

Conjugate-Gradients Algorithms: An MPI-OpenMP Implementation on Distributed Shared Memory Systems

Piero Lanucara¹ and Sergio Rovida²

¹ CASPUR, c/o Università “La Sapienza”, P. Aldo Moro 2, 00185 Roma, Italy

² Istituto Analisi Numerica - C.N.R., via Ferrata 1, 27100 Pavia, Italy

Abstract

We present a parallel implementation, on distributed shared memory architectures of two Conjugate-Gradients algorithms: the *CGS* and the *BiCGSTAB* methods associated with some algebraic preconditioners.

We analyze the programming environment supplied by an SMP cluster of Digital AlphaServer 4100.

We take into account two different implementations of the algorithms: one based on MPI and the other one based on a mix of MPI calls, for the message passing among different nodes, and OpenMP directives to parallelize the basic linear algebra operations within each node.

This multilevel parallelism can be also obtained with a combination of MPI and the Digital Extended Math Library Pthreads (DXMLP), a multi-threaded version of the BLAS, LAPACK routines, available on the considered architecture.

1 Parallel Implementation

In this work we address a parallel implementation, using the SPMD programming paradigm, of the *CGS* and *BiCGSTAB* methods [1] and of some algebraic preconditioners based on diagonal scalings and incomplete factorizations.

The SPMD programming model implies focusing on the data associated with the problem, determining an appropriate partition and working out how to associate computation with data. Several approaches are plausible, but we restricted our attention to partitioning the linear system by rows, choosing *block-cyclic* or *cyclic (k)* data distribution [2].

The basic computational kernels of the considered iterative schemes are: vector updates, inner products and matrix-vector products.

Vector updates are trivially parallelizable: each processor updates its own segment of each vector. The inner

products can be easily parallelized: each processor computes a local inner product (LIP) using the segments of each vector which are stored in its local memory; the LIPs travel across processors and are summed together in order to be reduced to the required inner product.

The parallelization of matrix-vector products is related to the structure of the matrix, because each processor has only a segment of the vector in its memory. Thus, communication may be necessary to get other elements of the vector, resulting in global or local message passing operations.

The software, which was first implemented on a Cray-T3D [3] and further tested on the HP-Convex Exemplar systems at CILEA [4], has been ported on the cluster of Digital AlphaServer at CASPUR in Rome.

This cluster consists in four AlphaServer 3/400 4100 nodes [5] interconnected with Memory Channel. Each node is a symmetric multiprocessing machine (SMP) with four CPUs and 8 Gigabytes of memory. Each CPU is an Alpha 21164 (400 Mhz) microprocessor, endowed with a primary 8-Kbytes instruction cache and two levels of data caches (8-Kbytes and 96-Kbytes respectively). Memory Channel Technology [6] is a high-performance network that implements a form of shared virtual memory. The current implementation consists of a 100-megabyte-per-second bus that provides a write-only path from a page of virtual address space of one node to a page of physical memory on another node. A Memory Channel cluster consists of one PCI adapter on each node and a hub connecting the nodes.

For portability issues we implemented the code employing the MPI to manage interprocessor communication and relying on the standard BLAS library to perform linear algebra operations.

This choice allowed us to exploit a combination of explicit message passing and shared memory parallelism using the Digital Extended Math Library Pthreads (DXMLP) [7]. This library, which includes BLAS and some LAPACK routines, is a set of mathematical subprograms optimized for Digital architectures, running in parallel on the single SMP node using Pthreads.

Table 1: Scaled speedup, fixed speedup, number of iteration, total execution time and time per iteration (*sec*) for a fixed size problem ($n = 3008$), varying the number p of processors.

p	<i>scaled speedup</i>	<i>fixed speedup</i>	<i>total time</i>	<i>time per iteration</i>	<i>number of iterations</i>
1	1.00	1.00	79.22	0.56	142
2	1.99	1.67	47.32	0.33	142
3	2.98	2.16	36.76	0.27	133
4	3.88	2.62	30.26	0.21	147
5	4.71	3.40	23.32	0.17	139
6	5.79	3.95	20.05	0.14	140
7	6.28	3.85	20.59	0.14	146
8	7.36	4.95	15.99	0.12	134
10	8.41	6.29	12.60	0.09	135
12	9.56	7.90	10.03	0.07	137
14	9.86	10.52	7.53	0.05	140
16	9.38	13.09	6.05	0.04	144

Such multilevel parallelism can be also achieved by means of Guide from the KAP/Pro Toolset [8], by exploiting the parallelism available on SMP machines. Through the use of directives, Guide translates a program running on one processor into one that runs simultaneously on multiple processors. It uses the standard OpenMP directives on all supported UNIX and NT systems, so that portability is guaranteed. For codes that are already parallel, Guide also translate common Cray and SGI directives to OpenMP directives. The Guide features include Control Parallelism (*parallel region*, *parallel do*, *parallel section*) and Storage Parallelism (*shared and private data specifications*).

2 Experimental results

We analyze the performance obtained on a test problem using the finite element Wathen matrix [9]. It represents the $n \times n$ consistency mass matrix for a regular nx by ny grid of 8-nodes element in 2 space dimension, where $n = 3nx ny + 2nx + 2ny + 1$. For the sake of conciseness, we discuss only the results of the unpreconditioned *BiCGSTAB* algorithm.

Table 1 shows the results obtained for the MPI version of the code, on a fixed size problem, varying the number of processor p .

A satisfactory scalability is achieved for $p \leq 8$; when all the available nodes of the cluster are involved in the computation ($p \geq 10$), the results are not so good, due to some conflicts in accessing the Memory Channel.

Tables 2–5 display the performance of a small number of multi-threaded MPI processes. We report the times per iteration (*sec*) of the DXLMP and Guide versions of the code.

Table 2: Comparison between the DXMLP and Guide versions of the code, increasing the number of threads, for a number $proc = 1$ of MPI processes.

	$proc = 1$	
<i>threads</i>	<i>DXMLP</i>	<i>GUIDE</i>
1	0.55	0.57
2	0.33	0.37
3	0.28	0.29
4	0.20	0.25

Table 3: Comparison between the DXMLP and Guide versions of the code, increasing the number of threads, for a number $proc = 2$ of MPI processes.

	$proc = 2$	
<i>threads</i>	<i>DXMLP</i>	<i>GUIDE</i>
1	0.55	0.57
2	0.33	0.37
3	0.28	0.29
4	0.20	0.25

Table 4: Comparison between the DXMLP and Guide versions of the code, increasing the number of threads, for a number $\text{proc} = 3$ of MPI processes.

	$\text{proc} = 3$	
threads	DXMLP	GUIDE
1	0.20	0.22
2	0.11	0.13
3	0.08	0.08
4	0.06	0.07

Table 5: Comparison between the DXMLP and Guide versions of the code, increasing the number of threads, for a number $\text{proc} = 4$ of MPI processes.

	$\text{proc} = 4$	
threads	DXMLP	GUIDE
1	0.20	0.22
2	0.11	0.13
3	0.08	0.08
4	0.06	0.07

The Guide version of the code was obtained by parallelizing through compiler directives the BLAS routine DGEMV, which computes the matrix-vector products and appears to be the most expensive computational kernel of the procedure. We split the outer loop of the routine in a number of *sections* to be executed in parallel.

The BLAS routine DAXPY was used within each section and the global matrix–vector product was achieved by summing in parallel the partial DAXPY contributions.

Table 6 displays a comparison between global MPI and mixed version of the code for a fixed number (4, 8, 12, 16) of cpus, varying the number of MPI processes and threads.

For the considered test problem the multithreaded version of the code doesn’t show an improvement in the performance with respect to the MPI version. However a satisfactory scalability is obtained using OpenMP directives.

A mixed (OpenMP-MPI or DXMLP-MPI) implementation could be more efficient for a larger problems giving poor scalability for a high number of processors.

With respect to the achieved results we remark that the MPI-Guide version of our software is easy to implement and the performance is quite satisfactory; moreover the code can be ported on other SMP architectures (Guide is a standard OpenMP) and the MPI structure of the software doesn’t change, so that it still works on non SMP platforms which support MPI.

It seems that a de-facto standard for the future architectures will be the cluster of SMP machines interconnected with faster links, each SMP limited in the number of cpu’s for scalability reasons. So, a great effort in developing software able to run efficiently on such machines has to be done, together with the porting of existing codes. In our opinion the mixed model MPI-OpenMP is the only way to portability and efficiency on such architectures, and this is shown in this paper for a kernel code of a very popular and useful iterative solvers. Future improvements in the parallelization, together with new and more difficult test programs will be treated from us in the next months.

Table 6: Total execution time and time per iteration (*sec*) for a fixed number of cpus, varying the number of MPI processes and threads.

4 cpus		<i>time per iteration</i>	<i>number of iterations</i>	<i>total time</i>
<i>4 MPI processes</i>		0.205	147	30.26
<i>1 MPI process 4 threads</i>	DXMLP	0.202	142	28.79
	GUIDE	0.251	154	38.79
<i>2 MPI processes 2 threads</i>	DXMLP	0.168	142	23.88
	GUIDE	0.185	138	25.57
<i>4 MPI processes 1 thread</i>	DXMLP	0.202	147	29.79
	GUIDE	0.222	139	30.80
8 cpus		<i>time per iteration</i>	<i>number of iterations</i>	<i>total time</i>
<i>8 MPI processes</i>		0.119	134	15.99
<i>2 MPI processes 4 threads</i>	DXMLP	0.116	153	17.75
	GUIDE	0.132	138	18.22
<i>4 MPI processes 2 threads</i>	DXMLP	0.112	135	15.20
	GUIDE	0.132	139	18.32
<i>8 MPI processes 1 thread</i>	DXMLP	0.120	140	16.92
	GUIDE	0.118	134	15.79
12 cpus		<i>time per iteration</i>	<i>number of iterations</i>	<i>total time</i>
<i>12 MPI processes</i>		0.072	137	10.03
<i>3 MPI processes 4 threads</i>	DXMLP	0.077	143	11.08
	GUIDE	0.092	140	12.92
<i>4 MPI processes 3 threads</i>	DXMLP	0.076	147	11.27
	GUIDE	0.077	139	10.76
<i>12 MPI processes 1 thread</i>	DXMLP	0.071	137	9.87
	GUIDE	0.065	138	9.08
16 cpus		<i>time per iteration</i>	<i>number of iterations</i>	<i>total time</i>
<i>16 MPI processes</i>		0.041	144	6.05
<i>4 MPI processes 4 threads</i>	DXMLP	0.064	147	9.47
	GUIDE	0.070	139	9.83
<i>8 MPI processes 2 threads</i>	DXMLP	0.058	140	8.22
	GUIDE	0.071	134	9.52
<i>16 MPI processes 1 thread</i>	DXMLP	0.043	145	6.41
	GUIDE	0.041	140	5.93

References

- [1] Saad, Y.: Iterative methods for sparse linear systems. PWS Publishing Company (1966)
- [2] Banerjee, P., Chandy, J. A., Hodges IV, E. W., Holm, J. G., Lain, A., Palermo, D. J., Ramaswamy, S., Su, E.: The Paradigm Compiler for Distributed-Memory Multicomputers. Computer (October 1995) 37-47
- [3] Fornasari, N., Rovida, S.: Conjugate-Gradients Algorithms on a Cray-T3D. Science and Supercomputing at CINECA, 1995 Report, (1995) 496-506
- [4] Fornasari, N., Rovida, S.: An Analysis of the Implementation on Different Parallel Architectures of Preconditioned Conjugate-Gradients Algorithms, IAN report 1015 (1995) 1-16
- [5] <http://www.europe.digital.com/alphaserver/alphasrv4100/as4000.html>
- [6] Gillett, R.: MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect, IEEE Micro, (February 1996) 12-18
- [7] <http://www.digital.com/hpc/software/dxml.html>
- [8] <http://www.kai.com/kpts/guide/features.html>
- [9] Higham, N. J.: The Test Matrix Toolbox for MATLAB (Version 3.0), Numerical Analysis Report No. 276 (September 1995) 1-70