A Parallel Algorithm for the Solution of the Deconvolution Problem on Heterogeneous Networks*†

Pedro Alonso, Antonio M. Vidal
Dpto. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/N, 46022 Valencia (Spain)
palonso@dsic.upv.es

Alexey L. Lastovetsky
School of Computer Science and Informatics
University College of Dublin
Belfield, Dublin 4 (Ireland)
alexey.lastovetsky@ucd.ie

Abstract

In this work we present a parallel algorithm for the solution of a least squares problem with structured matrices*. This problem arises in many applications mainly related to digital signal processing. The parallel algorithm is designed to speed up the sequential one on heterogeneous networks of computers. The parallel algorithm follows the HeHo strategy (Heterogeneous distribution of processes over processors with homogeneous distribution of computations over the processes) and is implemented using HeteroMPI, a recently developed extension of MPI for programming high performance computations on heterogeneous networks of computers. The obtained results validate HeteroMPI as a very useful tool for portable implementation of parallel algorithms for heterogeneous environments.*

1. Introduction

The *mpC* is a programming language for writing parallel programs for Heterogeneous Networks of Computers (HNOCs) [13]. mpC is an extension to the C[] language, which is a Fortran 90 like extension to ANSI C supporting array–based computations. mpC provides the programmer with a useful means for description of the performance model of the implemented parallel algorithm. The programming system uses this description to optimally map (at runtime) this algorithm to the computers of the executing network. The mpC programming system employs an advanced performance model of a heterogeneous network of computers (HNOC). As a result, mpC allows the programmer to write an efficient program for heterogeneous networks in a portable form. The program will be automatically tuned at runtime to each executing HNOC trying to run on the network with the maximal possible speed.

Heterogeneous Message Passing Interface (HeteroMPI) is an extension of MPI obtained by applying the mpC parallel programming model to the message–passing library [14]. Actually, HeteroMPI is an adaptation of mpC language to the MPI programming level. The main idea of HeteroMPI is to automate and optimize the selection of a group of processes that will execute the parallel algorithm faster than any other possible group. For this purpose, HeteroMPI provides a small and dedicated definition language for specification of the performance model of the algorithm. This language is a subset of mpC. In particular, HeteroMPI can be used to port a homogeneous parallel algorithm to heterogeneous environments by allowing the application programmer to specify the performance model of the homogeneous algorithm and create a group of processes executing the algorithm on each HNOC with maximal speed. It is supposed that in this case the HeteroMPI program employs the multiple–processes–per–processor configuration and tries to find the optimal number of processes on each processor to be involved in the execution of the algorithm.

In this work, a version of the HeteroMPI software freely available from the UCD Heterogeneous Computing Laboratory was used [17]. The main goal of this paper is to present an example of the use of HeteroMPI to develop an efficient and portable heterogeneous parallel program based on a homogeneous algorithm. The key part of the work is the development of an accurate performance model of the homogeneous parallel algorithm. Once it has been build, other important parameters of the algorithm that affect its performance such as the block size and the number of processes have to be correctly chosen to obtain the maximum possible performance whatever HNOC is used.

The example application deals with a particular issue of digital signal processing called *inverse filtering of multi-*

*Funded by Secretaría de Estado de Universidades e Investigación del Ministerio de Educación y Cultura of Spain.

†Supported by Spanish MCYT and FEDER under Grant TIC 2003-08238-C02-02.

1-4244-0328-6/06/$20.00

channel systems. However, the solution of the underlying mathematical problem covers a wider field of related digital signal analysis problems. Furthermore, the acquired experience can be generalised to a larger set of homogeneous parallel triangularization algorithms on heterogeneous networks using HeteroMPI.

The following section describes very briefly the inverse filtering problem. Section 3 explains the mathematical background and the sequential algorithm. Section 4 presents the programming model used to implement the parallel algorithm. Section 5 describes the performance model of the algorithm used by HeteroMPI to map processes to the HNOC. A short description of the parallel algorithm is given in Section 6 and some experimental results are presented in Section 7. A conclusions section closes the paper.

2. Inverse Filtering of Multichannel Systems

Inverse filtering and equalisation of multichannel systems is a field of growing interest. This fact is mainly due to the upcoming applications of multichannel systems such as digital communication (mainly new generation digital mobile communications that incorporates array processing at the base stations) and the modern multichannel audio reproduction systems such as three–dimensional (3–D) audio [12], or active noise control, and the availability of new technology resources which make possible the implementation of more complex signal processing algorithms.

The mathematical model of inverse filtering and equalisation multichannel systems are standing for large–scale matrix problems with structure. The major challenge in this area is to design fast and numerically reliable algorithms for large–scale structured linear matrix equations and the least squares matrix problem. We formulate the mathematical problem of inverse multichannel deconvolution systems [10] as

$$\min_x \|Mx - b\| \ , \tag{1}$$

where $M \in \mathbb{R}^{m\times n}$ is a Toeplitz–block matrix with each block being a special class of Toeplitz matrices arisen in convolution operations and polynomial multiplications.

Several algorithms have been traditionally used to solve the linear least squares problem of Toeplitz–like matrices exploiting its special structure to get a computational cost an order of magnitude lower than other classical algorithms for non–structured matrices. These are the well–known *fast* algorithms [11, 9, 16, 15, 8]. In this paper we used the normal equations associated to the least squares problem (1)

$$M^T M x = M^T b \ , \tag{2}$$

from which the seminormal equations

$$L^T L x = L^T b \ ,$$

are computed by making the triangular decomposition of $M^T M$ so the solution x is obtained by solving two triangular systems.

To develop the heterogeneous parallel algorithm we start from a homogeneous parallel algorithm for the solution of the inverse filtering multichannel systems presented in [2] but modified to use a more efficient approach based on the concept of *Cauchy–like* matrices. The use of Cauchy–like matrices to design efficient parallel algorithms for the solution of standard numerical linear algebra problems such as the linear systems solution or the minimisation of the least squares problem with structured matrices has been successfully applied in the complex, hermitian, real and symmetric real cases [3, 4, 7, 1, 5].

Let S be a Block Discrete Sine Transformation that applied to equation (2) and taking into account that $S = S^T$ and $SS^T = S^T S = I$ we have

$$(SM^T MS)Sx = Sb \rightarrow C\hat{x} = \hat{b} \ ,$$

where C is called a *Cauchy–like* matrix.

3. Parallel triangularization of symmetric Cauchy–like matrices

Cauchy–like matrices, accordingly we use them, are implicitly known by means of the generator pair (G, H) and the diagonal displacement matrices Λ as it can be seen in its displacement representation

$$\Lambda C - C\Lambda = GHG^T \ . \tag{3}$$

Generator $G \in \mathbb{R}^{n\times r}$ and Λ are computed by means of $O(n \log n)$ operations from the analog displacement representation of the Block–Toeplitz matrix $M^T M$ that give rise to the linear system to solve (2). However, the algorithm for the solution of a symmetric Cauchy–like system $Cx = b$ is independent of how C has been formed. Cauchy–like matrix is not explicitly formed in either case. Cauchy–like matrix is implicitly known by means of the generator pair (G, H) and the displacement matrix Λ (3). Algorithms to solve this linear system are called "fast" just because they work on the $n \times r$ entries of G instead of the $n \times n$ entries of C, being fulfilled $r \ll n$ for structured matrices.

For the parallel triangularization of C,

$$C = LDL^T \ , \tag{4}$$

being L unit lower triangular and D diagonal, we note that the triangularization process operations can be carried out *independently on each row* of G.

The BLACS distribution model used to manage logically distributed arrays highly helps to distribute the factors involved in the parallel process. A simply "logical column" of p processes denoted by P_i, $i = 0, \ldots, p-1$, is sufficient.

The generator G is partitioned into blocks of size $nb \times r$. The unit lower triangular factor L (4) obtained is partitioned in a two dimensional array of $(n/nb) \times (n/nb)$ square blocks of order nb. These blocks are cyclically distributed among processes so blocks $G_i, L_{i,j}, i, j = 0, \ldots, n/nb-1$, belong to process $P_{i \bmod p}$. For simplicity in the exposition we assume $(n \bmod nb) = 0$ although this condition have not to be accomplished in the implementation. The diagonal matrix D (4) is stored in the diagonal entries of L since all the diagonal entries of L are implicitly one. Fig. 1 shows an example of distribution of both factors G and L in a "logical column" of three processors.

P_0	G_0	$L_{0,0}$					
P_1	G_1	$L_{1,0}$	$L_{1,1}$				
P_2	G_2	$L_{2,0}$	$L_{2,1}$	$L_{2,2}$			
P_0	G_3	$L_{3,0}$	$L_{3,1}$	$L_{3,2}$	$L_{3,3}$		
P_1	G_4	$L_{4,0}$	$L_{4,1}$	$L_{4,2}$	$L_{4,3}$	$L_{4,4}$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 1. Example of data distribution ($p = 3$).

The value nb chosen has a great impact in the parallel algorithm. High values of nb produce low number of messages of big size but the load is unbalanced in this case. Low values of nb produces higher number of messages with lower size with better load balancing. Furthermore, this parameter highly depends on the hardware platform so it must to be chosen by experimental tuning.

The block triangular factorization is a finite iterative process that computes all blocks of one column of L in each iteration. The operation of the parallel algorithm can be easily seen in Fig. 2 by following the ordering numbers in the computed blocks of L. No more detailed explanation is needed but the amount of computations and communications and the order of these operations to understand the performance model described in the next sections. A more detailed description of triangularization of symmetric Cauchy–like matrices can be found, i.e. [7].

The following is the main piece of code of subroutine `pdtrf` that implements this process.

```
 1
 2 call blacs_gridinfo( ictxt, nprow, npcol,
 3                              myrow, mycol )
 4
 5 if( nprow.eq.1 ) then
 6
 7   call dtrfx( n, r, G, lld, L, lld )
 8   return
 9
10 end if
11
12 do k = 1, n-nb, nb
13
14   call infog1l( k, nb, nprow, myrow,
15                         rsrc, lk, pkrow )
16
17   if( myrow.eq.pkrow ) then
18
19     call dtrfx( nb, r, G( lk ), lld,
20                       L( lk+k*lld ), lld )
21     call dlacpy( 'A', nb, r,
22                    G( lk ), lld, V, nb )
23
24     call dgebs2d( ictxt, 'C', ' ',
25                            nb, r, V, nb )
26
27   else
28
29     call dgebr2d( ictxt, 'C', ' ', nb, r,
30                           V, nb, pkrow, 0 )
31
32   end if
33
34   kk = k+nb
35   call infog1l( kk, nb, nprow, myrow,
36                         rsrc, lk, pkrow )
37   np = numroc( n-kk+1, nb, myrow,
38                              pkrow, nprow )
39   call dupdx( np, r, nb, G( lk ), lld,
40               V, nb, L( lk+k*lld ), lld )
41
42 end do
43
44 call infog1l( k, nb, nprow, myrow,
45                         rsrc, lk, pkrow )
```

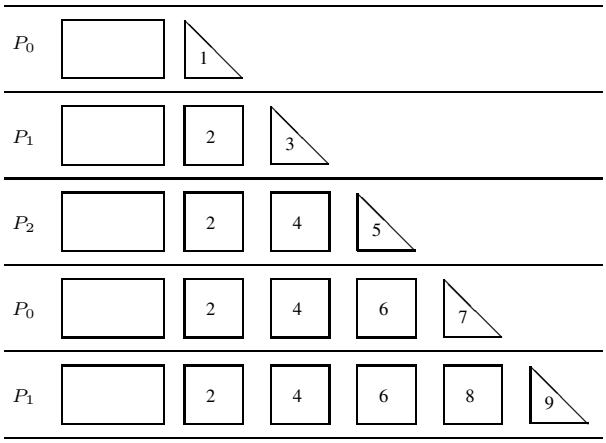

Figure 2. Order of the computation of the blocks that form the triangular factor L by means of the `pdtrf` subroutine.

```
46
47 if( myrow.eq.pkrow ) then
48
49   np = numroc( n-kk+1, nb, myrow,
50                            pkrow, nprow )
51   call dtrfx( np, r, G( lk ), lld,
52                       L( lk ), lld )
53
54 end if
```

Subroutine `pdtrf` calls to `blacs_gridinfo` (line 2) in order to obtain the coordinates of the calling process (`myrow` and `mycol`) together with the shape of the logical grid (`nprow`×`npcol`) on which the parallel algorithm runs, for a given context manager number `ictxt`. Next, if there is only one process in the network (line 5) subroutine `dtrfx` is called to solve the problem sequentially (line 7). `dtrfx` receives a generator `G` of size `n`×`r` and returns the triangular factor `L` of order `n`. Sequential subroutines mentioned here and advance are block routines and it can be found a further description in [6].

The general case when the number of processes is greater than one is solved by means of the loop between lines 12 to 42. A triangular block and the rectangular block down is computed in each iteration, that is, iteration `k` computes blocks marked as $2k-1$ and $2k$ (Fig. 2). The implementation of the algorithm uses the style of BLACS and ScaLAPACK routines as it can be seen by the use of the `infog1l` routine (line 14). Given a global index `k` of the distributed matrix `G`, the block size `nb`, the number of processes in one dimension `nprow`, the coordinate of the calling process in this dimension `myrow` and the coordinate of the source process that has the first element of the distributed array `rsrc`, this routine returns the index of the global index `k` in the local memory `lk` and the coordinate of the owner processor (`pkrow`) of the global `k`th element. The process owner of element `k` (lines 17–27) is in charge to compute the triangular block whereas the others only have to wait for data from process `pkrow`. Process `pkrow` computes the triangular factor calling `dtrfx`, stores the `nb`×`r` rectangular piece of the generator `G` used to compute de triangular factor in a different place of memory (`V`) (line 21) and broadcasts `V` to the rest of processes (line 24). The rest of processes receive `V` (line 29). BLACS routines `dgebs2d` and `dgebr2d` are used to perform this intercommunication operation.

The rest of the code until the end of the loop computes the square blocks down the actual triangular block that has just been computed by means of routine `dupdx` (lines 34 and 40). In lines 44–54 is computed a non–square block if it exists, that is, if $(n \bmod nb) \neq 0$.

This triangularization processes represents between 80% and 90% of the overall cost.

4. The HeHo strategy

The HeHo strategy for design of parallel algorithms for HNOCs uses a *He*terogeneous distribution of processes over processors and *Ho*mogeneous block distribution of data over the processes. This is carried out by mapping different number of processes to the physical processors according to their performance. The tool that allows to do that is HeteroMPI. Specifically, HeteroMPI is a programming environment that allows for porting a homogeneous parallel algorithm based on calls to BLACS/ScaLAPACK routines to a heterogeneous environment without changing the algorithm. However, this step is not easy due to a very accurate performance model of the algorithm is needed to be written in order the mapping runtime system to map the processes in the best way to achieve the maximum performance.

The cornerstone of the connection between a SPMD program consisting of several MPI processes and the program itself running on a HNOC is the *Performance Model*. The Performance Model is based on the notion of *network* as it is used in the context of the *mpC* language introduced by A. Lastovetsky [13]. A *network* corresponds to a group of processes jointly performing some parallel computations. A mpC network is an abstraction facilitating the work with actual processes of the parallel program. Firstly, the programmer must define a *network* consisting of a given number of abstract processors, and then describe the parallel computations on this *network*. Abstract processors representing the *network* will be mapped to real processors of the physical HNOC according to the performance description of the behaviour of the parallel algorithm. Therefore, the Performance Model must define the mpC *network* with sufficient detail so that the mapping algorithm can correctly map the program processes (represented by the abstract processors of the *network*) to suitable real processors to achieve the maximum performance.

HeteroMPI is an adaptation of the mpC language to the MPI programming level. HeteroMPI automates the selection of a group of processes that executes the heterogeneous algorithm faster than any other group. The algorithms used to solve the problem of process selection are essentially the same as those used in the mpC compiler. HeteroMPI introduces new routines for creating groups of processes. During the creation of a group of processes with the routines, the HeteroMPI runtime system solves the problem of selecting the optimal set of processes running on different computers of the heterogeneous networks. Summarizing, HeteroMPI helps automatically find the optimal configuration of MPI applications on a heterogeneous network by choosing the suitable number of processes that will run on each real processor according to its physical features and the defined performance model.

5 The Performance Model

The following code corresponds to the performance model of our parallel algorithm.

```
1 double cost(int n, int nb, int p, int I);
2
3 nettype Performance_model(int n, int r,
4                    int nb, int nbb, int p)
5 {
6    coord I=p;
7    node {
8       I>=0:
9         bench * cost( n, nb, p, I ) *
10          ( nb*nb / (double) (nbb*nbb) );
11   };
12   link ( J=p )
13   {
14      I>=0 && J!=I:
15        length * ( (nb*r)*(n/(nb*p)+
16            (n%nb?1:0))*sizeof(double) )
17              [I] -> [J];
18   };
19   parent [0];
20   scheme
21   {
22     int P, k, i, j;
23     double cc, cl;
24     int nblks = n/nb;
25     double propor;
26
27     cl = (n/(nb*p)+(n%nb?1:0));
28     nblks = nblks + ((n%nb)!=0);
29     for( k = 0; k < nblks; k++ ) {
30       P = k%p;
31       cc = cost( P, p, nb, n );
32       propor = 1.0;
33       if( k==nblks-1 ) {
34         propor = n%nb / (double) nb;
35       }
36       (100.00 * propor * propor / cc)
37                                 %% [ P ];
38       par(i = 0; i < p; i++) {
39         if( i!=P ) {
40           (100.00/cl) %% [ P ] -> [ i ];
41         }
42       }
43       par( i = 0; i < p; i++ ) {
44         cc = cost( i, p, nb, n );
45         for( j = k+1; j < nblks; j++ ) {
46           if( j%p == i ) {
47              propor = 1.0;
48              if( j==nblks-1 ) {
49                propor = n%nb/(double)nb;
50              }
51              (2.0*100.00*propor/cc)
52                                 %% [ i ];
53           }
54         }
55       }
56     }
57   };
58 };
59
60 double cost(int n, int nb, int p, int I)
61 {
62   double c=0.0;
63   double propor;
64   int nblocks, i;
65
66   nblocks = n/nb;
67   for( i = 0; i < nblocks; i++ ) {
68     if( (i%p)==I ) {
69       c = c + 2.0*i + 1.0;
70     }
71   }
72   propor = n%nb / (double) nb;
73   if( propor>0.0 && (nblocks%p)==I ) {
74     c += propor * ( 2.0 * i + propor );
75   }
76   return c;
77 }
```

The first item in the network definition corresponds to the association of the abstract processors with a coordinate system (line 6). Each abstract processor in the network is identified by an integer `I` representing its coordinate in a line of `p` processors ranging from `0` to `p-1`.

Lines 7–11 describe the total amount of computation performed by each of the abstract processors (`I>=0`). The runtime mapping algorithm used in the HeteroMPI environment performs a benchmark operation, whose time is represented by variable `bench`. This benchmark corresponds to the computation of a triangular block of order `nbb` by means of `dtrfx`. The value `nbb` is supposed to be specified in the main program and passed to the description network. The same is true for other problem parameters used in this program. The time returned in the `bench` variable is used to estimated the real computational cost of the `I`th abstract processor. Due to the complexity of the analytical formulae that describe the total amount of computational cost performed by each process, we have used the function `cost` to compute this amount of computation (lines 60–77).

The first loop of `cost` counts the number of square blocks multiplied by 2 plus the number of triangular blocks of the triangular factor belonging to each processor. Both type of blocks are of order `nb`. Fig. 3 shows a distribution example for a problem of size $n = 27$ with three processors and a block size of 5. The first loop of function `cost` returns 8 when calling by P_0, that is, 3 squares blocks $\times$ 2 plus 2 triangular blocks $\times$ 1. Also returns 12 and 5 for P_1 and P_2, respectively. The rest of `cost` returns the proportional part of an incomplete row of blocks as it happens in

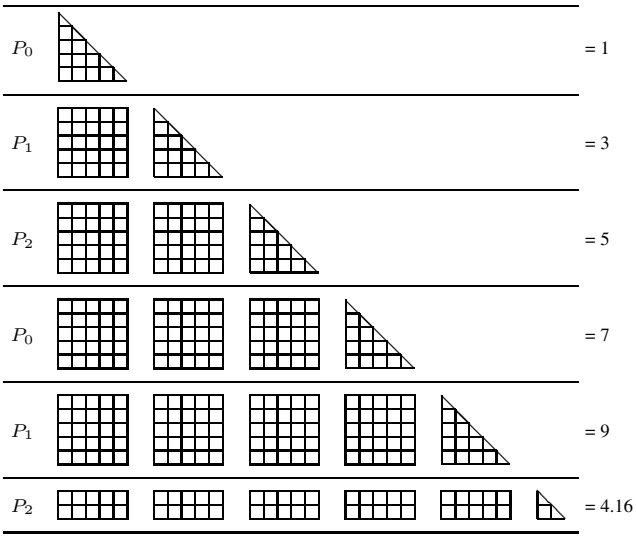

Figure 3. Example of operation of the `cost` function.

the example. This last part returns 4.16 when is called by P_2 and 0 otherwise. Thus, function `cost` returns 8, 12 and 9.16 when calling by P_0, P_1 and P_2, respectively.

Summarising, the total amount of computation performed by an abstract processor and described in lines 9 and 10 is the result of the product of the benchmark time for computing a triangular block of size `nbb` (`bench`), the number of times that an abstract processor performs a computation equivalent to the computation of a triangular block of size `nb` (`cost( n, nb, p, I )`), and the proportion between asymptotic cost of a `nb` block regarding a `nbb` block.

The next section of the performance model describes the total communication cost (lines 12–18). This cost corresponds to a broadcast per iteration of blocks of `nb`×`r` double precision scalars. Lines 15–17 show the physical link performance returned by the runtime environment (`length`) multiplied by the total amount of data sent from processor `I` to processor `J`.

Until line 19 we described the total computational (`node`) and communication (`link`) costs of the algorithm. However, as our preliminary results have shown, this description is not enough for the mapping algorithm to distribute the workload optimally. The next part of the performance model shows the behaviour of the parallel algorithm in terms of the order and the cost of different operations performed during its execution.

The `scheme` description for this case is given for the main loop of the algorithm. Each iteration of this loop corresponds to the computation of a triangular block. Processor `P` is the owner of the triangular block computed at iteration `k`. Lines 36–37 are the percentage of the total computational cost computed by processor `P` when computing only this triangular block. Lines 38–42 describe the communication percentage corresponding to the broadcast of a rectangular block of size `nb`×`r`. It is described by means of a parallel loop because one-to-all communication operations are supposed to be implemented as a concurrent combination of one-to-one communications. Lines 43–55 describe the computation of the squares blocks below the computed triangular one at the `k`th iteration. These blocks are computed concurrently by each processor so the loop indexed by variable `i` is a parallel structure. Inner loop (`j`) goes over each of these squares and only computes the ones owned by the calling processor. Lines 51–52 show the percentage of the total amount of computation performed by processor `i` during the computation of the square block `j`.

6. Implementation of the Parallel Algorithm

The HeteroMPI provides with the facility of using a homogeneous model of computation in the real running environment of a HNOC. This is done by means of performing certain operations before and after calling the main driver routine that solves the problem (`pdtrf`). We will show the main operations through the description of the main program.

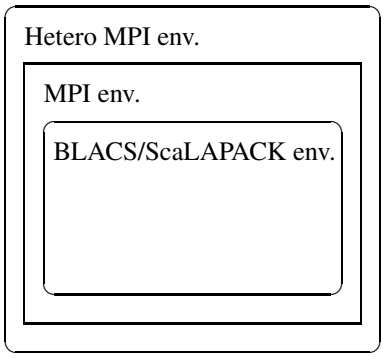

Figure 4. Structure of the parallel algorithm through the different programming environments.

Fig. 4 shows the interaction between different environments used. First of all, the main routine (written in C) works in the HeteroMPI environment. The first operation deals with the performance of the underlying physical processors, which are going to execute the parallel algorithm. All starting processes belong to the predefined communication universe known as `HMPI_COMM_WORLD_GROUP`, and call the `HMPI_Recon` routine. This routine performs a call to a benchmark routine chosen by the programmer. In our case, the benchmark routine is `dtrfx`, that is, the routine that allows to obtain a triangular block of size `nbb`×`r`,

Table 1. Main characteristics of the processors of the Heterogeneous cluster.

Name (number of processors)	Architecture	cpu MHz.	Total Main Memory (mBytes)	Cache (kBytes)	Relative speed dtrfx
pg1cluster02 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512	375
csultraXX (1)	SunOS 5.8 sun4 sparc SUNW,Ultra-5_10	440	512	2048	95

where nbb is a parameter that can be passed to the main program or can be of a fixed size, and r is the number of columns of the generator. The number nbb is chosen to be large enough so that the time spent by the benchmark function can be used to accurately estimate the relative performance of the underlying physical processors, and, at the same time, to be as small as possible in order to minimize the involved overhead. This choice is important because it has a significant impact on the accuracy of the mapping.

The main driver routine, pdtrfx, depends on several parameters. These parameters are not only parameters defined by the problem itself such as the generator size (n×r), but also some other additional parameters such as the block size (nb) and the number of processes. This fact means that these latter parameters would be tuned before calling the parallel routine in order to execute it with the best values and to free the user from their selection, which would force them to have a profound knowledge of the parallel algorithm and the environment.

The next step in the parallel algorithm deals with the tuning of the number of processes and the block size nb. Two nested loops indexed by attempted values perform this computation by calling to the HMPI_Timeof routine. This routine estimates the execution time of the parallel algorithm without its real execution. The execution time of HMPI_Timeof itself is negligible. HMPI_Timeof uses the information provided by the performance model. After the execution of the nested loop, the best values of the parameters are found and the parallel algorithm will work with this choice.

Really, the HeteroMPI environment setup starts next. This setup consists of similar steps to the MPI environment setup. The host processor (the *parent* processor under the mpC terminology) calls routine HMPI_Group_create with the suitable arguments to create a HeteroMPI work group of abstract processes. The number of processes (p) are the ones chosen by the previous tuning algorithm. The rest of processes call the same routine as well. This must be done in this way since HMPI_Group_create is a collective operation. The HeteroMPI group will be composed by only these p processes so the rest of the processes must exit the application calling HMPI_Finalize.

After these HeteroMPI operations, the MPI environment setup takes place in order to have the possibility of running the homogeneous routine as it is shown in Fig. 4. The link between HeteroMPI and MPI environments is made up by a call to the HeteroMPI routine

```
mpicomm = *(MPI_Comm*)HMPI_Get_comm(&gid);
```

This routine allows to obtain a MPI communicator from a HeteroMPI group of processes (mpicomm) identified in the example with the variable gid. This is the only step representing the MPI environment due to the rest of the parallel algorithm uses the BLACS model.

At the next step the BLACS environment is set up as usual when programming for homogeneous NOCs. The connection between the MPI and BLACS code is performed by

```
ictxt = Csys2blacs_handle(mpicomm);
```

where ictxt represents the BLACS context. Once the context identifier is obtained, the following is typical BLACS/ScaLAPACK code started by the initialisation of the logical BLACS grid.

Each of the environments finishes by calling the appropriate closing routines such as gridexit for BLACS or HMPI_Group_free and HMPI_Finalize for HeteroMPI.

7. Experimental Results

The experiments are carried out on a cluster of seven interconnected computers, one of which is a two–processor Linux workstation, pg1cluster02, and the others are identical uniprocessor Sun workstations called csultra01, csultra02, csultra03, csultra04, csultra05 and csultra06, respectively. Table 1 shows specification of the computers, including their relative speed measured with the core computation of our algorithm, dtrfx. One can see that for our algorithm a processor of pg1cluster02 is ≈ 4 times faster than that of Sun workstations.

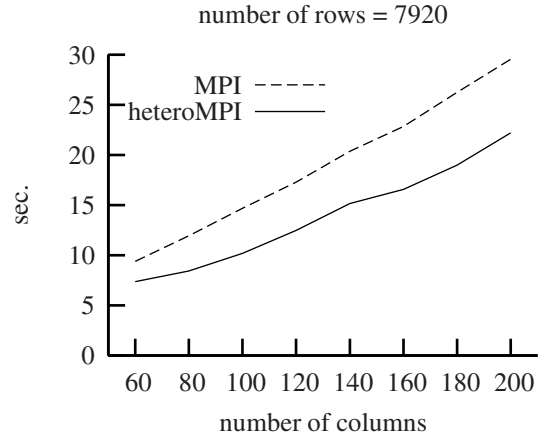

Figure 5. Execution time with fixed number of rows varying the number of columns.

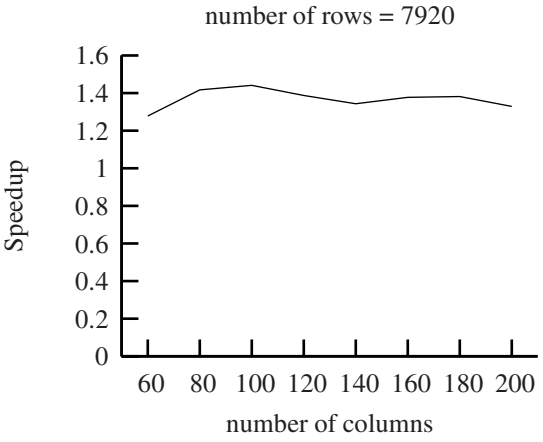

Figure 6. Speedup obtained using HeteroMPI over MPI with fixed number of rows varying the number of columns.

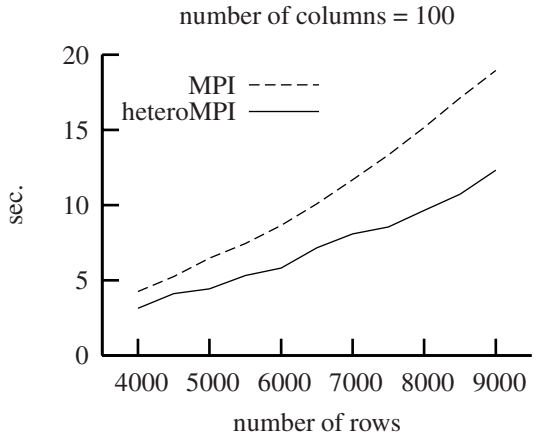

Figure 7. Execution time with fixed number of columns varying the number of rows.

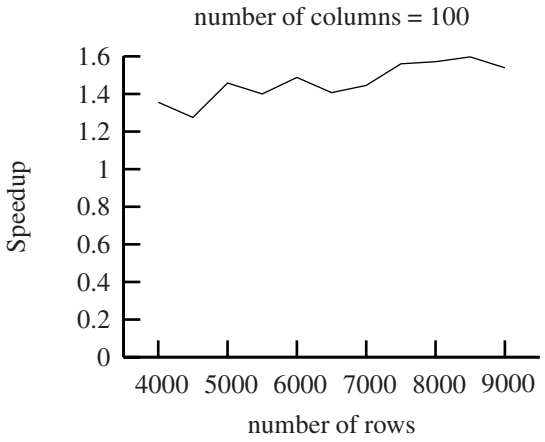

Figure 8. Speedup obtained using HeteroMPI over MPI with fixed number of columns varying the number of rows.

As it has been shown in the previous section, our application finds the optimal number of processes and the optimal block size before a call to the main routine executing the core computations and communications. In our experiments, 14 processes were always chosen to perform the main routine and a block size varied in a range of $14 - 24$.

Fig. 5 shows the execution time of the HeteroMPI application and the standard MPI application using a fixed number of rows ($n = 7920$) and varying the number of columns. Fig. 6 shows the speedup as a ratio between the execution time of the standard MPI application and the HeteroMPI application of Fig. 5. The speedup is about 1.4 in this case. Similar results have been obtained with a fixed number of columns of 100 and varying the number of rows. In this last case, it can be seen that the difference in time between the standard MPI application and the HeteroMPI application grows with the size of the problem (Fig. 7) since the cost of the problem grows quadratically with the number of rows. This can be better seen in Fig. 8. This fact also explains a higher speedup (1.6) than in the other case.

Both type of results show how with a detailed and well designed Performance Model it can be achieved a heterogeneous parallel algorithm that exploits the power of computation of each of the different computers of the heterogeneous network.

8. Conclusions

In this paper, we have presented our experience with a real application on a HNOC using HeteroMPI. HeteroMPI is an extension of MPI for programming high–performance computations for HNOCs. HeteroMPI automates creation of a group of processes which would execute the heterogeneous algorithm faster than any other group. HeteroMPI provides some features that allows the user to write efficient heterogeneous algorithms.

One of the most interesting advantages is the ability to Efficiently implement legacy homogeneous algorithms for heterogeneous environments without any change in the source code. However, the performance of the application strongly depends on the accuracy of the Performance Model designed by the application programmers to describe their implemented algorithms. HeteroMPI provides comprehensive features to express many scientific parallel applications such as the one presented in this paper. Other features deal with the use of `HMPI_Recon` and `HMPI_Timeof` routines. The first one allows the application programmer to express the computational core of the application and is used by the mapping algorithm to make the best on its work. The accuracy of `HMPI_Recon` depends on how accurately the benchmark code provided by the programmer reflects the core computations of each phase of the algorithm. Routine `HMPI_Timeof` allows for tuning the application parameters by estimating the execution time at runtime. The accuracy of `HMPI_Timeof` depends on both the accuracy of the Performance Model and `HMPI_Recon`.

Other experiments with this tool have been recently carried out by the developers of the HeteroMPI. Our contribution demonstrates even more the utility of this tool providing with a real application widely used in digital–signal analysis with a wide range of real applications like 3D sound reproduction systems. In addition, we have explored the behaviour of an irregular parallel heterogeneous algorithm with the particular feature of a very low cost derived from the use of structured matrices like it is the case of Toeplitz–Block and Block–Cauchy–like matrices. As our results confirm, even with these type of low cost algorithms it can be achieved a good speedup exploiting the aggregate power of a HNOC by written an accurate Performance Model of the application.

References

[1] P. Alonso, , and A. M. Vidal. An efficient parallel solution of complex toeplitz linear systems. In *Proceedeings of the Sixth International Conference On Parallel Processing and Applied Mathmatics*, Poznan, Poland, Sept. 2005.

[2] P. Alonso, J. M. Badía, A. González, and A. M. Vidal. Parallel design of multichannel inverse filters for audio reproduction. In *Parallel and Distributed Computing and Systems, IASTED*, volume II, pages 719–724, Marina del Rey, CA, USA, Nov. 2003.

[3] P. Alonso, J. M. Badía, and A. M. Vidal. Parallel algorithms for the solution of toeplitz systems of linear equations. *Lecture Notes in Computer Science*, 3019:969–976, 2004.

[4] P. Alonso, J. M. Badía, and A. M. Vidal. An efficient and stable parallel solution for non–symmetric Toeplitz linear systems. *Lecture Notes in Computer Science*, 3402:685–692, 2005.

[5] P. Alonso, M. O. Bernabeu, and A. M. Vidal. A parallel solution of hermitian toeplitz linear systems. In *Computational Science – ICCS 2006*, volume 3991 of *LNCS*, pages 348–355. Springer, 2006.

[6] P. Alonso, A. Lastovetsky, and A. M. Vidal. A parallel algorithm for the solution of the deconvolution problem in heterogeneous networks. Technical Report 2006–2, School of Computer Science and Informatics. University College of Dublin, 2006.

[7] P. Alonso and A. M. Vidal. The symmetric–toeplitz linear system problem in parallel. *LNCS*, 3514:220–228, 2005.

[8] A. W. Bojanczyk, R. P. Brent, and F. R. de Hoog. QR factorization of Toeplitz matrices. *Numerische Mathematik*, 49(1):81–94, July 1986.

[9] A. González and J. López. Two steps levinson algorithm for time domain multichannel deconvolution. *Electronic Letters*, 36(7):686–688, 2000.

[10] A. González and J. López. Fast transversal filters for deconvolution in multichannel sound reproduction. *IEEE Trans. on Speech and Audio Processing*, 9 (4):429–440, May 2001.

[11] H. Irisawa, S. Shimada, H. Hokari, and S. Hosoya. Study of a fast method to calculate inverse filters. *J. Audio Eng. Soc.*, 46(7/8):611–619, 1998.

[12] C. Kyriakakis, P. Tsakalides, and T. Holman. Surrounded by sound. *IEEE Signal Processing Magazine*, 16 (1):55–66, 1999.

[13] A. Lastovetsky. *Parallel Computing on Heterogeneous Networks*. John Wiley & Sons, NJ, USA, 2003.

[14] A. Lastovetsky and R. Reddy. HeteroMPI: Towards a Message–Passing Library for Heterogeneous Networks of Computers. *Journal of Parallel and Distributed Computing, Elsevier*, 66(2):197–220, 2006.

[15] J. López, A. Gónzalez, and F. Orduña Bustamante. Equalization zones for cross talk cancellation as a function of loudspeaker position and room acoustics. In *proc. de ACTIVE 99*, Ford Lauderdale, Florida, Dec. 1999.

[16] J. López, A. González, and F. Ordua-Bustamante. Measurement of cross-talk cancellation and equalization zones in 3-d sound reproduction under real listening conditions. In *proc. de AES 16th International Conference on Spatial Sound Reproduction*, Rovaniemi, Finlandia, 1999.

[17] R. Reddy and A. Lastovetsky. HeteroMPI: Programmer's and installation guide. Technical report, School of Computer Science and Informatics, University College of Dublin, 2005.