Experiments with mpC: Efficient Solving Regular Problems on Heterogeneous Networks of Computers via Irregularization

Dmitry Arapov, Alexey Kalinov, Alexey Lastovetsky, and Ilya Ledovskih

Institute for System Programming, Russian Academy of Sciences
25, Bolshaya Kommunisticheskaya str., Moscow 109004, Russia
`mpc@ispras.ru`

Abstract. mpC is a medium-level parallel language for programming heterogeneous networks of computers. It allows to write libraries of parallel routines adaptable to peculiarities of any particular executing multiprocessor system to ensure efficient running. The adaptable routines distribute data and computations in accordance with performances of participating processors. In this case even the problems traditionally considered regular, become irregular. Advantages of mpC for efficient solving of regular problems on heterogeneous networks of computers are demonstrated with an mpC routine implementing Cholesky factorization, with efficiency of the mpC routine being compared with ScaLAPACK one.

1 Introduction

A heterogeneous network of computers, being the most common parallel architecture available to common users, can be used for high-performance computing. Taking into account that in the 1990s network capacity increases surpassed processor speed increases [1], pp.6–7, one can predict increasing their importance as a low-cost platform for parallel high-performance computations. Efficient programming heterogeneous networks of computers has some difficulties which do not arise when programming traditional homogeneous supercomputers. The point is that to use the full performance potential of a heterogenous network of computers, it is necessary to distribute data and computations among processors in accordance with their performances. That heterogeneity of data distribution leads to considering irregular such problems as, for example, dense linear algebra problems, traditionally considered regular when solving on homogeneous multiprocessor computing systems.

One can ascertain absence of suitable and handy tools for parallel programming irregular applications for heterogeneous networks of computers. Both high-level parallel languages like HPF [2] and low-level message-passing packages like MPI [3] are not suitable for the purpose, since they do not have facilities to detect performance characteristics of a particular executing multiprocessor hardware and distribute data and computations among its processors in accordance with results of the detection.

The situation has induced us to develop mpC – a medium-level parallel language for programming heterogeneous networks of computers, that, like the C language, combines assembler (MPI) flexibility and efficiency with high-level programming language convenience. The mpC language is an ANSI C extension allowing to write libraries of parallel routines adaptable to peculiarities of any particular executing parallel computer system to ensure efficient running.

We will demonstrate advantages of mpC with such a generally-known challenge in parallel computations as Cholesky factorization. We use almost the same parallel algorithm, that is used in ScaLAPACK [4] in case of 1-D processor grid, and use LAPACK [5] and BLAS [6] for local computations. The main difference between our parallel algorithm and the ScaLAPACK one is data distribution. We consider the Cholesky factorization to be an irregular problem and distribute data among processors of an executing parallel machine in accordance with their relative performances. ScaLAPACK considers the problem regular and distributes data in accordance with homogeneous block-cyclic distribution. Of course, we make no pretensions to solve all problems but only want to demonstrate how mpC allows slightly to modify a good parallel algorithm to obtain an adaptable routine for heterogeneous networks of computers.

Our parallel mpC function will hide its parallel nature from a caller. It corresponds to an easy-to-use style of parallel programming, when all parallel computations are encapsulated in library functions. This approach makes transition from sequential programming in C to parallel programming in mpC very simple.

Section 2 introduces a parallel algorithm of Cholesky factorization. Section 3 describes implementation of the algorithm in mpC, introducing all necessary details of the mpC language. Section 4 compares the mpC and ScaLAPACK Cholesky factorization routines running on networks of workstations.

More about mpC can be found in [7–10] as well as at mpC home page (http://www.ispras.ru/~mpc), where additionally free mpC software is available.

2 Algorithm of Cholesky Factorization in mpC

Cholesky factorization of a real symmetric positive definite matrix is an extremely important computation, arising in a variety of scientific and engineering applications. It is a well-known challenge for efficient and scalable parallel implementation because of large volumes of interprocessor communications.

Cholesky factorization factors an $n \times n$, symmetric, positive-definite matrix A into a product of a lower triangular matrix L and its transpose, i.e., $A = LL^{\mathrm{T}}$. One can partition the nxn matrices A, L and L^{T} and write the system as

$$\begin{bmatrix} A_{11} & A_{21}^{\mathrm{T}} \\ A_{21} & A_{2}2 \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{2}2 \end{bmatrix} \cdot \begin{bmatrix} L_{11}^{\mathrm{T}} & L_{21}^{\mathrm{T}} \\ 0 & L_{2}^{\mathrm{T}}2 \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^{\mathrm{T}} & L_{11}L_{21}^{\mathrm{T}} \\ L_{21}L_{11}^{\mathrm{T}} & L_{21}L_{21}^{\mathrm{T}} + L_{22}L_{22}^{\mathrm{T}} \end{bmatrix},$$

where blocks $A11$ and $L11$ are $n_{b1} \times n_{b1}$, A_{21}, L_{21} are $(n - n_{b1}) \times n_{b1}$, A22, L22 are $(n - n_{b1}) \times (n - n_{b1})$. $L11$ and $L22$ is lower triangular, n_{b1} is the size of the first block. Assuming that L_{11}, the lower triangular Cholesky factor of A_{11}, is known, one can rearrange the block equations

$$L_{21} \leftarrow A_{21} \left(L_{11}^{\mathrm{T}}\right)^{-1} ,$$
$$\tilde{A}_{22} \leftarrow \left(A_{22} - L_{21} L_{21}^{\mathrm{T}} = L_{22} L_{22}^{\mathrm{T}}\right) .$$

The factorization can be done by recursively applying the step outlined above to the updated matrix A_{22}. The parallel implementation of the corresponding ScaLAPACK routine `PDPOTRF` [11] is based on the above scheme and a block cyclic distribution of matrix A over a $P \times Q$ process grid with a block size of $n_b \times n_b$. The routine assumes that the lower (upper) triangular portion of A is stored in the lower (upper) triangle of a two-dimensional array and that the computed elements of L overwrite the given elements of A (here and henceforth when speaking of an array we mean a Fortran array, that is, that column elements of an 2-D array are allocated contiguously).

Our mpC Cholesky factorization routine implements almost the same algorithm that is implemented by the ScaLAPACK one in the case of 1-D process grid (P=1) and the lower triangular portion of A used to compute L. Namely, to compute above steps it involves the following operations:

1. process Pr, which has L_{11}, L_{21}, calls LAPACK function `dpotf2` to compute Cholesky factor L_{11} and sets a flag if A_{11} is not positively defined;
2. process Pr calls BLAS function `dtrsm` to compute Cholesky factor L_{21} if A_{11} is positively defined;
3. process Pr broadcasts the column panel, L_{11} and L_{21}, as well as the flag to all other processes and stops the computation if A_{11} is not positively defined;
4. all processes stop the computation if A_{11} is not positively defined or otherwise update matrix A_{22} in parallel, that involves calls to BLAS functions `dsyrk` and `dgemm` by each process updating its local portions of matrix A_{22}.

The main difference between the mpC and the ScaLAPACK routines lies in data distribution. In fact, in our case (P=1) the ScaLAPACK routine divides matrix A into a number of column panels with just the same width n_b and distributes them cyclically over Q processes (see Fig. 1) where Q, nb are input parameters of the routine.

The mpC routine distribution is almost the same. The only difference is that column panel with width n_{bi}, calculated as follows

$$n_{bi} = Q \cdot n_b \frac{p_i}{\sum_{j=1}^{Q} p_j} \quad , \tag{1}$$

is placed to i-th process, where Q, n_b are input parameters of the routine, and p_j are the relative speed of j-th process ($j = 1, \ldots, Q$).

Suppose we have a 18×18 matrix, $P = 1$, $Q = 2$, $n_b = 3$ and the underlying network of computers consists of two processors, each running one process and the second being twice faster. Figure 1 shows the ScaLAPACK data distribution and figure 2 shows the mpC data distribution. In this case $n_{b1} = 2$, $n_{b2} = 4$.

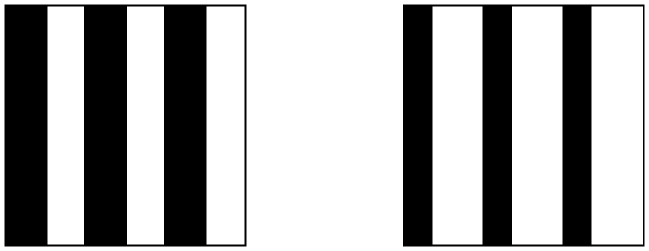

Fig. 1. ScaLAPACK(left) and mpC(right) data distributions when the underlying process grid consists of two processors, the second being twice as fast. Black columns belong to the first process and white columns belong to the second one

3 Implementation of the Cholesky Factorization in mpC

The mpC language is an ANSI C superset allowing the user to specify the topology of and to define the so-called network objects (in particular, dynamically) as well as to distribute data and computations over the network objects. The mpC programming environment uses this information to map (in run time) the mpC network objects to any underlying heterogeneous network in such a way that to ensures efficient running of the application on the network.

In mpC, a programmer deals with a new kind of resource – *computing space* – a set of virtual processors represented in run-time by actual processes. The resource can be managed with allocating and discarding regions of computing space called *network objects* (or simply *networks*). Allocating network objects in the computing space and discarding them is performed in similar fashion to allocating and discarding data objects in the storage in the C language. A network object may be used to distribute data, to compute expressions, and to execute statements. Every network has a *parent* – the virtual processor initiated its allocation and belonging to both the newly created network and one of networks created before. The only virtual processor defined from the beginning of program execution till its termination is the pre-defined virtual *host-processor*.

So, in our mpC application we, first of all, define a network object over which we want to distribute data and computations. Every network object, declared in an mpC program, has a type. The *network type* specifies the number and performances of virtual processors, links between these processors, as well as separates the parent. For our purpose we declare the family of network types, named `HeteroNet`,

```
/*1.1*/  nettype HeteroNet(n, p[n]) {
/*1.2*/    coord I=n;
/*1.3*/    node { I>=0: p[I]; };
/*1.4*/  };
```

parametrized with integer parameter `n` and vector parameter `p` consisting of `n` integers. The family of network types corresponds to network objects consisting of n virtual processors. The virtual processors are related to the coordinate system with coordinate variable I ranging from 0 to n-1. The relative performance of the virtual processor with coordinate I being characterized by the value of p[I]. By default the network parent has coordinate 0.

The high-level mpC function Ch, implementing the Cholesky factorization, looks as follows

```
/* 2.1*/ void [*]Ch(double *[host]A,
/* 2.2*/            int [host]N,
/* 2.3*/            int *[host]INFO) {
/* 2.4*/   repl int nprocs, * ipowers;
/* 2.5*/   repl double * powers;
/* 2.6*/   MPC_Processors_static_info(&nprocs, &powers);
/* 2.7*/   IntPowers(nprocs,powers,&ipowers);
/* 2.8*/   {
/* 2.9*/     net HeteroNet(nprocs,ipowers) w;
/*2.10*/     double* [w]da;
/*2.11*/     int [w]info=0;
/*2.12*/     repl int [w]n, * [w]map, [w]source[1]= {0};
/*2.13*/     n=N;
/*2.14*/     if(I coordof da == 0)
/*2.15*/       [host]da=A;
/*2.16*/     else
/*2.17*/       da=[w]malloc( n*n*[w](sizeof(double)));
/*2.18*/     map=[w]malloc(n*[w]sizeof(int));
/*2.19*/     [w]Distr(n,[w]nprocs,[w]powers,map);
/*2.20*/     ([([w]nprocs)w]) ChScatter(da,n,map);
/*2.21*/     info=([([w]nprocs)w])ParCh(da,n,map );
/*2.22*/     *INFO=[host]info[+];
/*2.23*/     if(I coordof da != 0) [w]free(da);
/*2.24*/     [w]free(map);
/*2.25*/   }
/*2.26*/   free(ipowers);
/*2.27*/ }
```

The function is a so-called *basic* mpC function with three arguments belonging to the virtual host-processor: pointer A to the source matrix, dimension N of the matrix, and pointer INFO to an indicator of the termination status. There are three kinds of function in mpC: basic, network, and nodal functions. *Basic function* is called and executed on the entire computing space. Only in basic functions networks may be defined. *Network function* is called and executed on a network object. *Nodal function* can be executed completely by any one virtual processor. Any C function is considered a nodal function in mpC.

Line 2.4 defines integer variable nprocs and pointer ipowers to integer. Both variable nprocs and data object, that ipowers points to, are declared *replicated* over the entire computing space. By definition, data object *distributed* over a region of the computing space (in particular, over the entire computing space) comprises a set of components of any one type so that each virtual processor of the region holds one component. By definition, a distributed data object is *replicated* if all its components is equal to each other.

Line 2.5 defines pointer `powers` distributed over the entire computing space and specifies that it points to a replicated data object.

Line 2.6 calls library nodal function `MPC_Processors_static_info` on the entire computing space returning the number of actual processors and their relative performances. So, after this call replicated variable `nprocs` will hold the number of actual processors, and replicated array `powers` will hold their relative performances. Note, that the possibility to detect in run time the detailed information about characteristics of an executing actual parallel machine is an important peculiarity of the mpC language making it a suitable tool for efficient programming heterogeneous networks of computers.

Line 2.7 calls nodal function `IntPower` which allocates and initializes replicated *integer* array `ipowers` holding relative processor performances.

At the point, we have obtained enough information about characteristics of the executing multiprocessor to define properly a network object to perform our parallel Cholesky factorization. Line 2.9 defines automatic network `w`, the type of which, being an instance of the corresponding family of network types, is defined only in run time, and which executes the most of the rest of computations and communications. It consists of `nprocs` virtual processors, the relative performance of the `i`-th virtual processor being characterized by the value of `ipowers[i]`. The definition of the network causes its allocation (or creation) in run time. The mpC programming environment will ensure the optimal mapping of virtual processors of the network `w` into a set of actual processes representing the entire computing space. So, just one process from processes running on each of actual processors will be involved in the Cholesky factorization, and the more powerful is the virtual processor, the more powerful actual processor will execute the corresponding process.

Note, that the possibility to define a network type (or requirements to virtual processors) in run time and to map the virtual processors into actual processes in accordance with the network type requirements is a key advantage of the mpC programming environment making it a suitable tool for programming heterogeneous networks.

Having defined the network, we can distribute data and computations over it. Construct `[w]` in the definition of pointer `da` in line 2.10 just says that the pointer is distributed over network `w`. By default, if the distribution of a variable is not specified, it means that the variable is distributed over:

- the entire computing space if defined in a basic function;
- the corresponding network if defined in a network function.

Construct `[*]` specifies that the data object is distributed over the entire computing space, and construct `[host]` specifies that the data object belongs to the host.

The assignment in line 2.13 broadcasts the value of N to all components of distributed variable n. In general, the simple assignment is extended in mpC to express data transfer between virtual processors of the same network object.

The if-else statement in lines 2.14–2.17 sets a value of distributed pointer da. On the virtual host-processor its component points to matrix A, and on other virtual processors of the network w its components point to an allocated array. Unlike the previous statement, the execution of this statement does not need any communications between virtual processors constituting network w. In fact, this statement is divided into a set of independent undistributed statements each of which is executed by the corresponding virtual processor using the corresponding data components. Such statement is called an *asynchronous* statement.

The control expression in the if-else statement contains unusual binary operator coordof. Its result is an integer value distributed over w, each component of which is equal to the value of coordinate variable I of the virtual processor to which the component belongs. The right operand of the operator coordof is not evaluated and used only to specify a region of the computing space. Note, that coordinate variable I is treated as an integer variable distributed over the region.

Line 2.19 calls nodal function Distr on network w. The execution of this function call just on network w is provided with prefix unary network cast operator [w], cutting from values of Distr, nprocs, and powers just the components, belonging to w, and resulting in all operands of the function call are distributed over network w. A compiler uses the information about distribution of operands of the expression to determine the region of the computing space where the expression is evaluated and the statement is executed.

The function Distr allocates array map and calculates its elements defining the distribution of columns of the matrix over virtual processors of network w. The calculation is based on formula (1). So, after this call, map[i] holds the number of the virtual processor to which the i-th column belongs, and the more powerful is the virtual processor, the more columns are assigned to it.

Line 2.20 calls network function ChScatter on network w to scatter the matrix A in accordance with mapping provided by the array map. The function ChScatter has the following function prototype:

```
/*3.1*/ int [net SimpleNet(p)v]ChScatter(double* da,
/*3.2*/                                  const repl int n,
/*3.3*/                                  const repl int* map);
```

In general, a *network function* is called and executed on some network, and its value is also distributed over the same network. The function ChScatter has two special formal parameters – so-called *network parameter* v, representing the network on which the function is executed, and parameter p, treated in the function as a replicated over network v integer variable. The family of network types, named SimpleNet, is the simplest one introducing only a coordinate system and declared in standard header file mpc.h as follows:

```
/*4.1*/ nettype SimpleNet(n) { coord I=n; };
```

Except the network parameter, no network can be used or declared in the network function. Only data objects belonging to the network parameter may be defined in its body. In addition, the corresponding components of an externally-defined distributed data object can be used. Unlike basic functions, network functions (as well as nodal functions) can be called in parallel. Any network of a relevant type can be used as an actual network parameter in a network-function call. In our case, the network `w` is such a network argument, `nprocs` being other special argument.

Line 2.21 calls network function `ParCh` (described below) to compute in parallel the Cholesky factor of the matrix. It returns the value of flag `info` detecting how computation is terminated on each of virtual processors of network `w`.

Line 2.22 calculates the sum of all components of `info` and assigns the result to `*INFO`. The result of postfix unary operator `[+]` is distributed over `w`. All its components are equal to the sum of all components of its operand `info`. Here, the result of prefix unary network cast operator `[host]` is the component of its operand belonging to the virtual host-processor. So, the statement assigns the sum of all components of `info` to *INFO on the virtual host-processor.

Line 2.23 frees the memory allocated for the matrix on all virtual processors of `w` different from the host-virtual processor.

Network `w` is discarded when execution of the block in lines 2.8–2.25 ends.

Note, that we do not gather result to the host. The algorithm implemented by `ParCh` ensures that Cholesky factor of the matrix will appear on each of virtual processors of network `w`.

The network function strictly computed Cholesky factor of the matrix is the following:

```
/* 5.1*/ int [net SimpleNet(p)w]ParCh(double* da,
/* 5.2*/                            const repl int n,
/* 5.3*/                            const repl int* map) {
/* 5.4*/   repl int ngroups,*displs,*ncols,group,k;
/* 5.5*/   repl int coor,displsg,ncolsg;
/* 5.6*/   int i,j,dim,info=0,displ,coor_l,displsk,ncolsk;
/* 5.7*/   double one=1.0, minus_one=-1.0;
/* 5.8*/   MakeGroups(n, map, &ngroups, &ncols, &displs);
/* 5.9*/   for (group=0;group<ngroups;group++) {
/*5.10*/     displsg=displs[group];
/*5.11*/     ncolsg=ncols[group];
/*5.12*/     coor=map[displsg];
/*5.13*/     if ( coor == (I coordof da)) {
/*5.14*/       /* calculate L11,                    */
/*5.15*/       /* call of the LAPACK function dpotf2 */
/*5.16*/       dpotf2_("L",&ncolsg,da+displsg*(n+1),&n,&info);
/*5.17*/       if(!info && group < ngroups-1) {
/*5.18*/         /* calculate L21,                       */
/*5.19*/         /* call of the BLAS lev.3 function dtrsm */
/*5.20*/         dim=n-displsg-ncolsg;
/*5.21*/         dtrsm_("R","L","T","N",
/*5.22*/                &dim,&ncolsg,&one,
/*5.23*/                da+displsg*(n+1),&n,
/*5.24*/                da+displsg*(n+1)+ncolsg,&n);
/*5.25*/       }
/*5.26*/     }
/*5.27*/     ([([w]p)w])ChBcast
/*5.28*/                  (&info,coor,group,da,displs,ncols,n);
/*5.29*/     if((repl int)info) return info;
/*5.30*/     for(k=group+1; k<ngroups; k++) {
/*5.31*/       int dim_n;
/*5.32*/       displsk=displs[k];
/*5.33*/       ncolsk=ncols[k];
/*5.34*/       coor_l=map[displsk];
/*5.35*/       if ( coor_l == (I coordof da)) {
/*5.36*/         /* update the triangle part of the group */
/*5.37*/         /* call the BLAS level 3 function dsyrk */
/*5.38*/         dsyrk_("L","N",&ncolsk,&ncolsg,
/*5.39*/                &minus_one,da+displsg*n+displsk,&n,
/*5.40*/                &one,da+displsk*(n+1),&n);
/*5.41*/         /* update the rectangle part of the group */
/*5.42*/         /* call the BLAS level 3 function dgemm */
/*5.43*/         dim_n=n-displsk-ncolsk;
/*5.44*/         if(dim_n != 0) {
/*5.45*/           dgemm_("N","T",&dim_n,&ncolsk,&ncolsg,
/*5.46*/                  &minus_one,da+displsg*n+displsk+ncolsk,
/*5.47*/                  &n,da+displsg*n+displsk,&n,
/*5.48*/                  &one,da+displsk*(n+1)+ncolsk,&n);
/*5.49*/         }
/*5.50*/       }
/*5.51*/     }
/*5.52*/   }
/*5.53*/   return 0;
/*5.54*/ }
```

Line 5.8 calls nodal function `MakeGroups` to collect columns in groups. After the call, variable `ngroups` holds the total number of groups, `ncols[i]` holds the number of columns in `i`-th group, and `displs[i]` holds its displacement. In the above example (presented in Fig. 2), there are 6 groups: each of three groups, belonging to the slower virtual processor, consists of two columns, and each of three groups, belonging to the faster virtual processor, consists of four columns.

Lines 5.9–5.52 are the main loop of the algorithm.

Lines 5.10–5.12 calculate the number of columns in the current group as well as the coordinate of the virtual processor holding it.

The statement in lines 5.13–5.26 computes matrix L_{11} by call to LAPACK function `dpotf2` and, if it has been successfully computed, and it is not the latest group, compute matrix L_{21} by call to BLAS level 3 function `dtrsm`. The computations are performed sequentially by the virtual processor holding the corresponding group.

Lines 5.27–5.28 calls network function `ChBcast` to broadcast flag `info` and matrix L_{21} over network `w`.

Line 5.29 returns control and the value of `info` to a caller, if matrix L_{11} has not been successfully computed. The value of control expression must be replicated, since all virtual processors must either return or not return control to the caller coordinately, and it is strictly checked by the compiler. This requirement is an example of the programming style, supported by mpC, which allows to avoid many errors which may arise in parallel programming. In this case, if we do not say to compiler, that we guarantee replication of the value, it will detect an error.

The loop in lines 5.30–5.51 updates matrix A_{22} in parallel, the triangle part of the group is updated by BLAS level 3 function `dsyrk`, and the rectangle part of the group is updated by BLAS level 3 function `dgemm`.

We have presented the most interesting part of about 200 lines of mpC code implementing Cholesky factorization. One can see that it is not very difficult to implement such a complex application in mpC, and as we demonstrate below, our implementation is good enough. It consists of basic library function `Ch` and a small set of network and nodal functions. We have demonstrated two levels of modularity in mpC. The first level of modularity is provided by basic function `Ch`, which creates a network and calls the network function `ParCh` on it, providing the second level of modularity. One can see that our network function does not depend on data distribution (1-D processor grid is assumed only) provided by array `map`, and can be called from different basic functions on different networks as well as from different network functions.

4 Experimental Results

We compared the running time of our `ParCh` mpC function and its ScaLAPACK counterpart `PDPOTRF`. We used SPARCstation 20 (hostname `alpha`), three SPARCstations 5 (hostnames `gamma`, `beta`, and `delta`), and SPARCclassic (`omega`), with relative performances 180, 160, 160, 160, and 77 correspondingly connected via 10Mbits Ethernet. We used the following networks: network `gbo` consisting of `gamma`, `beta`, and `omega`; network `gbd` consisting of `gamma`, `beta`, and `delta`, and so on. Note, that performances of processors in these networks are detected automatically with a command of the mpC programming environment.

We used MPICH version 1.0.13 as a communication platform, GNU C compiler with optimization option -O2, and GNU fortran 77 compiler with optimization option -O4. We started one process per workstation for both programs and tuned n_b to provide the best performance for the mpC and ScaLAPACK routines.

Tables 1 and 2 demonstrate speedups computed relative to the LAPACK routine `dpotf2` executing sequential Cholesky factorization on `gamma`. One can see that the mpC function **ParCh** and the ScaLAPACK routine **PDPOTRF** take approximately the same time when running on homogeneous networks **gb**, **gbd**, and practically homogeneous **gbda** (Table 1). After we enhance these networks with low-performance **omega**, the mpC program allows to utilize the parallel potential of performance-heterogeneous **gbo**, **gbdo**, and **gbdao** speeding up the Cholesky factorization. At the same time, its ScaLAPACK counterpart does not allow this, slowing down the Cholesky factorization (Table 2).

Table 1. Speedups on homogeneous networks

n	gb		gbd		gbda	
	mpC	ScaL	mpC	ScaL	mpc	ScaL
300	1.03	1.03	1.25	1.27	1.26	1.25
400	1.13	1.10	1.47	1.42	1.57	1.60
500	1.18	1.17	1.56	1.54	1.76	1.73
600	1.27	1.24	1.69	1.65	1.91	1.90
700	1.29	1.26	1.75	1.75	2.01	2.03
800	1.33	1.32	1.82	1.84	2.11	2.12

Table 2. Speedups on heterogeneous networks

n	gbo		gbdo		gbdao	
	mpC	ScaL	mpC	ScaL	mpc	ScaL
300	1.15	0.85	1.25	1.02	1.33	1.11
400	1.29	0.98	1.50	1.17	1.62	1.37
500	1.38	1.04	1.64	1.29	1.84	1.43
600	1.48	1.09	1.76	1.36	1.97	1.60
700	1.53	1.13	1.85	1.41	2.11	1.69
800	1.57	1.12	1.90	1.49	2.20	1.77

5 Summary

Efficient programming heterogeneous networks of computers implies distributing data and computations over processors in accordance with their performances. It makes even regular problems be considered irregular. We have demonstrated how mpC can be used for solving such irregular problems. The key facilities making the mpC language and its programming environment unique tools for programming such irregular applications are:

– facilities to detect the number and performances of processors of the executing network of computers;

- convenient facilities to formulate requirements on performances of virtual processors constituting the abstract parallel machine (network object) executing computations and communications (facilities to specify a network type in run time);
- convenient and natural facilities to allocate and discard network objects as well as distribute data and computations over them.

References

[1] El-Rewini, H., and Lewis, T.: Introduction To Distributed Computing. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[2] High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.1. Rice University, Houston TX, November 10, 1994

[3] Message Passing Interface Forum, MPI: A Message-passing Interface Standard, version 1.1, June 1995.

[4] Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., and Whaley, D.: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers – Design Issues and Performance. UT, CS-95-283, March 1995.

[5] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, S., Octrouchov, S., and Sorensen, D.: LAPACK Users' Guide, Second Edition. SIAM, Philadelphia, PA, 1995.

[6] Dongarra, J., Du Croz, J., Duff, I., and Hammarling, S.: A Set of Level 3 Basic Linear Algebra Subprograms. ASM Trans. Math. Soft., 16, 1, pp.1–17, March 1990

[7] Lastovetsky, A.: The mpC Programming Language Specification. Technical Report, ISPRAS, Moscow, December 1994.

[8] Arapov, D., Kalinov, A., and Lastovetsky, A.: Managing the Computing Space in the mpC Compiler. Proceedings of the 1996 Parallel Architectures and Compilation Techniques (PACT'96) conference, IEEE CS Press, Boston, MA, Oct. 1996, pp.150–155.

[9] Arapov, D., Kalinov, A., and Lastovetsky, A.: Resource Management in the mpC Programming Environment. Proceedings of the 30th Hawaii International Conference on System Sciences (HICSS'30), IEEE CS Press, Maui, HI, January 1997.

[10] Arapov, D., Kalinov, A., Lastovetsky, A., Ledovskih, I., and Lewis, T.: A Programming Environment for Heterogeneous Distributed Memory Machines. Proceedings of the 1997 Heterogeneous Computing Workshop (HCW'97) of the 11th International Parallel Processing Symposium (IPPS'97), IEEE CS Press, Geneva, Switzerland, April 1997, pp.32–45.

[11] Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., and Whaley, R.C.: The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. UT, CS-94-246, September, 1994.