Parallel Computing on Heterogeneous Networks: Challenges and Responses

Alexey Lastovetsky

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

E-mail: Alexey.Lastovetsky@ucd.ie

Abstract

In the paper, we analyse challenges associated with parallel programming for common networks of computers (NoCs) that are, unlike dedicated parallel computer systems, inherently heterogeneous and unreliable. This analysis results in description of main features of an ideal parallel program for NoCs. We also outline some recent parallel programming tools, which try and respond to some of the challenges.

1. Introduction

Local networks of computers (NoCs) are the most common and available parallel architecture. Nowadays not only big businesses and organisations but also practically any medium or small one has several computers interconnected in a local network.

In the most general case, a local network of computers consists of PCs, workstations, shared memory multiprocessor (SMP) servers, and even distributed memory multiprocessor supercomputers and clusters interconnected via mixed network equipment.

At a first glance, this architecture is very similar to the distributed memory multiprocessor (also know as MPP) architecture. Like the latter, it provides a number of processors not sharing global main memory and interconnected via a communication network. Therefore, the most natural model of program for NoCs is also a set of parallel processes, each running on a separate processor and using message passing to communicate with the others. That is, message passing is the basic programming model for this architecture.

Due to the similarity of MPPs and NoCs, it might be expected that NoCs be as widely used for high performance parallel computing as MPPs. In reality, NoCs are practically not used for parallel computing. The main reason, why the huge performance potential of millions NoCs around the world is so poorly utilised, is that parallel programming for NoCs is much more difficult than parallel programming for MPPs.

The point is that unlike MPPs, which are designed and manufactured specifically for high performance parallel computing, a typical NoC is a naturally developed computer system. A NoC is a general-purpose computer system, which is developed incrementally, for a relatively long time. As a result, NoCs are not as nicely regular or balanced for high performance computing as MPPs. On the contrary, irregularity, heterogeneity, and instability are their inherent features differentiating the architecture from the MPP architecture. The very features make parallel programming for NoCs so difficult and challenging.

There are three main sources of the difficulties. The first one is the heterogeneity of processors. Generally speaking, in a NoC, different processors are of the different

architecture.

The second source is the communication network itself, which is typically not designed for high performance parallel computing.

The third source is the multi-user nature of NoCs. A NoC is not a strongly centralized computer system. It consists of relatively autonomous computers, each of which may be used and administered independently by its users. In NoCs, different components are not as strongly integrated and controlled as in MPPs.

In the paper we discuss the sources of difficulties and analyse programming challenges coming from each of the sources. This analysis results in description of main features of an ideal parallel program for NoCs. We also outline some recent parallel programming tools, which try and respond to some of the challenges.

2. Heterogeneity of processors

2.1. Different processor speeds

An immediate implication from the fact that a NoC uses processors of different architectures is that the processors run at different speeds. Let us see what happens when a parallel application, which provides a good performance while running on homogeneous MPPs, runs on the cluster of heterogeneous processors.

A good parallel application for MPPs tries to evenly distribute computations over available processors. This very distribution ensures the maximal speedup on MPPs, which consist of identical processors. On the cluster of processors running at different speeds, faster processors will quickly perform their part of computations and wait for slower ones at points of synchronisation. Therefore, the total time of computations will be determined by the time elapsed on the slowest processor. In other words, when executing parallel applications, which evenly distribute computations among available processors, the heterogeneous cluster is equivalent to a homogeneous cluster that is composed of the same number but the slowest processors.

The following simple experiment, which has been really carried out, corroborates the statement. Two subnetworks of the same local network were used, each consisting of four Sun workstations. The first subnetwork included identical workstations of the same model, and was thus homogeneous. The second one included workstations of three different models. Their relative speeds demonstrated while executing a LAPACK [1] Cholesky factorisation routine were 1.9, 2.8, 2.8, and 7.1. As the slowest workstation (relative performance 1.9) was shared by both clusters, the total power of the heterogeneous cluster was almost twice that of the homogeneous one.

It might be expected that a parallel ScaLAPACK [2] Cholesky solver be executed on the more powerful cluster almost twice as fast as on the weaker one. But in reality, it ran practically at the same speed (~2% speedup for a 1800×1800 dense matrix).

Thus, a good parallel application for a NoC must distribute computations unevenly taking into account the difference in processor speed. The faster processor is, the more computations it must perform. Ideally, the volume of computation performed by a processor should be proportional to its speed.

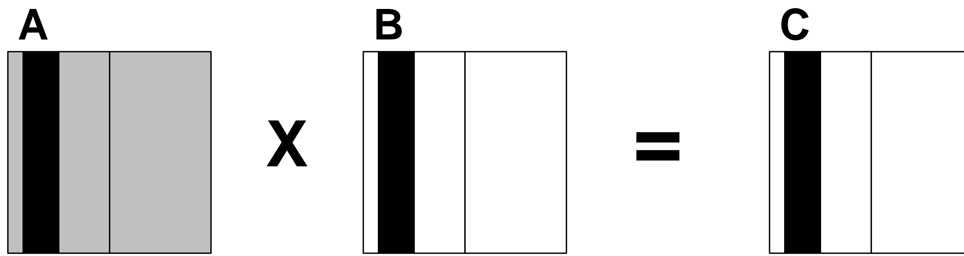

Figure 1. Matrix-matrix multiplication with matrices A, B, and C unevenly partitioned in one dimension. The area of the slice mapped to each processor is proportional to its speed. The slices mapped onto a single processor are shaded black. During execution, this processor requires all of matrix A (shown shaded grey).

For example, a simple parallel algorithm implementing matrix operation $C = A \times B$ on a p-processor heterogeneous cluster, where A, B are dense square $n \times n$ matrices, can be summarized as follows:

- Each element c_{ij} in C is computed as $c_{ij} = \sum_{k=0}^{n-1} a_{ik} \times b_{kj}$.
- The A, B, and C matrices are identically partitioned into p vertical slices. There is one-to-one mapping between these slices and the processors. Each processor is responsible for computing its C slice.
- Because all C elements require the same amount of arithmetic operations, each processor executes an amount of work proportional to the number of elements that are allocated to it, hence, proportional to the area of its slice. Therefore, to balance the load of the processors, the area of the slice mapped to each processor is proportional to its speed (see Fig. 1).
- In order to compute elements of its C slice each processor requires all elements of the A matrix. Therefore, during the execution of the algorithm, each processor receives from p-1 other processors all elements of their slices (shown grey in Fig. 1).

This heterogeneous parallel algorithm cannot be implemented in HPF 1.1 [3], since the latter provides no way to specify a heterogeneous distribution of arrays across abstract processors. But HPF 2.0 [4] addresses the problem by extending `BLOCK` distribution with the ability to explicitly specify the size of each individual block (`GEN_BLOCK` distribution).

For example, the following HPF program implements the above parallel algorithm to multiply two dense square 1000×1000 matrices on a 4-processor heterogeneous cluster, processors of which have relative speeds 2, 3, 5, and 10:

```
        PROGRAM HETEROGENEOUS
        INTEGER, DIMENSION(4), PARAMETER:: M=(/100, 150, 250, 500/)
        REAL, DIMENSION(1000,1000):: A, B, C
!HPF$   PROCESSORS p(4)
!HPF$   DISTRIBUTE (*, GEN_BLOCK(M)) ONTO p:: A, B, C
!HPF$   INDEPENDENT
        DO J=1,1000
!HPF$         INDEPENDENT
              DO I=1,1000
                    A(I,J)=1.0
                    B(I,J)=2.0
              END DO
        END DO
!HPF$   INDEPENDENT
```

```
          DO J=1,1000
!HPF$         INDEPENDENT
              DO I=1,1000
                   C(I,J)=0.0
                   DO K=1,1000
                        C(I,J)=C(I,J)+A(I,K)*B(K,J)
                   END DO
              END DO
          END DO
          END
```

In this program, the "generalized" block distribution, `GEN_BLOCK`, is used to map contiguous segments of arrays `A`, `B`, and `C` of unequal sizes onto processors. The sizes of the segments are specified by values of the user-defined integer mapping array `M`, one value per target processor of the mapping. That is, the *i*-th element of the mapping array specifies the size of the block to be stored on the *i*-th processor of the target processor arrangement `p`. The '*' in the `DISTRIBUTE` directive specifies that array `A`, `B`, and `C` are not to be distributed along the first axis; thus an entire column is to be distributed as one object. So, array elements `A(:,1:100)`, `B(:,1:100)`, and `C(:,1:100)` are mapped on `p(1)`, `A(:,101:250)`, `B(:,101:250)`, and `C(:,101:250)` are mapped on `p(2)`, `A(:,251:500)`, `B(:,251:500)`, and `C(:,251:500)` are mapped on `p(3)`, and `A(:,501:1000)`, `B(:,501:1000)`, and `C(:,501:1000)` are mapped on `p(4)`.

That distribution of matrices *A*, *B*, and *C* across processors ensures that the area of the vertical slice mapped to each processor is proportional to the speed of the processor. Note that this is responsibility of the programmer to explicitly specify the exact distribution of the arrays across processors. The specification is based on the knowledge of both the parallel algorithm and the executing heterogeneous cluster.

HPF 2.0 also allows the programmer to distribute the arrays with the `REDISTRIBUTE` directive, based on a mapping array whose values are computed at runtime. This allows writing a more portable application. But again, either the programmer or a user of the application must explicitly specify the data distribution, which ensures the best performance of this particular parallel algorithm on each particular heterogeneous cluster.

Apparently, the above algorithm can be implemented in MPI [5] as well. The corresponding MPI program will be not as simple as the HPF one because of much lower level of the MPI's programming model. Actually, MPI is a programming tool of the assembler level for message passing programming. Therefore, practically all message passing algorithms can be implemented in MPI.

Whatever programming tool is used to implement the above parallel algorithm, one can see that the efficiency of the corresponding application strongly depends on the accuracy of estimation of the relative speed of processors of the executing heterogeneous cluster. Distribution of arrays and, hence, distribution of computations across the processors are fully determined by the estimation of their relative speed. If this estimation is not accurate enough, the load of processors will be unbalanced, resulting in poorer execution performance.

The problem of accurate estimation of the relative speed of processors is not as easy as it may look. Of course, if you consider two processors, which only differ in clock rate, it is not a problem to accurately estimate their relative speed. The relative speed will be the same for any application.

But if you consider processors of different architectures, the situation changes drastically. Everything in the processors may be different: set of instructions, number of

instruction execution units, number of registers, structure of memory hierarchy, size of each memory level, and so on, and so on. Therefore, the processors may demonstrate different relative speeds for different applications. Moreover, processors of the same architecture but different models or configurations may also demonstrate different relative speeds on different applications.

Even different applications of the same narrow class may be executed by two different processors at significantly different relative speeds. To avoid speculation, consider the following experiment that has been really carried out. Three slightly different implementations of Cholesky factorisation of a 500×500 matrix were used to estimate the relative speed of a SPARCstation-5 and a SPARCstation-20. Code

```
for(k=0; k<500; k++) {
   for(i=k, lkk=sqrt(a[k][k]); i<500; i++)
     a[i][k] /= lkk;
   for(j=k+1; j<500; j++)
     for(i=j; i<500; i++)
       a[i][j] -= a[i][k]*a[j][k];
}
```

estimated their relative speed as 10:9, meanwhile code

```
for(k=0; k<500; k++) {
   for(i=k, lkk=sqrt(a[k][k]); i<500; i++)
     a[i][k] /= lkk;
   for(i=k+1; i<500; i++)
     for(j=i; j<500; j++)
       a[i][j] -= a[k][j]*a[k][i];
}
```

as 10:14. Routine `dptof2` from the LAPACK package, solving the same problem, estimated their relative speed as 10:10.

2.2. Heterogeneity of machine arithmetic

As processors of a NoC may do floating-point arithmetic differently, there are special challenges associated with writing numerical software on NoCs. Specifically, there are two main issues potentially affecting the behaviour of a numerical parallel application running on a heterogeneous NoC.

Firstly, different processors do not guarantee the same storage representation and the same results for operations on floating point numbers.

Secondly, if a floating-point number is communicated between processors, the communication layer does not guarantee the exact transmittal of the floating-point value. Normally, transferring a floating point number in a heterogeneous environment includes two conversions of its binary representation: the representation of the number on the sender site is first converted into a machine independent representation, which is then converted into the representation for floating point numbers on the receiver site. The two successive conversions may change the original value, that is, the value received by the receiver may differ from the value sent by the sender.

To illustrate the potential problems, consider the iterative solution of a system of linear equations where the stopping criterion depends upon the value of some function, *f*, of the relative machine precision, ε. A common definition of the relative machine precision, or unit roundoff, is the smallest positive floating point value, ε, such that $fl(1+\varepsilon) > 1$, where $fl(x)$ is the floating point representation of *x*. The test for convergence might well include a test of the form:

```
if( ‖e_r‖_2 / ‖x_r‖_2 < f(ε) ) goto converged;
```

In a heterogeneous setting the value of *f* may be different on different processors and e_r and x_r may depend upon data of different accuracies, and thus one or more processes may converge in a fewer number of iterations. Indeed, the stopping criterion used by the most accurate processor may never be satisfied if it depends on data computed less accurately by other processors. If the code contains communication between processors within an iteration, it may not complete if one processor converges before the others. In a heterogeneous environment, the only way to guarantee termination is to have one processor make the convergence decision and broadcast that decision.

Another problem is that overflow and underflow exceptions may occur during floating-point representation conversions, resulting in a failure of the communication.

3. *Ad hoc* communication network

One can imagine a local network of heterogeneous computers, whose communication layer is almost as good as the communication layer of the MPP architecture. Parallel programming for such networks called in this book *heterogeneous clusters* faces no specific communication-related challenges. Heterogeneous clusters are normally designed specifically for high performance distributed computing.

At the same time, the topology and structure of the communication network in a typical common local network of computers is determined by many different factors, among which high performance computing is far away from being a primary one if considered at all. The primary factors include the structure of the organisation, the tasks that are solved on computers of the NoC, the security requirements, the construction restrictions, the budget limitations, the qualification of technical personnel, etc.

An additional important factor is that the communication network is constantly developing rather than fixed once and forever. The development is normally occasional and incremental. Therefore, the structure of the communication network reflects the evolution of the organization rather than its current snapshot.

All the factors make the common communication network far away from the ideal MPP communication network, which is homogeneous with communication speedup and bandwidth being balanced with the number and speed of processors.

First of all, the common communication network is heterogeneous. The speed and bandwidth of communication links between different pairs of processors may differ significantly.

Secondly, some of the communication links may be of low speed and/or narrow bandwidth.

This makes the problem of optimal distribution of computations and communications across a NoC much more difficult than across a cluster of heterogeneous processors interconnected with a homogeneous high-performance communication network. The additional difficulty comes from the larger size of the problem, which is now $O(n^2)$, where *n* is the total number of processors (respectively, n^2 is the total number of inter-processor communication links).

Apart from that, due to low performance of some communication links, the optimal distribution of computations and communications may be across some subnetwork of the NoC, not across the entire NoC. This substantially extends the space of possible solutions and increases the complexity of the distribution problem even further.

4. Multi-user decentralised computer system

Unlike MPPs, NoCs are not strongly centralized computer systems. A typical NoC consists of relatively autonomous computers, each of which may be used and administered independently by its users.

4.1. Unstable performance characteristics

The first implication from the multi-user decentralised nature of NoCs is that computers, executing a parallel program, may be also used for other computations and involved in other communications. In that case, the real performance of processors and communication links can dynamically change depending on the external computations and communications.

Therefore, a good parallel program for a NoC must be sensitive to such dynamic variations of its workload. In such a program, computations and communications are distributed across the NoC in accordance to the actual performance at the moment of execution of the program.

4.2. Higher probability of resource failures

Fault tolerance is not a primary problem for parallel applications running on MPPs. The probability of unexpected resource failures in a centralised dedicated parallel computer system is quite small. But this probability reaches much higher figures for NoCs. Firstly, any single computer in a NoC may be switched off or rebooted unexpectedly for other users in the NoC. The same may happen with any other resource in the NoC.

Secondly, not all building elements of the common NoC as well as interaction between different elements are equally reliable.

These make fault tolerance a desirable feature for parallel applications that run on NoCs; and the longer the execution time of the application is, the more important the feature becomes.

The basic programming tool for distributed-memory parallel architectures, MPI, does not address the problem. The point is that a fault-tolerant parallel program assumes a dynamic process model. Failure of one or other process of the program should not necessarily lead to failure of the entire program. The program may continue running even after its set of processes has changed.

The MPI 1.1 process model is fully static. MPI 2.0 does include some support for dynamic process control, although this is limited to the creation of new MPI process groups with separate communicators. These new processes cannot be merged with previously existing communicators to form intracommunicators needed for a seamless single application model and are limited to a special set of extended collective communications.

To date, there is no industrial fault-tolerant implementation of MPI. At the same time, there are a few research versions of MPI suggesting different approaches to the problem of fault-tolerant parallel programming.

The first approach to making MPI applications fault tolerant is through the use of check pointing and roll back. This approach is that all processes of the MPI program will flush their message queues to avoid in flight messages getting lost, and then they will all synchronously checkpoint. At some later stage if any error occurs, the entire MPI program will be rolled back to the last complete checkpoint and be re-started. This approach needs the entire application to checkpoint synchronously, which, depending on the application and its size, may become expensive in terms of time (with potential

scaling problems).

The second approach is to use “spare” processes that are utilized when there is a failure. For example, MPI-FT [6] supports several master-slave models where all communicators are built from grids that contain “spare” processes. To avoid loss of message data between the master and slaves, all messages are copied to an observer process, which can reproduce lost messages in the event of any failures. This system has a high overhead for every message and considerable memory needs for the observer process for long running applications. This system is not a full checkpoint system in that it assumes any data (or state) can be rebuilt using just the knowledge of any passed messages, which might not be the case for non-deterministic unstable solvers.

MPI-FT is an example of an implicit fault tolerant MPI. Such implementations of MPI do not extend MPI interface itself. No specific design is needed for application using an implicit fault tolerant MPI. The system takes full responsibility over fault tolerant features of application. The drawback of that approach is that the programmer cannot control fault tolerant features of the application and fine tune for better balance between fault tolerance and performance as system and application conditions may dictate.

Unlike MPI-FT, FT-MPI [7] is an explicit fault tolerance MPI, which extends standard MPI’s interface and semantics. An application using FT-MPI has to be specifically designed to take advantage of its fault tolerant features.

5. Summary of programming challenges

In summarizing challenges associated with parallel programming for NoCs, let us describe main features of an ideal parallel program running on a NoC.

Such a program distributes computations and communications unevenly across processors and communications links, taking into account their actual performance demonstrated during the execution of the code of the program. The distribution is not static and may be different not only for different NoCs but also for different executions of the program on the same NoC, depending on the workload of its elements. The program may find profitable to involve in computations not all available computers. In other words, the program must be efficiently portable.

The program keeps running even if some resources in the executing network fail. In the case of a resource failure, it is able to reconfigure itself and resume computations from some point in the past.

The program takes into account differences in machine arithmetic on different computers and avoids erroneous behaviour of the program that might be caused by the differences.

6. Any response to the challenges?

Let us see how the challenges are responded. First, we outline how standard parallel programming tools such as HPF and MPI address the highlighted challenges. Then, we briefly introduce mpC, a dedicated programming language designed specifically for parallel computing on heterogeneous networks of computers.

6.1. High Performance Fortran

As we have demonstrated in Section 2.1, HPF provides some basic support for programming heterogeneous algorithms. It allows the programmer to specify uneven distribution of data across abstract HPF processors.

At the same time, it is full responsibility of the programmer to provide a code, which analyses the implemented parallel algorithm and the executing NoC, and calculates the best distribution.

Another problem is that the HPF programmer cannot influence the mapping of abstract HPF processors to computers of the NoC. HPF provides no language constructs allowing the programmer to control better mapping of the heterogeneous algorithms to heterogeneous clusters. The HPF programmer should rely on some default mapping provided by the HPF compiler. The mapping cannot be sensitive to peculiarities of each individual algorithm just because the HPF compiler has no information about the peculiarities. Therefore, to control the mapping and take into account both the peculiarities of the implemented parallel algorithm and the peculiarities of the executing heterogeneous environment, the HPF programmer needs to additionally write a good peace of quite complex code. HPF does not address the problem of fault tolerance at all.

Actually the lack of means for advising the compiler about the features of implemented parallel algorithm that have a major impact on its execution time is the general drawback of HPF, which makes the language difficult for compiling not only for heterogeneous platforms but for MPPs as well.

To illustrate the associated difficulties, consider the following simple HPF program:

```
        PROGRAM SIMPLE
        REAL, DIMENSION(1000,1000):: A, B, C
!HPF$   PROCESSORS p(4,4)
!HPF$   DISTRIBUTE (BLOCK,BLOCK) ONTO p:: A, B, C
!HPF$   INDEPENDENT
        DO J=1,1000
!HPF$         INDEPENDENT
              DO I=1,1000
                    A(I,J)=1.0
                    B(I,J)=2.0
              END DO
        END DO
!HPF$   INDEPENDENT
        DO J=1,1000
!HPF$         INDEPENDENT
              DO I=1,1000
                    C(I,J)=0.0
                    DO K=1,1000
                          C(I,J)=C(I,J)+A(I,K)*B(K,J)
                    END DO
              END DO
        END DO
        END
```

The program implements matrix operation $C = A \times B$ on a 16-processor MPP, where A, B are dense square 1000×1000 matrices. Figure 2 illustrates the implemented parallel algorithm.

The `PROCESSORS` directive specifies a logical 4×4 grid of abstract processors, `p`.

The `DISTRIBUTE` directive recommends the compiler to partition each of the arrays `A`, `B`, and `C` into equal-sized blocks along each of its dimension. This will result in a 4×4 configuration of blocks each containing 250×250 elements, one block per processor. The corresponding blocks of arrays `A`, `B`, and `C` will be mapped to the same abstract processor and, hence, to the same physical processor.

Each of the four `INDEPENDENT` directives in the program is applied to a `DO` loop and advises the compiler that the loop does not carry any dependences and therefore its different iterations may be executed in parallel.

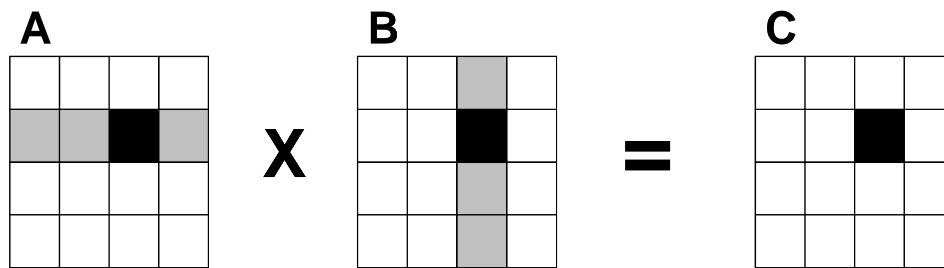

Figure 4.3. Matrix-matrix multiplication with matrices A, B, and C evenly partitioned in two dimensions. The blocks mapped onto a single processor are shaded black. During execution, this processor requires corresponding rows of matrix A and columns of matrix B (shown shaded grey).

Altogether the directives give the compiler enough information in order to generate a target message-passing program. Additional information is given by a general HPF rule saying that evaluation of an expression should be performed on the processor, in the memory of which its result will be stored.

Thus, a clever HPF compiler would be able to generate SPMD message-passing code like that:

```
 PROGRAM SIMPLE
 REAL, DIMENSION(250,250):: A, B, C
 REAL, DIMENSION(250,1000):: Arows, Bcols
 INTEGER colcom, rowcom, col, row
 INTEGER rank, colrank, rowrank
 INTEGER err
 CALL MPI_INIT(ierr)
 CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank);
 row = rank/4
 col = rank-row*4
 DO J=1,250
       DO I=1,250
             A(I,J)=1.0
             B(I,J)=2.0
       END DO
 END DO
 CALL MPI_COMM_SPLIT(MPI_COMM_WORLD, row, rank, rowcom, err)
 CALL MPI_COMM_SPLIT(MPI_COMM_WORLD, col, rank, colcom, err)
 CALL MPI_ALLGATHER(A, 40000, MPI_REAL, Arows, 62500,
&MPI_REAL, rowcom, err)
 CALL MPI_ALLGATHER(B, 40000, MPI_REAL, Bcols, 62500,
&MPI_REAL, colcom, err)
 DO J=1,250
       DO I=1,250
             C(I,J)=0.0
             ind1=1
             ind2=J
             DO K=1,1000
                   C(I,J)=C(I,J)+Arows(I,K)*Bcols(ind1,ind2)
                   IF(ind1.LT.250) THEN
                         ind1=ind1+1
                   ELSE
                         ind1=1
                         ind2=ind2+250
                   END IF
             END DO
       END DO
 END DO
 CALL MPI_COMM_FREE(rowcom, err)
 CALL MPI_COMM_FREE(colcom, err)
 CALL MPI_FINALIZE(err)
```

```
END
```

This code is in Fortran 77 with calls to MPI routines. It is supposed to be executed by all 16 processes making up the parallel program. Each process locally contains one 250×250 block of global arrays `A`, `B`, and `C` of the source HPF program. A logical 4×4 process grid is formed from the 16 participating processes, and each process gets its coordinates `row` and `col` in the grid. In order to compute its block of the resulting matrix *C*, the process needs blocks of matrix *A* from its horizontal neighbours in the 4×4 process grid, and blocks of matrix *B* from its vertical neighbours (see Figure 2). The necessary communication is achieved by calls to the `MPI_COMM_SPLIT` and `MPI_ALLGATHER` routines.

The main specific optimisation performed by an HPF compiler is the minimization of the cost of the inter-processor communication. This is not a trivial problem. It needs profound analysis of both the source code and the executing MPP. HPF provides no specific constructs or directives helping the compiler to solve the problem. This is one of the reasons why HPF is considered a difficult language to compile.

For example, many real HPF compilers (i.e., the ADAPTOR HPF compiler from GMD) will translate the above HPF program into a message-passing program, each process of which sends its blocks of matrices *A* and *B* to *all* other processes. That straightforward communication scheme guarantees that each process receives all the elements of global arrays `A` and `B`, it needs to compute its elements of global array `C`. At the same time, in many particular cases, including ours, this universal scheme involves a good deal of redundant communications, sending and receiving data that are never used in computation. The better a compiler is, the more accurate communication patterns it generates to avoid redundant communications as much as possible. The above message-passing program, generated by an imaginary clever HPF compiler, performs no redundant communication. Each process of the program sends its blocks of matrices *A* and *B* only to 3 other processes, not to 15 as each process of the straightforward program does.

HPF does not address the problem of fault tolerance at all.

6.2. Message Passing Interface

As a general-purpose message-passing tool of assembler level, MPI allows the programmer to write efficiently portable programs for NoCs. At the same time, it provides no specific support to facilitate such programming. It is responsibility of the programmer to write all the code making the application efficiently portable among NoCs. In other words, every time, when programming for NoCs, a programmer must solve the extremely difficult problem of portable efficiency from scratch. Standard MPI also does not address the problem of fault tolerance.

6.3. mpC and HMPI

An original approach to parallel computing on heterogeneous networks that has been proposed and implemented in the framework of the mpC language [8-9] and the HMPI library [10] and their programming systems. In brief, this approach can be summarised as follows:

- The programmer provides the programming system with comprehensive information about the features of the implemented parallel algorithm that have a major impact on the execution time of this algorithm. In other words, the programmer provides a detailed description of the performance model of this algorithm.

- The programming system uses the provided information to optimally map at runtime this algorithm to the computers of the executing network. The quality of this mapping strongly depends on the accuracy of the estimation of the actual performance of the processors and communication links demonstrated at runtime on the execution of this application. Therefore, the mpC programming system employs an advanced performance model of a heterogeneous network of computers, and the mpC language provides constructs that allow the programmer to update the parameters of this model at runtime by tuning them to the code of this particular application.

This approach to parallel computing on heterogeneous networks has proved its efficiency. Many mpC and HMPI applications have been developed that efficiently solve real-life problems on common heterogeneous networks of computers.

The mpC language in its current form addresses all the challenges associated with writing efficiently portable programs for NoCs except for the fault tolerance.

The mpC parallel language allows the programmer to define all main features of the implemented parallel algorithm that can have an impact on the performance of execution of the algorithm on a heterogeneous NoC. The features include the total number of participating parallel processes, the total volume of computations to be performed on each of the processes, the total volume of data to be transferred between each pair of the processes, and how exactly the processes interact during the execution of the algorithm. The mpC programming system uses that performance model of the parallel algorithm together with the model of the executing heterogeneous network to map the processes of the parallel program to this network so as to ensure better execution time. The mapping is executed at runtime; therefore its efficiency is crucial for the total execution performance of mpC applications. The model of a heterogeneous network and the mapping algorithm are developed to keep balance between the accuracy and efficiency.

To briefly introduce the mpC language, consider an mpC application simulating the evolution of groups of bodies under the influence of Newtonian gravitational attraction. Since the magnitude of interaction between bodies falls off rapidly with distance, a single equivalent body may approximate the effect of a large group of bodies. This allows us to solve the problem in parallel. The parallel application will use a few parallel processes, each of which will update data characterizing a single group of bodies. Each process holds attributes of all the bodies constituting the corresponding group as well as masses and centres of gravity of other groups. The attributes characterizing a body include its position, velocity and mass. The application will implement the following parallel algorithm:

```
Initialisation of galaxy on host-process
Scattering groups of bodies over processes
Parallel computing masses of groups
Interchanging the masses among processes
while(1) {
  Visualization of galaxy by host-process
  Parallel computing centers of gravity
  Interchanging the centers among processes
  Parallel updating groups
  Gathering groups on host-process
}
```

It is assumed that at each iteration of the main loop, new coordinates of all bodies in some fixed interval of time are calculated.

The core of the mpC application, implementing the above algorithm, is the following description of the performance model of this algorithm:

```
nettype Galaxy(m, k, n[m]) {
  coord I=m;
  node { I>=0: bench*((n[I]/k)*(n[I]/k)); };
  link { I>0 : length(Body)*n[I] [I]->[0]; };
  parent [0];
  scheme {
    int i;
    par (i=0; i<m; i++) 100%%[i];
    par (i=1; i<m; i++) 100%%[i]->[0];
  };
};
```

Informally, it looks like a description of an abstract network of processors, which executes the algorithm, complemented by the description of the workload of its processors and communication links, and the description of the scenario of interaction between the abstract processors during the algorithm execution.

The first line of the above definition introduces the name **Galaxy** of the type of the abstract mpC network and a list of parameters – integer scalar parameters **m** and **k** and vector parameter **n** of **m** integers. Next line declares the coordinate system to which abstract processors will be related. It introduces coordinate variable **I** ranging from **0** to **m-1**.

Next line associates abstract processors with this coordinate system and describes the volumes of computation to be performed by each of the processors. As a unit of measurement, the volume of computation performed by some benchmark code is used. In this particular case, it is assumed that the benchmark code computes a single group of **k** bodies. It is also assumed that **i**-th element of the vector parameter **n** is equal to the number of bodies in the group computed by the **i**-th abstract processor. The number of operations to compute one group is proportional to the number of bodies in the group squared. Therefore, the volume of computation to be performed by the **I**-th virtual processor is $(\mathbf{n[I]}/\mathbf{k})^2$ times bigger than the volume of computation performed by the benchmark code. This line just says it.

Next line specifies volumes of data in **Body**s to be transferred between the virtual processors during execution of the algorithm. It simply says that **i**-th virtual processor (**i**=1,…) will send attributes of all its bodies to the host-processor where they should be visualized. Note, that this definition describes one iteration of the main loop of the algorithm, which is a quite good approximation because practically all computations and communications concentrate in this loop. Therefore, the total time of the execution of this algorithm is approximately equal to the running time of a single iteration, multiplied by the total number of iterations.

Finally, the **scheme** block describes how exactly virtual processors interact during execution of the algorithm. It says that first all the virtual processors perform in parallel 100 per cent of computations that should be performed, and then all the processors, except the host processor, send in parallel 100 per cent of data that should be sent to the host-processor.

The most principal fragments of the rest code of this mpC application are:

```
void [*] main(int [host]argc, char **[host]argv)
{
   ...
   TestGroup[]=(*AllGroups[0])[];
   recon Update_group(TestGroup, TestGroupSize) ;
   {
      net GalaxyNet(NofG, TestGroupSize, NofB) g;
      …
   }
}
```

The **recon** statement uses a call of the function **Update_Group** with actual parameters **TestGroup** and **TestGroupSize** to update the estimation of the performance of the physical processors executing the application. The main part of the total volume of computations performed by each virtual processor just falls into execution of calls to the function **Update_Group**. Therefore, the obtained estimation of performances of the real processors will be very close to their actual performances shown while executing this program.

Next line defines the abstract network **g** of type **GalaxyNet** with the actual parameters **NofG** – the actual number of groups of bodies, **TestGroupSize** – the size of the test group of bodies used in the benchmark code, and **NofB** – an array of **NofG** elements containing actual numbers of bodies in the groups. The rest computations and communication will be performed on this abstract network.

The mpC programming system maps virtual processors of the abstract network **g** to real parallel processes constituting the running parallel program. While performing the mapping, the programming system uses, on the one hand, the information about configuration and performance of physical processors and communication links of the network of computers executing the program, and on the other hand, the specified performance model of the parallel algorithm. The programming system does the mapping at runtime and tries to minimise the total running time of the parallel program.

7. Conclusion

In this paper, we have analysed challenges associated with parallel programming for common heterogeneous networks of computers. This analysis has resulted in the description of the main features of an ideal parallel program for NoCs. We have taken a look at how standard parallel programming tools, such as HPF and MPI, addresses the programming challenges. We have also introduced the mpC language, which is the first language specifically designed for parallel programming for heterogeneous networks of computers. Detailed introduction to parallel computing on heterogeneous networks can be found in [11].

8. References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, *LAPACK User's Guide*, SIAM, Philadelphia, third edition, 1999.

[2] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, *ScaLAPACK User's Guide*, SIAM, Philadelphia, 1997.

[3] *High Performance Fortran Language Specification. Version 1.1.* High Performance Standard Forum, Rice University, Houston, Texas, November 10, 1994.

[4] *High Performance Fortran Language Specification. Version 2.0.* High Performance Standard Forum, Rice University, Houston, Texas, January 31, 1997.

[5] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, June 12, 1995.

[6] S.Louca, N.Neophytou, A.Lachanas, P.Evripidou, MPI-FT: Portable Fault Tolerance Scheme for MPI, *Parallel Processing Letters*, 10(4), pp.371-382, 2000.

[7] G.Fagg, A.Bukovsky, J.Dongarra, HARNESS and fault tolerant MPI, *Parallel Computing* 27(11), pp.1479-1496, 2001.

[8] A.Lastovetsky, D.Arapov, A.Kalinov, I.Ledovskih, A Parallel Language and Its Programming System for Heterogeneous Networks, *Concurrency: Practice and Experience*, 12(13), pp.1317-1343, 2000.

[9] A.Lastovetsky, Adaptive Parallel Computing on Heterogeneous Networks with mpC, *Parallel Computing*, 28(10) , pp.1369-1407, 2002.

[10] A.Lastovetsky, R.Reddy, HMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers, *Proceedings of the 17th International Parallel and Distributed Processing Symposium* (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2003.

[11] A.Lastovetsky, *Parallel Computing on Heterogeneous Networks*, John Wiley & Sons, 423 pp, June 2003, ISBN: 0-471-22982-2.