

OpenMP and MPI programming with a CG algorithm

P. Kloos¹, P. Blaise² and F. Mathey¹

¹ CEA/DI, CEA Saclay, Bat. 474, 91191 Gif-sur-Yvette, France

² CEA/DI, CEA Grenoble, 17 rue des Martyrs, 38054 Grenoble Cedex 9, France

pkloos@cea.fr

Abstract

This paper focuses on how a conjugate gradient code running on a SMP cluster can benefit from shared memory using OpenMP. Different approaches of using OpenMP to implement shared memory parallelism are presented. The traditional way is to use OpenMP directives to parallelize loops. Another way is to decompose the loops in a SPMD (Single Program Multiple Data) programming style. The performance and scalability of two OpenMP versions of a conjugate gradient solver are compared. Calculations are performed on a cluster of Compaq Alphaserver nodes, each node having 4 CPUs. Experimental results with varying number of nodes and threads are shown. The results are interpreted with hardware considerations. A mixed MPI/OpenMP version is also presented.

Introduction

Clusters of SMP have recently become a widely used architecture. On such machines the use of message passing paradigm like MPI [5] is necessary for a program to be scalable on more than one node, since the CPUs of a node cannot access the memory of the other nodes. However, it may be interesting to use OpenMP to exploit efficiently the shared memory and avoid intranode message passing : to achieve this it is possible to mix the MPI and OpenMP paradigms, MPI and OpenMP being used respectively for the internode and intranode parallelism. This paper presents the optimization of the OpenMP implementation of the conjugate gradient (CG) on a regular mesh, and how the mixed MPI/OpenMP implementation improves the MPI version.

Other projects have focussed on implementing the CG algorithm with OpenMP. Piero Lanucara and Sergio Rovida [1] have compared the performance of CG solvers parallelized with OpenMP directives and DXNLP, a

multithreaded version of the BLAS. Dixie Hisley et al. [2] have compared the use of loop parallelization directives with the SPMD programming style to implement unstructured mesh applications (including the CG algorithm).

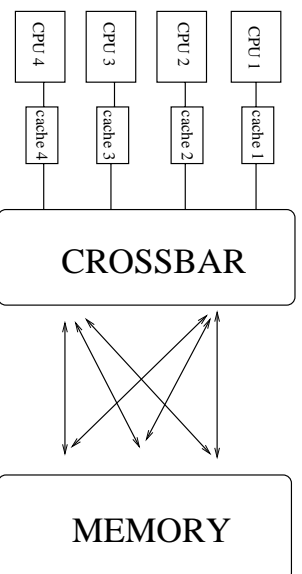


Figure 1: overview of a Compaq ES40

Hardware considerations

Calculations are performed on a cluster of Compaq ES40 [3] nodes. The nodes are interconnected with a fast Quadrics link at 200 MB/sec. Each node has four 667 Mhz Alpha CPUs and 4 Gigabytes shared memory. Each CPU is linked to a crossbar that has two connections to the memory (see Figure 1). Thus, all CPUs have a uniform view of the memory, which means that remote memory accesses may not affect performance as in the case of NUMA architectures.

On a ES40 node the memory bandwidth per CPU decreases with the number of CPUs used (see Figure 2). (This is also the case on other architectures [4]). The decreasing memory bandwidth has an influence on the performance of codes (this can be easily seen when running two sequential processes on the same node). To address this problem we calculate a "potential speedup" taking into account the memory bandwidth.

Each iteration of the CG algorithm involves one matrix vector products, two scalar products and 3 vector updates. The most expensive operation is the matrix vector product, and the algorithm is roughly $O(n^2 + 5n) =$

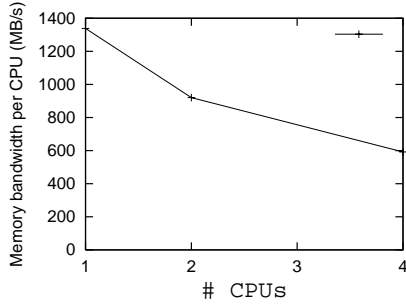


Figure 2: Memory bandwidth per CPU on the ES40 (McCalpin)

$O(n^2)$. When the algorithm is run in parallel on p CPUs, each CPU solves a $O(n^2/p)$ problem.

Otherwise, if we run p sequential problems each of size n/\sqrt{p} , we also have p CPUs each solving a $O(n^2/p)$ problem. Therefore we assume that on a SMP node a parallel program with p CPUs solving a problem of size n cannot run faster than p sequential programs solving concurrently p problems of size n/\sqrt{p} .

We shall use the latter to define the "potential speedup" :

$$\frac{\text{time for one } O(n^2) \text{ problem on a node}}{\text{time for } p \text{ sequential } O(n^2/p) \text{ problems on a node}}$$

Figure 3 shows the results of the "potential speedup" for the CG solver on a ES40 node. The fact that the slope is less than 1 comes from the memory bandwidth limitation.

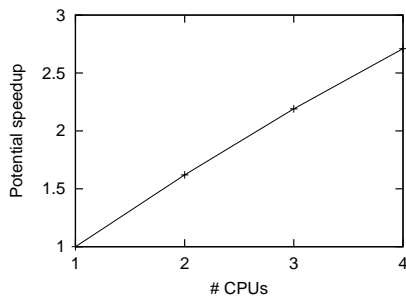


Figure 3: Potential speedup on a ES40 node

OpenMP programming styles

OpenMP is composed of a set of directives and some library routines (used to get the number of threads and the rank of the current thread, for example). With OpenMP there are two ways of distributing the data to the tasks :

- the usual way has been called "loop level parallelism" by Hisley et al. [2]. The iterations of loops

are divided between the threads, using directives. A simple loop parallelized this way looks like this :

```
!$OMP PARALLEL PRIVATE(i) SHARED(a,b,n)
!$OMP DO
DO i=1,n
    a(i) = a(i) + b(i)
ENDDO
!$OMP ENDDO
...
!$OMP END PARALLEL DO
```

- another way could be named "program level decomposition". Data are distributed once and for all at the start of the program. In that case each thread uses its rank and the number of threads to determine which part of the data it will process. This is the SPMD programming style also used with MPI. A simple loop parallelized this way looks like this :

```
!$OMP PARALLEL PRIVATE(start,end,i)
!$OMP+SHARED(a,b)
nb_threads = omp_get_num_threads()
thread_num = omp_get_thread_num()
start = n * thread_num / nb_threads + 1
end = n * (thread_num+1) / nb_threads
do i = start, end
    a(i) = a(i) + b(i)
enddo
...
!$OMP END PARALLEL
```

Performance of loop level and SPMD parallelism

Figure 4 shows the speedup achieved with loop level parallelism (with directives), compared with the "potential speedup" defined above and the speedup for the SPMD case.

The speedups for the directives case are similar to those obtained by Lanucara and Rovida with directives [1]. The speedups for the SPMD programming style are similar to those obtained by Lanucara and Rovida with the DXMLP library.

The speedup for the SPMD case is very close to the "potential speedup" calculated above, which shows that the memory bandwidth has a great influence on the performance of parallel programs running on SMP machines. We see that the scalability for the loop-level case is not very good. One reason is that some compiler optimizations are disabled by OpenMP directives. To illustrate this, let us compare the computation time of the sequential code compiled with or without the directives :

the run takes 72 seconds with directives versus 66 seconds without directives, so that a speedup of less than 1 is obtained for a 1 CPU run.

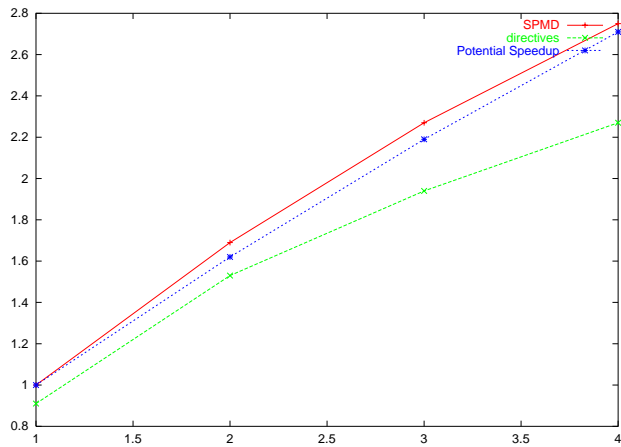


Figure 4: loop-level vs SPMD parallelism

Hisley *et al.* [2] have mentioned other reasons why loop level parallelism does not produce an ideal speedup. In our case, there is no load imbalance among the processors, because we use dedicated CPUs and the loop iterations are evenly distributed among the processors. On the other hand, there are no remote memory access overheads, due to the structure of the ES40, as we have mentioned above. An important factor is the cost of the synchronisation between the threads. Indeed at the beginning of each `!$OMP DO` loop threads are implicitly synchronised and the loop iterations are redistributed among the threads. On the contrary, in the SPMD case data are decomposed among the threads once and for all at the start of the program. Another factor affecting performance is the fact that the run time library may not decompose two successive loops in the same manner. That is, a portion of the iterations may be assigned to a certain thread in a loop, and to another thread in the next loop.

Data decomposition

In both versions (with directives and SPMD-style) the matrix vector product is computed by forming blocks of rows assigned to each thread. Each thread performs a local matrix vector product with a $n/nthreads \times n$ matrix (see Figure 5).

When accessing the multiplying vector a conflict occurs between the threads : all threads access the same portion of the vector at the same time. We have implemented a block matrix vector product, where each block of row of the matrix assigned to a thread is divided into $nthreads$ columns. The matrix-vector product is done in $nthreads$ steps. When a thread has finished a block,

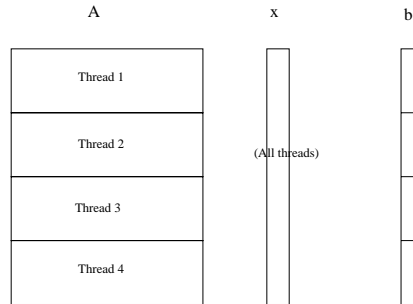


Figure 5: data decomposition for a matrix vector product

it continues with the next block in the same row and the next portion of the multiplying vector. All threads start the algorithm with a block in a different column. Figure 6 shows the first two steps of the algorithm.

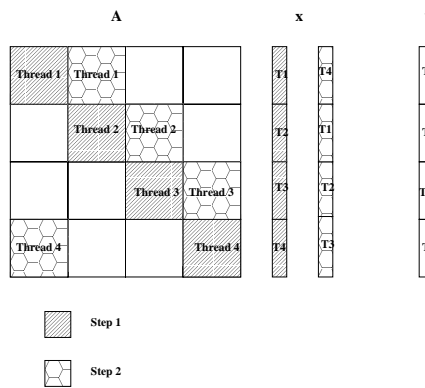


Figure 6: block matrix vector product

Figure 7 compares the speedup of the SPMD version with the original matrix vector product and with the block matrix vector product. The results are nearly the same with 2 CPUs, while the block version is about 5% faster with 4 CPUs.

Mixed MPI/OpenMP implementation

We now present a mixed MPI/OpenMP implementation of the CG algorithm. The data are distributed among the processes which synchronise each other and communicate via the MPI message passing library [5]. Each OpenMP thread attached to a process treats a part of the data assigned to this process. Figure 8 shows how the matrix vector product is implemented. The matrix is divided into blocks of columns, each block being assigned to a process. Each thread attached to a process is assigned a block of lines. The multiplying vector is distributed among the processes. In that way, the matrix vector product is performed in blocks

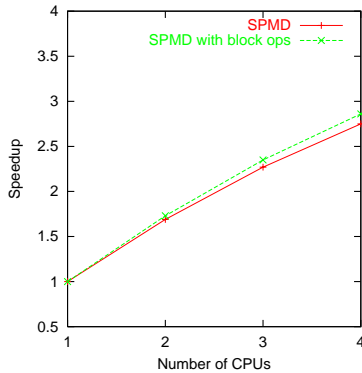


Figure 7: scalability gain with block operations

$A_{ij}x_j$ where i denotes the thread number and j the process number. When the block products have been performed each process owns the product of its column block of the matrix and its portion of the multiplying vector. An MPI reduction operation is performed to sum the local product of all processes.

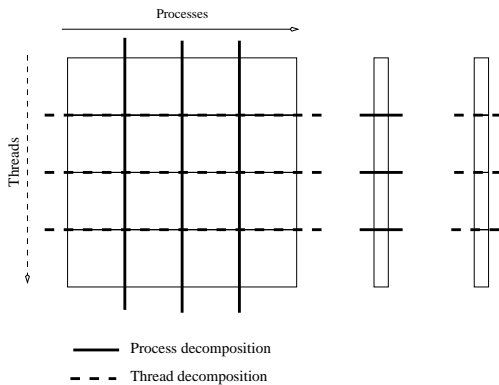


Figure 8: matrix vector product in the MPI/OpenMP implementation

The OpenMP part has been implemented in a SPMD style as shown above. A more elaborated matrix vector product can be done by using the block method in Figure 6 to perform the local matrix vector product of each process.

To perform the tests we have increased the problem size linearly with the number of nodes, to keep a constant problem size on each node. Since the CG algorithm is of complexity $O(n^2)$, the time used for the calculation should ideally grow linearly with the number of nodes. The mixed MPI/OpenMP implementation is intended to be run with one MPI process on each SMP node and to use OpenMP to synchronise the threads inside the node (e.g. two 4-CPU nodes, 1 process per node and 4 threads per process). However, it is also possible to use several MPI processes on the same node (e.g. two 4-CPU nodes, 2 processes per node and 2 threads per pro-

cess) or to use only MPI processes (without threads). Figure 9 shows the computation time for each possibility, as a function of the number of nodes.

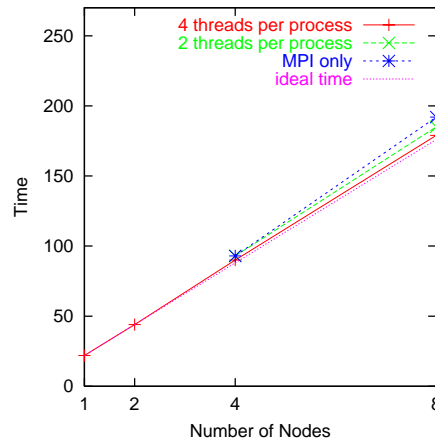


Figure 9: Computing time depending on number of threads per process

We see that the best time is obtained with the maximum number of threads (i.e. with one process per node). On eight nodes the MPI-only version is 7% slower than the MPI/OpenMP version with 4 threads per process.

Conclusion

We have seen that on the CG algorithm the OpenMP SPMD programming style performs much better than the loop parallelization directives, possibly because the data distribution is recalculated at each loop and may change from one loop to the other. A directive imposing to keep the same distribution among the threads may be useful : it would avoid some synchronisations. In our mixed MPI/OpenMP implementation we have obtained better results than with the MPI-only code, which shows that OpenMP allows to take advantage of shared memory parallelism in clusters of SMP. In case of parallel algorithms with important volumes of communication a mixed MPI/OpenMP implementation may bring decisive improvements to an MPI version.

References

- [1] Piero Lanucara and Sergio Rovida. Conjugate Gradient Algorithms : An MPI-OpenMP implementation on Distributed Memory Systems. In *Proceedings of the European Workshop on OpenMP, 1999*.
- [2] Dixie Hisley, Gagan Agrawal, Punyam Satya-Narayana and Lori Pollock. Porting and Performance Evaluation of Irregular Codes using

OpenMP. In *Proceedings of the European Workshop on OpenMP*, 1999.

[3] Compaq Alphaserver. <http://www.compaq.com/AlphaServer/es40/es40.html>

[4] <http://www.cs.virginia.edu/mccalpin>

[5] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 1994.