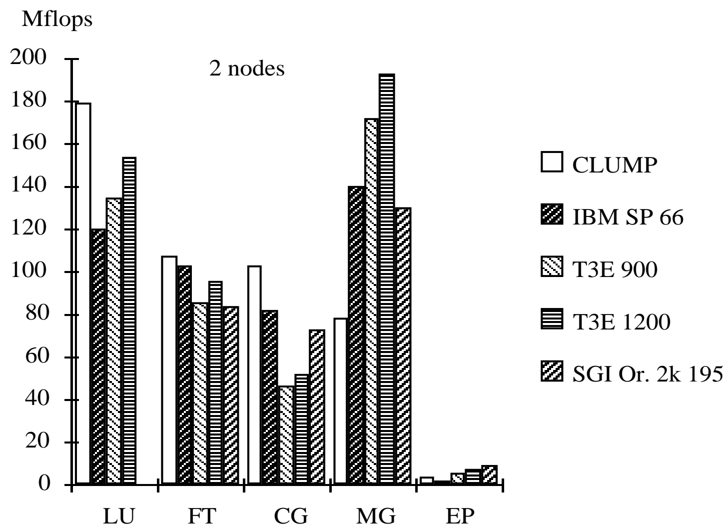


Figure 20: *Performance of the CLUMP and some parallel supercomputers on the NAS NPB 2.3 Benchmark suite for the configuration with 2 nodes*

6 Conclusion

In this paper, we have investigated a method for programming the cluster of multiprocessors. This method belongs to the HMM (hybrid memory model) approach. It requires the programmer to deal both with the message passing and the shared memory paradigms.

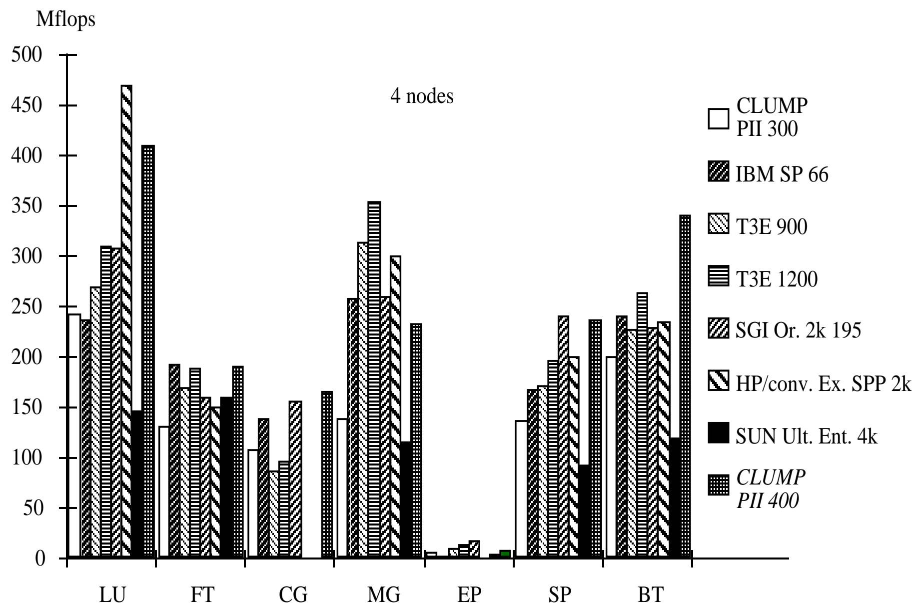


Figure 21: *Performance of the CLUMP and some parallel supercomputers on the NAS NPB 2.3 Benchmark suite for the configuration with 4 nodes*

The method consists in the intra-node parallelization of the MPI programs by using an OpenMP directives based parallel compiler. We have presented a framework to select the loop to parallelize. The NAS NPB 2.3 benchmark suite has been parallelized using this method. We have presented a detailed analysis of the intra-node parallelization for each benchmark program. We have pointed-out some advantages of the method: 1) the existing MPI programs can be reused with few modifications, 2) the programming model is coherent with the performance hierarchy of the data movements inside the CLUMP, 3) the effort of the programmer is limited while ensuring the portability of the codes on a wide variety of CLUMP configurations.

and also a main drawbacks: a sort of Amdahl's law governing the intra node speed-up.

A preliminary performance study has shown the potential speed-up of the biprocessor PC over the mono-processor PC. For most of the programs of the SPASH2 benchmark suite, the potential speed-up of the biprocessors is close to 2. The intra-node speed-up for the NAS parallel benchmark is lower (between 1.2 and 1.8 depending of the program for the PII 400 node). Moreover the speed-up evolves with the number of nodes following one of the three behavior: staying constant, reducing of increasing, depending of the benchmark program features.

Despite the method provides variable local speed-ups, it is much more practical than the manual parallelization approach to program the CLUMP. Using this method we have compared the performance of the CLUMP and some of the high end supercomputers from the NAS NPB 2.3 benchmark. For a same number of nodes (2 processors per node on the CLUMP), our PC based CLUMP approximatively reaches the performance of the supercomputers.

Finally, from the cost/performance point of view and under certain conditions, the biprocessors are a competitive alternative to the mono-processors the nodes of a parallel platform.

References

[1] E. L. Lusk W. W. Gropp. A taxonomy of programming models for symmetric multiprocessors and smp clusters. In *in Proceedings of Programming Models for Massively Parallel Computers*, pages 2–7, 1995.

[2] Hakon o. Bugge and Per O. Husoy. Efficient sar processing on the scali system. Report IPPS97, Scali Computer AS, 1997.

[3] M. Bernaschi. Efficient message passing on shared memory multiprocessors. *Lecture Notes in Computer Science*, 1156:221, 1996.

[4] Steven S. Lumetta, Alan Mainwaring, and David E. Culler. Multi-protocol active messages on a cluster of SMPs. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.

[5] L. Prylli and B. Tourancheau. Bip: a new protocol designed for high performance networking on myrinet. In *Workshop on Personal Computers based Networks Of Workstations*, 1998.

[6] David A. Bader and Joseph J J . SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). Technical Report CS-TR-3798 and UMIACS-TR-97-48, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, May 1997.

[7] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[8] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[9] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and Michael Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[10] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[11] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In IEEE, editor, *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 262–273, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[12] Charlie Hu Honghui Lu and Willy Zwaenepoel. Openmp on networks of workstations. In *Proc. of Super Computing 98*, Orlando, 1998.

[13] Mike Norman Karl-Heinz Winkler Bill Dannevik Michael Levine Matthew O'Keefe Paul R. Woodward, Larry Smarr. University of Minnesota, Minneapolis, 1995.

[14] NAS Parallel Benchmark Home page. http://science.nas.nasa.gov/software/npb/. Technical report.

[15] M. Ando K. Kazuto Y. Tanaka, M. Matsuda and M. Sato. Compas: A pentium pro pc-based smp cluster and its experience. In *IPPS Workshop on Personal Computer Based Networks of Workstations*, pages 486–497. LNCS, 1998.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

[17] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Mail Stop T 27 A-1, Moffett Field, CA 94035-1000, USA, December 1995.

[18] Abdul Waheed and Jerry Yan. Parallelization of nas benchmarks for shared memory multiprocessors. In Marian Bubak Peter Sloot and Bob Hertzberger, editors, *HPCN'98: High Performance Computing and Networking: Proceedings of the 1998 Conference: April, 1998, Amsterdam, The Netherlands.*, pages 377–385. Springer verlag, 1998.