Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

J. Parallel Distrib. Comput. 66 (2006) 197–220

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

HeteroMPI: Towards a message-passing library for heterogeneous networks of computers

Alexey Lastovetsky*, Ravi Reddy

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

Received 18 March 2003; received in revised form 20 June 2005; accepted 24 August 2005
Available online 10 October 2005

Abstract

The paper presents Heterogeneous MPI (HeteroMPI), an extension of MPI for programming high-performance computations on heterogeneous networks of computers. It allows the application programmer to describe the performance model of the implemented algorithm in a generic form. This model allows the specification of all the main features of the underlying parallel algorithm, which have an impact on its execution performance. These features include the total number of parallel processes, the total volume of computations to be performed by each process, 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. Given a description of the performance model, HeteroMPI tries to create a group of processes that executes the algorithm faster than any other group. The principal extensions to MPI are presented. We demonstrate the features of the library by performing experiments with parallel simulation of the interaction of electric and magnetic fields and parallel matrix multiplication.
© 2005 Elsevier Inc. All rights reserved.

Keywords: MPI; Message-passing library; Heterogeneous networks; Parallel computing

1. Introduction

The standard Message Passing Interface (MPI) [9] is the main programming tool used for programming high-performance computations on homogeneous distributed-memory computer systems such as supercomputers and clusters of workstations. It is also normally used to write parallel programs for heterogeneous networks of computers (HNOCs). However, it does not provide tools that address some additional challenges posed by HNOCs, which are outlined below:

Heterogeneity of processors: A good parallel application for HNOCs must distribute computations unevenly taking into account the speeds of the processors. The efficiency of the parallel application also depends on the accuracy of estimation of the speeds of the processors of the HNOCs. This estimation is difficult because the processors may demonstrate different speeds for different applications due to differences in the instruction sets, the number of instruction execution units, the number of registers, the structure of memory hierarchy and so on.

Ad hoc communication network: The common communication network is normally heterogeneous. The latency and bandwidth of communication links between different pairs of processors may differ significantly. This makes the problem of optimal distribution of computations and communications across the HNOCs much more difficult than across a dedicated cluster of workstations interconnected with a homogeneous high-performance communication network. Another issue is that the common communication network can use multiple network protocols for communication between different pairs of processors. A good parallel application should be able to use multiple network protocols between different pairs of processors within the same application for faster execution of communication operations.

Multi-user decentralized computer system: Unlike dedicated clusters and supercomputers, HNOCs are not strongly centralized computer systems. A typical HNOC consists of relatively autonomous computers, where each one may be used and administered

* Corresponding author.
E-mail addresses: Alexey.Lastovetsky@ucd.ie (A. Lastovetsky), Manumachu.Reddy@ucd.ie (R. Reddy).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.08.002

independently. This leads to unstable performance characteristics of processors during the execution of a parallel program as the computers may be used for other computations and communications. Also there is a much higher probability of resource failures in HNOCs compared to dedicated cluster of workstations, which makes fault tolerance a desired feature for parallel applications running on HNOCs.

Thus, there are three main challenges posed by HNOCs, which are not addressed by the standard MPI specification.

Firstly, the standard MPI specification does not provide a means to use multiple network protocols between different pairs of processors for efficient communication in the same MPI application. A standard implementation of MPI does not address the challenge either. There have been majority of vendor implementations addressing this issue especially the use of shared memory and TCP/IP in MPICH [12], the support for multiple communication mediums (but not more than one device simultaneously) TCP, SMP, Myrinet, and InfiniBand in MPI/Pro [7,8], and support of multiple communication devices simultaneously in WMPI [20]. Advanced non-standard implementations of the standard MPI specification [11,1] have also been investigated.

Secondly, the standard MPI does not provide a means for the writing of fault-tolerant parallel applications for HNOCs. There are some research efforts made recently to address this challenge such as MPI-FT [19], MPI/FT [2], and the fault-tolerant MPI (FT-MPI) [10]. MPI-FT proposes a fault tolerant and recovery scheme for MPI, consisting of a mechanism for detecting and recovering from process failures. The recovery function simulates all the communication of the processes with the dead one by re-sending to the replacement process all the messages destined for the dead one. MPI/FT is a high-performance MPI-1.2 implementation enhanced with low-overhead functionality to detect and recover from process failures. FT-MPI is also an MPI-1.2 specification implementation that provides process level fault tolerance at the MPI API level. It allows the application to continue using a communicator with the failed rank while explicitly excluding communication with the failed rank, or to shrink the communicator by excluding the failed rank, or to spawn a new process to take the place of the failed process. However, it is still the responsibility of the application to recover the data-structures and the data on the crashed processes.

Thirdly, the standard MPI does not provide features, which facilitate the writing of parallel programs that distribute computations and communications unevenly, taking into account the speeds of the processors, and the speeds and bandwidths of communication links. To the best of the authors' knowledge, there is no research effort made to address this challenge. This paper presents an effort in this direction—a small set of extensions to MPI, called HeteroMPI (Heterogeneous MPI), aimed at efficient parallel computing on HNOCs, and its research implementation. The main goal of the design of the API in HeteroMPI is to smoothly and naturally extend the MPI model for HNOCs. This involves the design of a layer above MPI that does not involve any changes to the existing MPI API. The HeteroMPI API must be easy-to-use and suitable for most scientific applications. The HeteroMPI API must also facilitate transformation of MPI applications to HeteroMPI applications that run efficiently on HNOCs.

We start with presentation of the principal extensions to MPI. Then we demonstrate the features of the library with two parallel HeteroMPI applications. The first application simulates the interaction of electric and magnetic fields on a three-dimensional object. The second application multiplies two dense square matrices. These applications are representative of many scientific applications. Results of experiments with these applications on a HNOC are also presented. This is followed by survey of related work. We conclude the paper by quick analysis of some alternative approaches to heterogeneous extension of MPI.

2. Outline of HeteroMPI

The standard MPI specification provides communicator and group constructors, which allow the application programmers to create a group of processes that execute together some parallel computations to solve a logical unit of a parallel algorithm. The participating processes in the group are explicitly chosen from an ordered set of processes. This approach to the group creation is quite acceptable if the MPI application runs on homogeneous distributed-memory computer systems, one process per processor. In this case, the explicitly created group will execute the parallel algorithm typically with the same execution time as any other group with the same number of processes, because the processors have the same computing power, and the latency and the bandwidth of communication links between different pairs of processors are the same. However on HNOCs, a group of processes optimally selected by taking into account the speeds of the processors, and the latencies and the bandwidths of the communication links between them, will execute the parallel algorithm faster than any other group of processes. Selection of processes in such a group is usually a very difficult task. It requires the programmers to write a lot of complex code to detect the actual speeds of the processors and the latencies of the communication links between them, and then to use this information to select the optimal set of processes running on different computers of heterogeneous network.

The main idea of HeteroMPI is to automate the process of selection of such a group of processes that executes the heterogeneous algorithm faster than any other group.

The first step in this process of automation is the specification of the performance model of the heterogeneous parallel algorithm in a performance model definition language. The performance model allows an application programmer to specify his or her high-level knowledge of the application that can assist in finding the most efficient implementation on HNOCs. This model allows specification of all the main features of the underlying parallel algorithm that have an essential impact on application execution performance on HNOCs. These features are

- the total number of processes executing the algorithm;
- the total volume of computations to be performed by each of the processes in the group during the execution of the algorithm;
- the total volume of data to be transferred between each pair of processes in the group during the execution of the algorithm;
- the order of execution of the computations and communications by the parallel processes in the group, that is, how exactly the processes interact during the execution of the algorithm.

HeteroMPI provides a small and dedicated model definition language for specifying this performance model. This language uses most of the features in the specification of network types of the mpC language presented in [16]. mpC is a high-level parallel language (an extension of ANSI C), designed specially to develop portable adaptable applications for heterogeneous networks of computers. HeteroMPI's performance model definition language only uses the specification of the network types in mpC. The specification of performance model in HeteroMPI is the same as the specification of the performance model in mpC in the form of network type. Thus it can be said that HeteroMPI's model definition language is a subset of mpC language in that it uses only the feature of network types in mpC. A compiler compiles the description of this performance model to generate a set of functions. The functions make up an algorithm-specific part of the HeteroMPI runtime system.

Having provided such a description of the performance model, application programmers can use a new operation, whose interface is shown below. This operation tries to create a group that would execute the heterogeneous algorithm faster than any other group of processes.

```
HMPI_Group_create(HMPI_Group * gid, const HMPI_Model* perf_model,
            const void * model_parameters)
```

The parameter **`perf_model`** is a handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model, **`model_parameters`** are the parameters of the performance model (see example shown below). This function returns an HeteroMPI handle to the group of MPI processes in **`gid`**.

In HeteroMPI, groups are not absolutely independent of each other. Every newly created group has exactly one process shared with already existing groups. That process is called a *parent* of this newly created group, and is the connecting link, through which results of computations are passed if the group ceases to exist. **`HMPI_Group_create`** is a collective operation and must be called by the parent and all the processes, which are not members of any HeteroMPI group.

During the creation of this group of processes, the HeteroMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network. The solution to the problem is based on the following:

- The performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model.
- The performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm. This model considers the executing heterogeneous network as a multilevel hierarchy of interconnected sets of heterogeneous multiprocessors. This model takes into account the material nature of communication links and their heterogeneity.

The algorithms used to solve the problem of selection of processes are discussed in [16]. We present an informal description of the mapping algorithm here.

Each particular mapping, $\mu : I -> C$, where I is a set of processes of the group, and $C = \{c_0, c_1, \ldots, c_{M-1}\}$ is a set of computers of the executing network, is characterized by the estimation of the time of execution of the algorithm on the network of computers. The estimation is calculated based on the performance model of the parallel algorithm and the model of the executing network of computers.

Ideally, the HeteroMPI runtime system should find such a mapping that is estimated to ensure the fastest execution of the parallel algorithm. In general, for an accurate solution of this problem as many as $\boldsymbol{M^K}$ possible mappings have to be probated to find the best one ($\boldsymbol{K}$ is the number of processes of the group). Obviously, that computational complexity is not acceptable for a practical algorithm that should be performed at runtime. Therefore, the HeteroMPI runtime system searches for some approximate solution that can be found in some reasonable interval of time, namely, after probation of $\boldsymbol{M} \times \boldsymbol{K}$ possible mappings instead of $\boldsymbol{M^K}$.

Informally, the algorithm first maps the most loaded process not taking into account other processes as well as communications. Then, given the first process mapped, it maps the second most loaded process only taking into account communications between these two processes and so on. At the ith step, it maps the ith most loaded virtual process only taking into account data transfer between these i processes. This algorithm exploits the obvious observation that the smaller are things, the easier they can be evenly distributed. Hence, bigger things should be distributed under weaker constraints than smaller ones. For example, if you want to distribute a number of balls of different size over a number of baskets of different size, you better start with the biggest ball and put it into the biggest basket; then put the second biggest ball into the basket with the freest space and so on. This algorithm keeps balance between ball sizes and free basket space and guarantees that if at some step you do not have enough space for the next ball, it simply means that there is no way to put all the balls in the baskets. Similarly, if the above algorithm cannot balance the load of processors in case of practically zero communication costs, it means there is no way to balance them. This algorithm will also work well if data transfer between more loaded processes is more significant than data transfer between less loaded ones. In this case, more loaded virtual communication links are taken into account at earlier stages of the algorithm.

An obvious case when this mapping algorithm may not work well is when the least loaded processes is involved in transfer of much bigger volume of data than more loaded ones, and the contribution of communications in the total execution time is significant. But even quick analysis shows that it is not the case for most parallel algorithms.

The accuracy of the model of the executing network of computers depends upon the accuracy of the estimation of the actual speeds of processors. HeteroMPI provides an operation to dynamically update the estimation of processor speeds at runtime. It is especially important if computers, executing the target program, are used for other computations as well. In this case, the actual speeds of processors can dynamically change dependent on the external computations. The use of this operation, whose interface is shown below, allows the application programmers to write parallel programs, sensitive to such dynamic variation of the workload of the underlying computer system,

```
typedef void (*HMPI_Benchmark_function)(const void*, int, void*);
HMPI_Recon(HMPI_Benchmark_function func, const void* input_p,
        int num_of_parameters, void* output_p)
```

where all the processors execute the benchmark function **func** in parallel, and the time elapsed by each of the processors to execute the code is used to refresh the estimation of its speed. This is a collective operation and must be called by all the processes in the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HeteroMPI.

HeteroMPI provides a predefined communication universe **HMPI_COMM_WORLD**, which is a communicator consisting of all processes available for the computation in an HeteroMPI application; this communicator has the same value in all processes. It is an analog of **MPI_COMM_WORLD**, the predefined communication universe defined in the standard MPI specification. It cannot be deallocated during the life of the process. The group corresponding to this communicator is a pre-defined constant **HMPI_COMM_WORLD_GROUP**.

A typical parallel application is composed of one or more phases, which are sections of code comprised of computations and communications. If the phases are distinct, the application programmer has to optimally distribute computations involved in each phase amongst processors involved in executing the phase. To achieve load balance in each phase, we distribute computations in that phase so that the volume of computations executed by each processor be proportional to its speed. Thus if the phases are distinct, the application programmer has to determine the speeds of the processors for each phase. To determine the speeds of the processors for each phase, the application programmer provides benchmark code that is representative of the basic computations performed in that phase to **HMPI_Recon**. For each phase, all the processors execute the benchmark code. The relative speeds are determined from the time taken to execute the benchmark code. It is to be noted that the computation portions of each communication operation in a phase (preparation of the message by adding header, trailer, and error correction information, execution of the routing algorithm) must be taken into account. If the phases are not distinct, it is sufficient to use the speeds determined for one phase to be used in all the phases. However if the processors used in the execution of the parallel application are used for external computations, it is recommended that **HMPI_Recon** be used to determine the speeds of the processors for each phase so that computations are distributed over the processors in accordance to their actual performances at the moment of execution of the computations.

The accuracy of **HMPI_Recon** depends upon how accurately the benchmark code provided by the user reflects the core computations of each phase. If the benchmark code provided is an accurate measurement of the core computations in each phase, **HMPI_Recon** gives an accurate measure of the speeds.

Fig. 1 illustrates the usage of **HMPI_Recon** to write parallel programs sensitive to the dynamic variation of the workload of the underlying computer system. As can be seen from the figure, the combination of calls **HMPI_Recon** and **HMPI_Group_create** can be used for each distinct phase of the parallel application to create a group of processes that executes the computations and communications in that phase with best execution performance. One approach to tackle applications that do not have very uniform iterations is to break down the non-uniform iterations in the application into sets of uniform iterations. For each such set of uniform iterations, a performance model is designed. A group of processes is then created that can execute the set of uniform iterations with best execution performance and destroyed at the end of execution of the set.

Another principal operation provided by HeteroMPI allows application programmers to predict the total time of execution of the algorithm on the underlying hardware without its real execution. Its interface is shown below,

```
HMPI_Timeof (const HMPI_Model* perf_model, const void* model_parameters)
```

This function allows the application programmers to write a parallel application that can use different parallel algorithms to solve the same problem, making choice at runtime depending on the particular executing network and its actual performance. This is a local operation that can be called by any process, which is a member of the group **HMPI_COMM_WORLD_GROUP**.

```
void Phase1_benchmark_code(const void*, int, void*);
void Phase2_benchmark_code(const void*, int, void*);
int main() {
   …
   for (i = 0; i < number_of_iterations; i++) {
      double *phase1_speeds, *phase2_speeds;
      //Phase1
      if ((HMPI_Is_member(HMPI_COMM_WORLD_GROUP)) {
         HMPI_Recon(&Phase1_benchmark_code,…);
         HMPI_Get_processors_info(phase1_speeds);
      }
      //Distribute computations using the speeds
      HMPI_Group_create(…);
      //Execution of the computations and communications
      //Free the group
      //Phase2
      if ((HMPI_Is_member(HMPI_COMM_WORLD_GROUP)) {
         HMPI_Recon(&Phase2_benchmark_code,…);
         HMPI_Get_processors_info(phase2_speeds);
      }
      //Distribute computations using the speeds
      HMPI_Group_create(…);
      //Execution of the computations and communications
      //Free the group
      …
   }
}
```

Fig. 1. An example illustrating the usage of the operation **HMPI_Recon** to write parallel programs sensitive to dynamic changing loads.

This function invokes the HeteroMPI runtime system, which selects the optimal set of processes based on the performance model of the parallel algorithm **perf_model**, and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm. The estimated execution time of the algorithm by this optimal set of processes is then returned. The performance model parameters, **model_parameters**, are usually the following:

- number of processes in each dimension of the process arrangement;
- data distribution parameters specifying how the data is distributed amongst the processes, and the amount of data that is transferred between pairs of processes;

The process calling this function provides this information to the HeteroMPI runtime system, which uses it along with the model of the executing network of computers to estimate the time of execution of the algorithm. The model of the executing network of computers is implementation-dependent. Each computer in the model is characterized by 7 parameters:

(1) the first parameter called *scalability* determines how many non-interacting processes may run on the computer in parallel without loss of speed. This is useful, for example, if the computer is a multiprocessor workstation;
(2) the speed of the computer demonstrated on execution of some serial test code. This value is updated at runtime by the execution of the function **`HMPI_Recon`** ;
(3) the total number of parallel processes to run on the computer;
(4) the scalability of the communication layer provided by the computer;
(5) the last three parameters determine the speed of point-to-point data transfer between processes running on the same computer as function of size of the transferred data block. The fifth parameter specifies the speed of transfer of data block of 64 bytes (measured in bytes per second);
(6) the speed of transfer of data block of 64^2 bytes;
(7) the speed of transfer of data block of 64^3 bytes.

The speed of transfer of a data block of an arbitrary size is calculated by interpolation of the measured speeds.

The estimation procedure is explained in detail in [16]. The time of execution for each mapping, $\mu : I{-}{>}\ C$, where I is a set of the processes of the group, and $C = \{c_0, c_1, \ldots, c_{M-1}\}$ is a set of computers of the executing network, is estimated. The estimation time for the optimal mapping, which would ensure the fastest execution of the parallel algorithm, is returned. In general, for accurate solution of this problem as many as $\boldsymbol{M^K}$ possible mappings have to be probated to find the best one (here, $\boldsymbol{K}$ is the number of processes of the group). Obviously, that computational complexity is not acceptable for a practical algorithm that should be performed at runtime. Therefore, the HeteroMPI runtime system searches for some approximate solution that can be found in some reasonable interval of time, namely, after probation of $\boldsymbol{M \times K}$ possible mappings instead of $\boldsymbol{M^K}$.

The estimation procedure is summarized here. Each computation unit in the **scheme** declaration of the form $e\%\%[i]$ is estimated. Each communication unit of the form $e\%\%[i] \rightarrow [j]$ specifying transfer of data from virtual processor with coordinates i to the virtual processor with coordinates j is estimated. Simple calculation rules are used to estimate the sequential algorithmic patterns in the **scheme** declaration. For example, the estimation of the pattern

```
for (e1; e2; e3) a
```

is calculated as follows:

```
for (T =0, e1; e2; e3)
  T += time taken to execute action a
```

The rules just reflect semantics of the corresponding serial algorithmic patterns. The rule to estimate time for a parallel algorithmic pattern

```
par (e1; e2; e3) a
```

is more complicated and is explained in detail in [16].

HMPI_Timeof can thus be used to estimate the execution time on HNOCs for each possible set of model parameters. Application programmers can use this function creatively to design best possible heuristics for the set of parameters. Depending on the time estimated for each set, the optimal values of the parameters are determined. These values are then passed to the performance model during the actual creation of the group of processes using the function **HMPI_Group_create** .

The accuracy of the estimation by **HMPI_Timeof** is dependent on the following:

- the accuracy of the performance model of the algorithm designed by the user;
- the quality of the heuristics designed for the set of parameters provided to the performance model;
- the accuracy of the performance model of the executing network of computers. This depends on the accuracy of the measurements of the processor speeds given by **HMPI_Recon** and the communication model of the executing network of computers. Currently the communication model used in HeteroMPI runtime system is static. Future works would address the issue of efficiently updating the parameters of communication model at runtime.

One of the most important parameters, which influence the performance of the parallel application on HNOCs, is the number of processes used to execute the parallel application. Another principal operation provided by HeteroMPI frees application programmers from having to find the optimal number of processes that can execute the parallel application. They can specify only the rest of the parameters thus leaving the detection of the optimal number of processes to the HeteroMPI runtime system. Its interface is shown below,

```
HMPI_Group_auto_create (
    HMPI_Group* gid, const HMPI_Model* perf_model,
    const void * model_parameters)
```

This function returns an HeteroMPI handle to the group of MPI processes in **gid** . The parameter **perf_model** is a handle that encapsulates all the features of the performance model. These features are in the form of a set of functions generated by the compiler from the description of the performance model. The parameter **model_parameters** is an input parameter. Application programmer fills the parameter **model_parameters** with values of the input parameters to the performance model and ignores the return parameters specifying the number of processes to be involved in executing the algorithm and their relative performances.

HMPI_Group_auto_create is a collective operation and must be called by the parent and all the processes, which are not members of any HeteroMPI group.

There are no restrictions imposed by the function **HMPI_Group_create** . It just uses the input parameters provided to create a group of MPI processes. However, the function **HMPI_Group_auto_create** imposes certain restrictions, which are explained below:

(1) The application programmers describe the performance model of their implemented heterogeneous algorithm. The output parameters to the performance model are placed last in the list of parameters to the performance model. The output parameters are the number of processes in each dimension of the processor arrangement and an array representing the relative performances of the processors. Consider for example the performance model of an application multiplying two dense **n** $\times$ **n** matrices on a one-dimensional processor arrangement.

```
algorithm AXB_1d (int n, int p, int speeds [p])  {
  ...
};
```

In this performance model, the scalar parameter **n** is an input parameter. The output parameters are the scalar parameter **p** representing the number of processes in the linear array and the vector parameter **speeds** of size **p** representing the relative performances. These are the output parameters as they are determined by the function call **HMPI_Group_auto_create.** Consider for example the performance model of an application multiplying two dense **n** × **n** matrices on two-dimensional processor grid.

```
algorithm AxB_2d (int n, int p, int q, int speeds [p × q]) {
  ...
};
```

In this performance model, the scalar parameter **n** is an input parameter. The output parameters are the scalar parameter **p** , representing the number of processes in the column dimension of the processor grid arrangement, scalar parameter **q** , representing the number of processes in the row dimension of the processor grid arrangement, and the vector parameter **speeds** , of size **p** × **q**, representing the relative performances.
So generally speaking, the output parameter list contains scalar parameters, each parameter representing the number of processes in a dimension of the processor arrangement and a vector parameter representing the relative performances of the processors, the size of the vector being equal to the product of these scalar parameters.

(2) If the value of any parameter used in the body of the performance model declaration is dependent on the output parameters, then it should be obtained using functions. This is mainly the case for data distribution parameters, whose values are parameterized by number of processes in each dimension of the processor arrangement and the relative performances of the processors. This is because the function **HMPI_Group_auto_create** executes the performance model for different processor arrangements and hence the values of the parameters dependent on the arrangement of processors and their speeds should be obtained by using functions. Consider for example the performance model of an application implementing a parallel algorithm of the simulation of evolution of **n** bodies on one-dimensional processor arrangement.

```
int My_allocation_using_function (
    int I, int p, int *speeds, int n) ;
algorithm Nbody (int n, int p, int speeds [p]) {
  coord I=p;
  node {
    I >=0: bench * (n*
       My_allocation_using_function(I, p, speeds, n));
  } ;
  ...
};
```

In this performance model, to calculate the volume of computations in the **node** declaration performed by the processor with coordinate **I** , a function must be used (a user-defined function `My_allocation_using_function` is used in this case) which calculates and returns the number of bodies allocated to the processor **I** proportional to its speed.

(3) For the function call to **HMPI_Group_auto_create** , the application programmers supply values for the input parameters in the parameter list to the performance model. The output parameters are ignored.

After the call to the function **HMPI_Group_auto_create** , the output parameters, namely, the number of processes in each dimension of the processor arrangement can be obtained by using the HeteroMPI group accessor function **HMPI_Group_topology** . The relative performances of the processors can be obtained by using the HeteroMPI accessor function **HMPI_Group_performances** . All the members of the group then use the optimal performances to distribute computations so that the volumes of computations be proportional to their performances. This is followed by execution of the algorithm by the members of the group.

The HeteroMPI runtime system uses some default generic heuristics to find the optimal number of processes that can execute the parallel application on HNOCs. The function **HMPI_Group_auto_create** evaluates all the possible process arrangements. For each process arrangement, the function call **HMPI_Timeof** is used to estimate the time of execution of the algorithm. The function **HMPI_Group_auto_create** returns the process arrangement that results in the least estimated time of execution of the algorithm. As discussed before, the function call **HMPI_Timeof** invokes the mapping algorithms of the HeteroMPI runtime system to select such a mapping that is estimated to ensure the fastest execution of the parallel algorithm. During the execution of this functional call, the mapping algorithm re-orders the set of processors in accordance with the volume of computations to be performed by the processors, so that the most loaded processor comes first. The mapping algorithm thus re-orders processors in a non-increasing order of their speeds along each dimension of the processor arrangement. Beaumont et al. [3] show that the optimal mapping is one of the possible non-increasing arrangements where processors are arranged in a non-increasing order of their speed along each row and along each column of the two-dimensitonal processor grid arrangement. The function **HMPI_Group_auto_create** thus internally invokes mapping algorithms that use this heuristic in order to find the optimal number of processes that can execute the parallel application on HNOCs. The heuristics used may not be the best possible heuristics in most particular cases. HeteroMPI

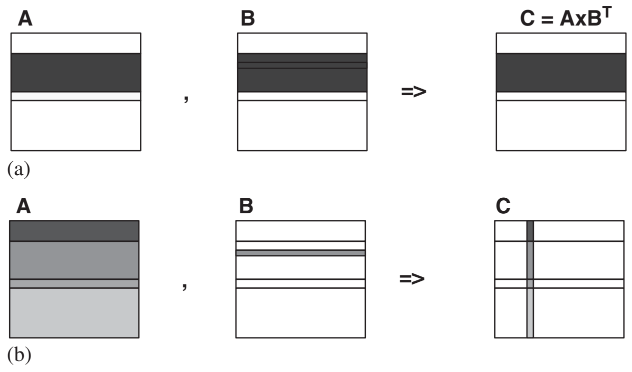

Fig. 2. (a) Matrix operation $C = A \times B^{\mathrm{T}}$ with matrices A, B, and C unevenly partitioned in one dimension. The slices mapped onto a single processor are shaded in black. (b) One step of parallel multiplication of matrices A and B. The pivot row of matrix B (shown slashed) is first broadcast to all processors. Then each processor computes, in parallel with the others, its part of the corresponding column of the resulting matrix C.

```
int Get_my_partition(int i, int p, const int *speeds, int n);
int Partition_set(int p, const int *speeds, int n, int *d);
algorithm ParallelAxBT(int n, int p, int speeds[p]) {
   coord I=p;
   node {
     I>=0: bench*(n*Get_my_partition(I, p, speeds, n));
   };
   link (J=p) {
     I!=J: length*(Get_my_partition(J, p, speeds, n)
                 *n*sizeof(double)) [J]->[I]
   };
   parent [0];
   scheme {
      int i, j, PivotProcessor=0, PivotRow=0, d[p];
      Partition_set(p, speeds, n, d);
      for (i = 0; i < n; i++, PivotRow+=1) {
         if (PivotRow >= d[PivotProcessor]) {
            PivotProcessor++;
            PivotRow = 0;
         }
         for (j = 0; j < p; j++)
            if (j != PivotProcessor)
               (100.)/d[PivotProcessor]) %% [PivotProcessor]->[j];
         par (j = 0; j < p; j++)
            (100./n) %% [j];
      }
   };
};
```

Fig. 3. Specification of the performance model of an algorithm of parallel matrix multiplication in the HeteroMPI's performance definition language.

provides an additional operation **`HMPI_Group_heuristic_auto_create`** that allows application programmers to supply heuristics that minimize the number of process arrangements evaluated.

The example shown in Figs. 2–4 illustrates the usage of the function **`HMPI_Group_auto_create`**. The example used is an application multiplying matrix A and the transposition of matrix B on **p** interconnected heterogeneous processors, i.e., implementing matrix operation $C = A \times B^T$, where A, B are dense $\mathbf{n} \times \mathbf{n}$ matrices. This application assumes one process per processor configuration and implements a naive heterogeneous algorithm shown in Fig. 2. It can be summarized as follows:

- The A, B, and C matrices are identically partitioned into **p** horizontal slices. There is one-to-one mapping between these slices and the processors. Each processor is responsible for computing its C slice;
- At each step, a row of matrix B (the pivot row), representing a column of blocks of matrix B^T, is communicated (broadcast) vertically; and all processors compute the corresponding column of C in parallel;
- To balance the load of the processor, the area of the slice mapped to each processor is proportional to it speed;
- Communication overheads may exceed gains due to parallel execution of computations. Therefore, there exists some optimal subset of available processors to perform the matrix multiplication. The function **`HMPI_Group_auto_create`** detects this optimal subset.

The definition of **`ParallelAxBT`** given in Fig. 3 describes the performance model of this heterogeneous algorithm.

```
int main() {
    int opt_p, *opt_speeds, *model_params, n, nd, **dp;
    int output_p, input_p = n;
    HMPI_Group gid;
    HMPI_Init(argc, argv);
    if ((HMPI_Is_member(HMPI_COMM_WORLD_GROUP))
       HMPI_Recon(&serialAxBT, &input_p, 1, &output_p);
    if (HMPI_Is_host()) {
       // The user fills in only the first parameter
       // Parameters 'p' and 'speeds' returned by
       // the call to function HMPI_Group_auto_create
       model_params[0] = n;
    }
    if (HMPI_Is_host())
       HMPI_Group_auto_create (&gid, &HMPI_Model_parallelAxBT,
                               model_params)
    if (HMPI_Is_free())
       HMPI_Group_auto_create (&gid, &HMPI_Model_parallelAxBT,
                               NULL)
    if (HMPI_Is_member(&gid)){
       HMPI_Group_topology(&gid, &nd, dp);
       opt_p = (*dp)[0];
       HMPI_Group_performances(&gid, opt_speeds);
       // Distribute computations using the optimal speeds of
       // processes.
       // computations and communications are performed here
    }
    if (HMPI_Is_member(&gid))
       HMPI_Group_free(&gid);
    HMPI_Finalize(0);
}
```

Fig. 4. The most principal fragments of the usage of function **HMPI_Group_auto_create** for detection of the optimal subset of processors to execute the parallel matrix multiplication and creation of the corresponding optimal group of processes (one process per processor configuration is assumed).

The performance model **ParallelAxBT** describing the algorithm has 3 parameters. Parameter **n** is the size of square matrices *A*, *B*, and *C*. Parameters **p** and **speeds** are output parameters. They represent the number and the performances of processors in the optimal subset respectively.

Function **Get_my_Partition** , used in the **node** and **link** declarations, partitions a set of **n** elements into **p** disjoint partitions such that the number of elements in each partition is proportional to the speed of the processor owning that partition. It returns the number of elements in the *i*th partition belonging to the *i*th processor. The **node** declaration specifies the volume of computations to be performed by the *i*th processor executing the algorithm. The unit of computation used to measure the volume is the computation of one element of the resulting matrix *C*. It is presumed that function **serialAxBT** used in **HMPI_Recon** to update the performance model of the executing network of computers just implements this elementary computation. The **link** declaration specifies that each processor will send its *B* slice to all other processors executing the algorithm.

The **scheme** declaration specifies **n** successive steps of the algorithm. At each step, the processor **PivotProcessor** , which holds the pivot row, sends it to rest of the processors thus executing **(1/d[PivotProcessor])**×**100** percent of total data transfer through the corresponding data link. Then, all processors compute the corresponding column of blocks of matrix *C* in parallel, each thus executing **(1/n)**×**100** percent of the total volume of computation to be performed by the processor. Function **Partition_set** used in the **scheme** declaration partitions a set of size **n** into **p** disjoint subsets on a linear array of **p** processors such that the number of elements in each subset is proportional to the speed of the processor owning that subset.

The principal fragments of rest of the code of the parallel application are shown in the **main** function in Fig. 4.

The HeteroMPI runtime system is initialized using operation **HMPI_Init** . Then, the **HMPI_Recon** operation updates the estimation of performances of processors using some serial multiplication of test matrices using function **serialAxBT** .

This is followed by the creation of a group of processes using operation **HMPI_Group_auto_create** . Users specify only the first model parameter to the performance model and ignore the return parameters specifying the number of processes to be involved in executing the algorithm and their performances. This function calculates the optimal number of actual processes to be involved in the parallel matrix multiplication and their performances. After the execution of the function **HMPI_Group_auto_create** , the optimal number of actual processes **opt_p** is obtained by using the HeteroMPI group accessor function **HMPI_Group_ topology** . The performances, **opt_speeds** , are obtained by using the HeteroMPI group accessor function **HMPI_Group_ preformances** .

The members of this group then perform the computations and communications of the heterogeneous parallel algorithm using standard MPI means. This is followed by freeing the group using operation **HMPI_Group_free** and the finalization of HeteroMPI runtime system using operation **HMPI_Finalize** .

A typical HeteroMPI application starts with the initialization of the HeteroMPI runtime system using the operation

```
HMPI_Init (int argc, char ** argv)
```

where **`argc`** and **`argv`** are the same as the arguments passed to **`main`**. This routine must be called before any other HeteroMPI routine and must be called once. This routine must be called by all the processes running in the HeteroMPI application.

After the initialization, application programmers can call any other HeteroMPI routines. In addition, MPI users can use normal MPI routines, with the exception of MPI initialization and finalization, including the standard group management and communicator management routines to create and free groups of MPI processes. However, they must use the predefined communication universe **`HMPI_COMM_WORLD`** of HeteroMPI instead of **`MPI_COMM_WORLD`** of MPI.

The application programmers should avoid using groups created with the MPI group constructor operations, to perform computations and communications in parallel with HeteroMPI groups, as it may not result in the best execution performance of the application. The point is that the HeteroMPI runtime system is not aware of any group of MPI processes, which is not created under its control. Therefore, the HeteroMPI runtime system cannot guarantee that an HeteroMPI group will execute its parallel algorithm faster than any other group of MPI processes if some groups of MPI processes, other than HeteroMPI groups, are active during the algorithm execution.

The only group constructor operations provided by HeteroMPI are **`HMPI_Group_create`** and **`HMPI_Group_auto_create`** the only group destructor operation provided by HeteroMPI is

```
HMPI_Group_free(HMPI_Group* gid)
```

where **`gid`** is the HeteroMPI handle to the group of MPI processes. This is a collective operation and must be called by all the members of this group. There are no analogs of other group constructors of MPI such as the set-like operations on groups and the range operations on groups in HeteroMPI. This is because:

- firstly, HeteroMPI does not guarantee that groups composed using these operations can execute a logical unit of parallel algorithm faster than any other group of processes;
- secondly, it is relatively straightforward for application programmers to perform such group operations by obtaining the groups associated with the MPI communicator given by the **HMPI_Get_comm** operation (see the interface shown below).

The other additional group management operations provided by HeteroMPI apart from the group constructor and destructor are the following group accessors:

- **HMPI_Group_rank** to get the rank of the process in the HeteroMPI group;
- **HMPI_Group_size** to get the number of processes in this group.

The initialization of HeteroMPI runtime system is typically followed by

- updating of the estimation of the speeds of processors with **HMPI_Recon**;
- finding the optimal values of the parameters of the parallel algorithm with **HMPI_Timeof**;
- creation of a group of processes, which will perform the parallel algorithm, by using **HMPI_Group_create** or **HMPI_Group_auto_create**;
- execution of the parallel algorithm by the members of the group. At this point, control is handed over to MPI. MPI and HeteroMPI are interconnected by the operation

  ```
  const MPI_Comm* HMPI_Get_comm (const HMPI_Group* gid) ,
  ```

  which returns an MPI communicator with communication group of MPI processes defined by **`gid`**. This is a local operation not requiring inter-process communication. Application programmers can use this communicator to call the standard MPI communication routines during the execution of the parallel algorithm. This communicator can safely be used in other MPI routines;
- freeing the HeteroMPI groups with **`HMPI_Group_free`**;
- finalizing the HeteroMPI runtime system by using operation

  ```
  HMPI_Finalize (int exitcode).
  ```

An HeteroMPI application is like any other MPI application and can be deployed to run in any environment where MPI applications are used. HeteroMPI applications can be run in environments where batch queuing and resource management systems are used. However, HeteroMPI uses its own measurements and performance models of the underlying system for running parallel applications efficiently.

One simple application of HeteroMPI is the conversion of conventional parallel programs that are designed to run on *massively parallel processors* (MPP) such as ScaLAPACK programs to HeteroMPI programs with minor rewriting of these applications, which mainly includes the insertion of HeteroMPI group creation and destruction calls. These HeteroMPI programs do not aim to extract the maximum performance from a heterogeneous network but provide an easy and simple way to execute the conventional parallel programs on HNOCs with good performance improvements. To write such a HeteroMPI program, firstly, application programmers describe the performance model of their homogeneous algorithm. Secondly, the transformed HeteroMPI program uses a multiprocessing algorithm that allows more than one process involved in its execution to be run on each processor. The upper bound on the number of processes executed on each processor is roughly equal to the ratio of speed of the fastest processor to speed of the slowest processor on the executing network of computers. During the creation of a HeteroMPI group of processes, the mapping of the parallel processes to the executing network of computers is performed such that the number of processes running on each processor is proportional to its speed. In other words, while distributed evenly across parallel processes, data and computations are distributed unevenly over processors of the heterogeneous network, and this way each processor performs the volume of computations proportional to its speed.

3. Translation of MPI to HeteroMPI

The section explains the steps involved in the transformation from an MPI program to HeteroMPI program. The straightforward transformations consist of one-to-one replacement of the MPI components by the HeteroMPI counterparts. They are:

- replacing **`MPI_Init`** with **`HMPI_Init`** and **`MPI_Finalize`** with **`HMPI_Finalize`**;
- replacing the MPI predefined universe **`MPI_Comm_world`** with the HeteroMPI predefined universe **`HMPI_Comm_world`**;
- replacing the MPI group accessors **`MPI_Group_rank`**, and **`MPI_Group_size`** with the HeteroMPI group accessors **`HMPI_Group_rank`**, and **`HMPI_Group_size`**.

There is absolutely no change in the code consisting of computations and communications of the parallel algorithm between an HeteroMPI program and the MPI program. The MPI communicator used in this code can be replaced with the MPI communicator provided by the operation **`HMPI_Get_comm`** on the HeteroMPI group of processes.

The other transformations are a bit involved and are outlined below in the order of increasing complexity:

- determination of the speeds of the processors using **HMPI_Recon**;
- creation of an HeteroMPI group of processes that will execute the heterogeneous parallel algorithm using the operation **`HMPI_Group_create`**. The parameters to the performance model passed to this operation can be composed using the function **`HMPI_Pack_model_parameters`**;
- destruction of an HeteroMPI group once the execution of the algorithm is finished using the operation **`HMPI_Group_free`**. This is similar to the group destructor for an MPI group of processes **`MPI_Group_free`**;
- description of the heterogeneous algorithm in the form of a performance model.

It can be seen that the most involved part in the transformation process is the design of the performance model.

The main constructs of the language are briefly described here. The **coord** declaration specifies the arrangement of processes. The **node** declaration describes the total volume of computations to be performed by each of the processes in the group during the execution of the algorithm. The **link** declaration specifies the total volume of data to be transferred between each pair of processes in the group, during the execution of the algorithm. The **scheme** declaration describes the order of execution of the computations and communications by the involved parallel processes in the group, that is, how exactly the processes interact during the execution of the algorithm.

The parameters to the performance model are mainly and usually the number of processes in each dimension of the process arrangement and data distribution parameters specifying how the data is distributed amongst the processes, and the amount of data that is transferred between the pair of processes. For example, if the mathematical objects used in the parallel algorithm are sets, the data distribution parameters are usually an array giving the number of elements in the set assigned to each processor proportional to the speed of the processor and an array giving the number of elements transferred between pairs of processors. If the mathematical objects used are matrices, the data distribution parameters are arrays giving the geometric dimensions of the partitions, which are rectangles assigned to each processor. If the mathematical objects used are graphs and trees, the data distribution parameters are arrays giving the number of nodes assigned to each processor and the edges that cross between pairs of processors.

It would appear that the description of the heterogeneous algorithm in the form of the performance model could be very complicated. However, the user who has designed an MPI application has complete knowledge of the essential features of the parallel algorithm used in the MPI application. While designing the performance model, all that the user has to do is to explicitly specify these features in a parametric way. The specification of the performance model provides all the features to allow the user to specify all these features outlined previously in a generic form without going into the nitty-gritty of the parallel algorithm.

The complexity of the performance model depends on the complexity of the algorithm that the user has designed for the parallel application. The user can simplify the design of the performance model by ignoring some details of the parallel algorithm that have little or no influence on the performance of the parallel application. The performance model language offers all the features allowing the users to design all types of performance models ranging from the simplest to most complicated, and from the not very accurate to the very accurate for their parallel application. In some cases, a simple performance model can be designed that can accurately represent the essential features of the parallel algorithm used in their parallel applications. The specification of the performance model is comprehensive enough for expressing many scientific applications, as subsequent sections show. At the same time it is expected to be improved based on the feedback from the scientific community using it.

The following sections illustrate the transformation steps.

4. Example of irregular HeteroMPI application

To explain how an application programmer can use HeteroMPI to write a real-life irregular application, consider the EM3D application simulating the interaction of electric and magnetic fields on a three-dimensional object [23,6]. The system consists of a few large subbodies resulting from a decomposition of the three-dimensional object. The subbodies contain varying number of E nodes where electric field values are calculated and H nodes where magnetic fields are calculated. The changes in the electric field of an E node are calculated as a linear function of the magnetic field values of its neighbouring H nodes and vice versa. Thus, the dependencies between E and H nodes form a bipartite graph. In a bipartite graph, the nodes are decomposed into two disjoint sets such that no two nodes within the same set are adjacent. Here the two disjoint sets are the set of E nodes and the set of H nodes. The subbodies are so decomposed from the three-dimensional object that the nodes in each subbody have few dependencies on the nodes residing in other subbodies thereby reducing the communications between a pair of subbodies. A sample decomposition of a three-dimensional object into three subbodies is shown in Fig. 5(a).

A simple example of bipartite graph is shown in Fig. 5(b).

The parallel algorithm of this application consists of a few parallel processes, each of which updates data characterizing a single subbody. The heterogeneous algorithm can be summarized as follows:

- At each step of the algorithm,
  - For each of the E nodes in its subbody, if any of the neighbouring H nodes reside remotely, each process receives the values of these nodes from the process owning them;
  - Each process in parallel computes the new value of the electric field of each of the E nodes in its subbody;
  - For each of the H nodes in its subbody, if any of the neighbouring E nodes reside remotely, each process receives the values of these nodes from the process owning them;
  - Each process in parallel computes the new value of the magnetic field of each of the H nodes in its subbody.

The most interesting fragments of the MPI version of this parallel application are shown in Fig. 6.

As shown in the MPI program above, the participating parallel processes in the group associated with the MPI communicator **em3dcomm** are explicitly chosen from an ordered set of processes specified by the group associated with the MPI communicator **MPI_COMM_WORLD**. If the MPI application runs on a homogeneous distributed-memory computer system, this group will execute the parallel algorithm with the same execution time as any other MPI group of processes, just because all processors run at the same speed, and all communication links transfer data at the same speed. However, if the MPI program runs on a HNOC, this group will execute the parallel algorithm sometimes slower and sometimes faster than other groups of processes. This is because different processors of the HNOC will execute the same computations at different speeds, and different pair of processors will communicate at different speeds. MPI does not facilitate creation of a group of processes where the processes are optimally selected taking into account the speeds of the processes, and the speeds and the bandwidths of the communication links between them. It is only a pure chance if the MPI group of processes executes the parallel algorithm faster than any other MPI group of processes on the HNOC.

If there is more than one process per processor, the first **p** processes are used to execute the MPI application. However, the HeteroMPI application will select an optimal set of processes consisting of **p** processes dropping the rest of the processes from the computation when their participation can degrade performance. The MPI communicator **em3dcomm** represents this optimal set of processes in the HeteroMPI application whereas it consists of first **p** processes from the predefined MPI communication universe **MPI_COMM_WORLD** in the corresponding MPI application.

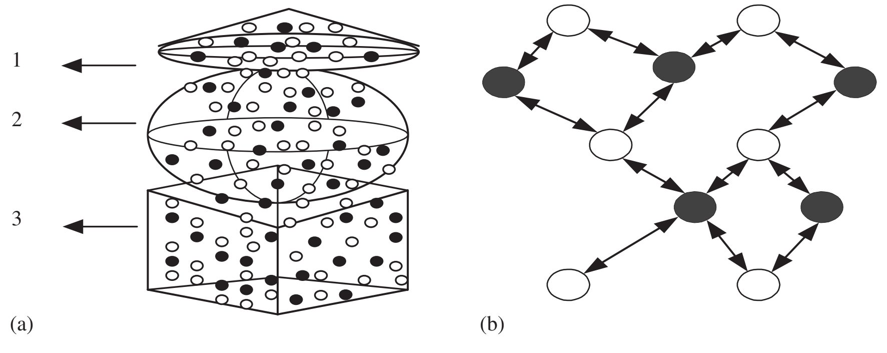

Fig. 5. (a) A sample three-dimensional object consists of three subbodies. In each subbody, the electric field value is represented as a white dot, an E node, and the magnetic field value represented by a black dot, an H node. (b) A bipartite graph showing the dependencies between E and H nodes.

```
int main(int argc, char **argv) {
   MPI_Comm em3dcomm;
   int i, me, is_executing_algo = MPI_UNDEFINED, E = 0, H = 1;
   int p, niter;              /* Inputs to the program */
   struct EM3D_body_t* bodies;  /* Inputs to the program */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &me);
   if (me >= 0 && me < p) is_executing_algo = 1;
   MPI_Comm_split(MPI_COMM_WORLD, is_executing_algo, 1, &em3dcomm);
   if (is_executing_algo) {
     Initialize_system(p, bodies);
     MPI_Comm_rank(&em3dcomm, &me);
     for (i = 0; i < niter; i++) {
       Gather_remote_H_boundary_values(me, H, p, bodies, &em3dcomm);
       Compute_E_values(me, E, p, bodies);
       Gather_remote_E_boundary_values(me, E, p, bodies, &em3dcomm);
       Compute_H_values(me, H, p, bodies);
     }
     MPI_Comm_free(&em3dcomm);
   }
   MPI_Finalize();
}
```

Fig. 6. The most relevant fragments of code of the MPI program implementing the EM3D algorithm.

```
algorithm Em3d(int p, int k, int d[p], int dep[p][p]) {
  coord I=p;
  node {I>=0: bench*(d[I]/k);};
  link (L=p) {
    I>=0 && I!=L && (dep[I][L] > 0) :
      length*(dep[I][L]*sizeof(double)) [L]->[I];
  };
  parent[0];
  scheme {
    int current, owner, remote;
    par (owner = 0; owner < p; owner++)
        par (remote = 0; remote < p; remote++)
            if ((owner != remote) && (dep[owner][remote] > 0))
               100%%[remote]->[owner];
    par (current = 0; current < p; current++) 100%%[current];
  };
 }
```

Fig. 7. Specification of the performance model of the EM3D algorithm in the HeteroMPI's performance definition language.

The HeteroMPI version of this parallel application involves first describing the performance model of the parallel algorithm. The definition of **`Em3d`** shown in Fig. 7 describes the performance model of the heterogeneous algorithm of this parallel application.

The model describing the algorithm has 4 parameters:

- parameter **`p`** specifies the number of processes executing the algorithm;
- parameter **`k`** specifies the number of nodes in a single subbody, whose data is computed in the benchmark code that is truly representative of the underlying application;
- it is supposed that *i*th element of the vector parameter **`d`** gives the number of nodes in the subbody computed by the *i*th process participating in the execution of the algorithm;
- parameter **`dep`** specifies the number of nodal values communicated between different pairs of subbodies: **`dep[I][J]`** gives the number of nodal values in the subbody **`J`** that subbody **`I`** needs to compute its nodal values.

The **`coord`** declaration introduces one coordinate variable **`I`** ranging from **`0`** to **`p-1`**.

The **`node`** declaration associates the processes with this coordinate system to form a linear processor arrangement. It also describes the absolute volume of computation to be performed by each of the processes. 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 the nodal values of **`k`** nodes in a single subbody. At each step of the algorithm, abstract processor P_I updates **`d[I]`** nodes. As computations during the updating of one single subbody mainly falls into the calculation of nodal values, the volume of computations performed by the abstract processor P_I will be approximately **`d[I]/k`** times larger than the volume of computations performed by the benchmark code.

```
int main(int argc, char **argv) {
      MPI_Comm em3dcomm;
      int i, me, k, E = 0, H = 1;
      HMPI_Group gid;
      void* model_params;
      int p, niter;                /* Inputs to the program */
      struct EM3D_body_t* bodies;  /* Inputs to the program */
      HMPI_Init(argc, argv);
      if (HMPI_Is_member(HMPI_COMM_WORLD_GROUP)) {
         int output_p;
         Body recon_body;
         // Construct recon parameters that are
         // representative of the application
         ...
         HMPI_Recon(&Serial_em3d, &recon_body, 1, &output_p);
      }
      if (HMPI_Is_host()) {
         HMPI_Pack_model_parameters(p, k, d, dep, model_params);
         HMPI_Group_create(&gid, &HMPI_Model_Em3d, model_params);
      }
      if (HMPI_Is_free())
         HMPI_Group_create(&gid, &HMPI_Model_Em3d, NULL);
      if (HMPI_Is_member(&gid)) {
        em3dcomm = *(MPI_Comm*)HMPI_Get_comm(&gid);
        Initialize_system(p, bodies);
        MPI_Comm_rank(&em3dcomm, &me);
        for (i = 0; i < niter; i++) {
          Gather_remote_H_boundary_values(me, H, p, bodies, &em3dcomm);
          Compute_E_values(me, E, p, bodies);
          Gather_remote_E_boundary_values(me, E, p, bodies, &em3dcomm);
          Compute_H_values(me, H, p, bodies);
        }
      }
      if (HMPI_Is_member(&gid)) HMPI_Group_free(&gid);
      HMPI_Finalize(0);
   }
```

Fig. 8. The most relevant code fragments of the HeteroMPI program implementing the algorithm of EM3D.

The **link** declaration specifies the volumes of data to be transferred between the processes at each step of the algorithm. Process P_I owning subbody **I** receives **dep[I][L]** remote boundary values from the subbody **L** owned by processor P_L. Thus, the total volume of data to be transferred from P_L to P_I will be equal to **dep[I][L]*sizeof(double)**.

The **scheme** declaration describes how the abstract processors interact during the execution of one iteration of the algorithm:

- Each processor P_{owner} first receives the remote values required for the calculation of the nodal values in its subbody. During this communication operation, 100% of data that should be sent from each process P_{remote} to process P_{owner} at this step will be sent. The second nested **par** statement in the main **for** loop of the **scheme** declaration just specifies this. The **par** algorithmic patterns are used to specify that during the execution of this communication, data transfer between different pairs of processes is carried out in parallel.
- Each process then computes the new values for each of the nodes in its subbody. The process will perform 100% of computations it should perform during this iteration. The **par** algorithmic patterns are used here to specify that all processes perform their computations in parallel.

Note that the above performance model describes only one iteration of the algorithm. This approximation is accurate enough because at any iteration each process performs the same volume of computations, and the same volume of data is transferred between each pair of processes.

The most interesting code fragments of the HeteroMPI parallel application are shown in Fig. 8. The HeteroMPI runtime system is initialized using operation **HMPI_Init** . Then, operation **HMPI_Recon** updates the estimation of performances of processors using the serial EM3D program computing nodal values for a single subbody. The computations performed by each processor mainly fall into the execution of calls to function **Serial_em3d** .

This is followed by the creation of a group of processes using operation **HMPI_Group_create** . The members of this group then perform the computations and communications of the heterogeneous parallel algorithm using standard MPI means. This is followed by freeing the group using operation **HMPI_Group_free** , and by finalizing the HeteroMPI runtime system using operation **HMPI_Finalize** .

On HNOCs, the running time of the HeteroMPI program shown above will normally be less than the running time of the corresponding MPI program. This is because an HeteroMPI group of processes is created to execute the parallel algorithm faster than any other group of processes including the groups of processes created using MPI means. The processes participating in the HeteroMPI group are chosen to minimize the execution time of the algorithm taking into account all its main features, which have an impact on the application execution performance. The application programmer describes all the main features of the parallel algorithm using the performance model **Em3d**, which are:

- the total number of participating processes `p` ;
- the total volume of computations to be performed by each of the processes as specified in **node** declaration. The volume of computations is mainly the computation of field values of nodes in a subbody. This is determined by the number of nodes within a subbody;
- the total volume of data to be transferred between each pair of processes as specified by the **link** declaration. The volume of data transferred equals the number of bytes of remote boundary values communicated between the subbodies;
- how exactly the processes interact during the execution of the algorithm as specified by the **scheme** declaration. Informally this looks like the description of the algorithm describing the interaction between the processes during the execution of the algorithm.

During the creation of the group of processes, the HeteroMPI runtime system uses the information from the performance model to solve the problem of selection of the optimal set of processes running on different computers of a heterogeneous network.

It can also be seen from the MPI and HeteroMPI programs described in this section that there is essentially no change in code of the parallel algorithm executed by the members of the group of processes participating in the parallel program. The main difference lies only in the creation of a group of processes.

5. Example of regular HeteroMPI application

An irregular problem is characterized by some inherent coarse-grained or large-grained structure implying quite deterministic decomposition of the whole program into a set of processes running in parallel and interacting via message passing. As a rule, there are essential differences in volumes of computations and communications to perform by different processes. The EM3D problem is an example of irregular problem.

Unlike an irregular problem, for a regular problem the decomposition of the whole program into a large set of small equivalent programs, running in parallel and interacting via message passing, is the most natural one. Multiplication of dense matrices is an example of a regular problem. The main idea of efficiently solving a regular problem is to reduce it to an irregular problem, the structure of which is determined by the irregularity of underlying hardware rather than the irregularity of the problem itself. So, the whole program is decomposed into a set of programs, each made from a number of the small equivalent programs stuck together and running on a separate processor of the underlying hardware.

Consider the problem of parallel matrix multiplication (MM) on HNOCs. The algorithm for the matrix operation $C = A \times B$, on a HNOC, is obtained by modification of the ScaLAPACK [5] two-dimensional block-cyclic MM algorithm. The modification is that the heterogeneous two-dimensional block-cyclic data distribution of [14] is used instead of the standard homogeneous data distribution. Thus, the heterogeneous algorithm of multiplication of two dense square $(\mathtt{n} \times \mathtt{r}) \times (\mathtt{n} \times \mathtt{r})$ matrices A and B on an $\mathtt{m} \times \mathtt{m}$ grid of heterogeneous processors can be summarized as follows:

- Each element in A, B, and C is a square $\mathtt{r} \times \mathtt{r}$ block and the unit of computation is the updating of one block, i.e., a matrix multiplication of size `r` . Each matrix is partitioned into generalized blocks of the same size $(\mathtt{l} \times \mathtt{r}) \times (\mathtt{l} \times \mathtt{r})$, where $\mathtt{m} \leqslant \mathtt{l} \leqslant \mathtt{n}$. The generalized blocks are identically partitioned into $\mathtt{p}^2$ rectangles, each being assigned to a different processor. The area of each rectangle is proportional to the speed of the processor that stores the rectangle. The partitioning of a generalized block is performed as follows:
  - Each element in the generalized block is a square $\mathtt{r} \times \mathtt{r}$ block of matrix elements. The generalized block is an $\mathtt{l} \times \mathtt{l}$ square of $\mathtt{r} \times \mathtt{r}$ blocks.
  - First, the $\mathtt{l} \times \mathtt{l}$ square is partitioned into `m` vertical slices, so that the area of the jth slice is proportional to $\sum_{i=1}^{m} s_{ij}$(see Fig. 9(a)). It is supposed that blocks of the jth slice will be assigned to processors of the jth column in the $\mathbf{m} \times \mathbf{m}$ processor grid. Thus, at this step, we balance the load between processor columns in the $\mathbf{m} \times \mathbf{m}$ processor grid, so that each processor column will store a vertical slice whose area is proportional to the total speed of its processors.
  - Then, each vertical slice is partitioned independently into `m` horizontal slices, so that the area of the ith horizontal slice in the jth vertical slice is proportional to s_{ij} (see Fig. 9(b)). It is supposed that blocks of the ith horizontal slice in the jth vertical slice will be assigned to processor P_{ij}. Thus, at this step, we balance the load of processors within each processor column independently.
- At each step `k` ,
  - each $\mathtt{r} \times \mathtt{r}$ block a_{ik} of the pivot column of matrix A is sent horizontally from the processor, which stores this block, to $\mathtt{m} - 1$ processors (see Fig. 10);
  - each $\mathtt{r} \times \mathtt{r}$ block b_{kj} of the pivot row of matrix B is sent vertically from the processor, which stores this block, to $\mathtt{m} - 1$ processors (see Fig. 10).
- Each processor updates its rectangle in the C matrix with one block from the pivot row and one block from the pivot column.

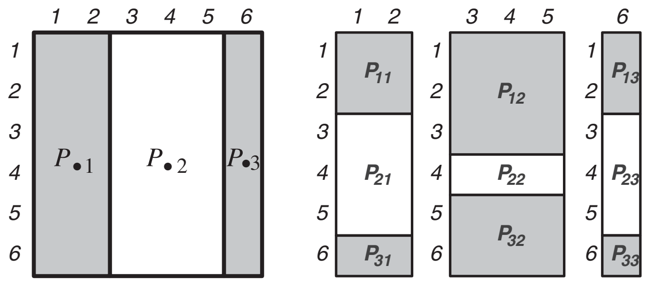

Fig. 9. Example of two-step distribution of a 6×6 generalized block over a 3×3 processor grid. The relative speed of processors is given by matrix $s = \begin{pmatrix} 0.11 & 0.25 & 0.05 \\ 0.17 & 0.09 & 0.08 \\ 0.05 & 0.17 & 0.03 \end{pmatrix}$. (a) At the first step, the 6×6 square is distributed in a one-dimensional block fashion over processor columns of the 3×3 processor grid in proportion $0.33 : 0.51 : 0.16 \approx 2 : 3 : 1$. (b) At the second step, each vertical rectangle is distributed independently in a one-dimensional block fashion over the processors of its column. The first rectangle is distributed in proportion $0.11 : 0.17 : 0.05 \approx 2 : 3 : 1$. The second one is distributed in proportion $0.25 : 0.09 : 0.17 \approx 3 : 1 : 2$. The third is distributed in proportion $0.05 : 0.08 : 0.03 \approx 2 : 3 : 1$.

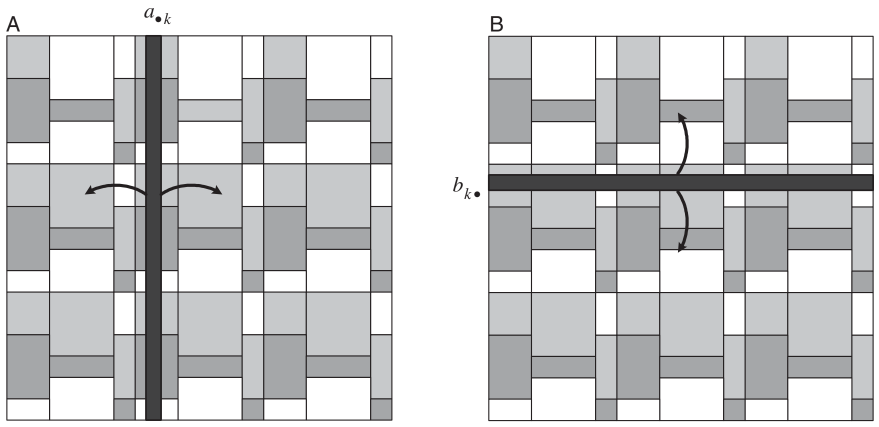

Fig. 10. One step of the algorithm of parallel matrix–matrix multiplication based on heterogeneous two-dimensional block distribution of matrices A, B, and C. First, each $\mathbf{r} \times \mathbf{r}$ block of the pivot column $a_{\bullet k}$ of matrix A (shown shaded dark grey) is broadcast horizontally, and each $\mathbf{r} \times \mathbf{r}$ block of the pivot row $b_{k\bullet}$ of matrix B (shown shaded dark grey) is broadcast vertically.

The definition of **`ParallelAxB`** given in Fig. 11 describes the performance model of this heterogeneous algorithm.

The performance model **`ParallelAxB`** describing the algorithm has 6 parameters. Parameter **`m`** specifies the number of abstract processors along the rows and columns of the processor grid executing the algorithm. Parameter **`r`** specifies the size of a square block of matrix elements, the updating of which is the unit of computation of the algorithm. Parameter **`n`** is the size of square matrices A, B, and C measured in $\mathbf{r} \times \mathbf{r}$ blocks. Parameter **`l`** is the size of a generalized block also measured in $\mathbf{r} \times \mathbf{r}$ block.

Vector parameter **`w`** specifies the widths of the rectangles of a generalized block assigned to different abstract processors of the $\mathbf{m} \times \mathbf{m}$ grid. The width of the rectangle assigned to processor P_{IJ} is given by element **`w[J]`** of the parameter (see Fig. 9). All widths are measured in $\mathbf{r} \times \mathbf{r}$ blocks.

Parameter **`h`** specifies the heights of rectangle areas of a generalized block of matrix A, which are horizontally communicated between different pairs of abstract processors. Let R_{IJ} and R_{KL} be the rectangles of a generalized block of matrix A assigned to processors P_{IJ} and P_{KL}, respectively. Then, **`h[I][J][K][L]`** gives the height of the rectangle area of R_{IJ}, which is required by processor P_{KL} to perform its computations. All heights are measured in $\mathbf{r} \times \mathbf{r}$ blocks.

Fig. 12 illustrates possible combinations of rectangles R_{IJ} and R_{KL} in a generalized block. Let us call an $\mathbf{r} \times \mathbf{r}$ block of R_{IJ} a *horizontal neighbour* of R_{KL} if the row of $\mathbf{r} \times \mathbf{r}$ blocks that contains this $\mathbf{r} \times \mathbf{r}$ block will also contain an $\mathbf{r} \times \mathbf{r}$ block of R_{KL}. Then, the rectangle area of R_{IJ}, which is required by processor P_{KL} to perform its computations, comprises of all horizontal neighbours of R_{KL}.

Fig. 12(a) shows the situation when rectangles R_{IJ} and R_{KL} have no horizontal neighbours. Correspondingly, **`h[I][J][K][L]`** will be zero.

Fig. 12(b) shows the situation when all $\mathbf{r} \times \mathbf{r}$ blocks of R_{IJ} are horizontal neighbours of R_{KL}. In that case, both **`h[I][J][K][L]`** will be equal to the height of R_{IJ}.

```
typedef struct {int I; int J;} Processor;
algorithm ParallelAxB(int m, int r, int n, int l, int w[m],
                      int h[m][m][m][m])
{
  coord I=m, J=m;
  node {I>=0 && J>=0: bench*(w[J]*(h[I][J][I][J])*(n/l)*(n/l)*n);};
  link (K=m, L=m)
  {
    I>=0 && J>=0 && I!=K :
      length*(w[J]*(h[I][J][I][J])*(n/l)*(n/l)*(r*r)*sizeof(double))
             [I, J] -> [K, J];
    I>=0 && J>=0 && J!=L && ((h[I][J][K][L])>0) :
      length*(w[J]*(h[I][J][K][L])*(n/l)*(n/l)*(r*r)*sizeof(double))
             [I, J] -> [K, L];
  };
  parent[0,0];
  scheme
  {
    int k, *w, *h, *trow, *tcol;
    Get_trow_tcol(m, w, h, trow, tcol);
    Processor Root, Receiver, Current;
    for(k = 0; k < n; k++)
    {
      int Acolumn = k%l, Arow;
      int Brow = k%l, Bcolumn;
      par(Arow = 0; Arow <l; )
      {
        Get_matrix_processor(Arow, Acolumn, p, q, w, h, trow, tcol, &Root);
        par(Receiver.I = 0; Receiver.I < m; Receiver.I++)
          par(Receiver.J = 0; Receiver.J < m; Receiver.J++)
            if((Root.I != Receiver.I || Root.J != Receiver.J) &&
               Root.J != Receiver.J)
              if((h[Root.I][Root.J][Receiver.I][Receiver.J]) > 0)
                (100.00/(w[Root.J]*(n/l)))%%
                      [Root.I, Root.J] -> [Receiver.I, Receiver.J];
        Arow += h[Root.I][Root.J][Root.I][ Root.J];
      }
      par(Bcolumn = 0; Bcolumn < l; )
      {
        Get_matrix_processor(Brow, Bcolumn, p, q, w, h, trow, tcol, &Root);
        par(Receiver.I = 0; Receiver.I < m; Receiver.I++)
          if(Root.I != Receiver.I)
            (100.00/((h[Root.I][Root.J][Root.I][Root.J])*(n/l))) %%
                  [Root.I, Root.J] -> [Receiver.I, Root.J];
        Bcolumn += w[Root.J];
      }
      par(Current.I = 0; Current.I < m; Current.I++)
        par(Current.J = 0; Current.J < m; Current.J++)
          (100.00/n) %% [Current.I, Current.J];
    }
  };
};
```

Fig. 11. Specification of the performance model of the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices in the HeteroMPI's performance definition language.

Figs. 12(c) and 12(d) shows the situation when only some of the **r** $\times$ **r** blocks of R_{IJ} are horizontal neighbours of R_{KL}. In this case, **h[I][J][K][L]** will be equal to the height of the rectangle subarea of R_{IJ} comprising the horizontal neighbours of R_{KL}.

Note that **h[I][J][I][J]** specifies the height of R_{IJ}, and **h[I][J][K][L]** will always be equal to **h[K][L][I][J]**.

The **coord** declaration introduces 2 coordinate variables, **I** and **J**, both ranging from **0** to **m** -1.

The **node** declaration associates the abstract processors with this coordinate system to form an **m** $\times$ **m** grid. It also describes the absolute volume of computation to be performed by each of the processors. As a unit of measure, the volume of computation performed by the code multiplying two **r** $\times$ **r** matrices is used. At each step of the algorithm, abstract processor P_{IJ} updates $(w_{IJ} \times h_{IJ}) \times n_g$ **r** $\times$ **r** blocks, where w_{IJ}, h_{IJ} are the width and height of the rectangle of a generalized block assigned to processor P_{IJ}, and n_g is the total number of generalized blocks. As computations during the updating of one **r** $\times$ **r** block mainly fall into the multiplication of two **r** $\times$ **r** blocks, the volume of computations performed by the processor P_{IJ} at each step of the algorithm will be approximately $(w_{IJ} \times h_{IJ}) \times n_g$ times larger than the volume of computations performed to multiply two **r** $\times$ **r** matrices. As w_{IJ} is given by **w[J]**, h_{IJ} is given by **h[I][J][I][J]**, n_g is given by **(n/l)** $*$ **(n/l)**, and the total number of steps of the algorithm

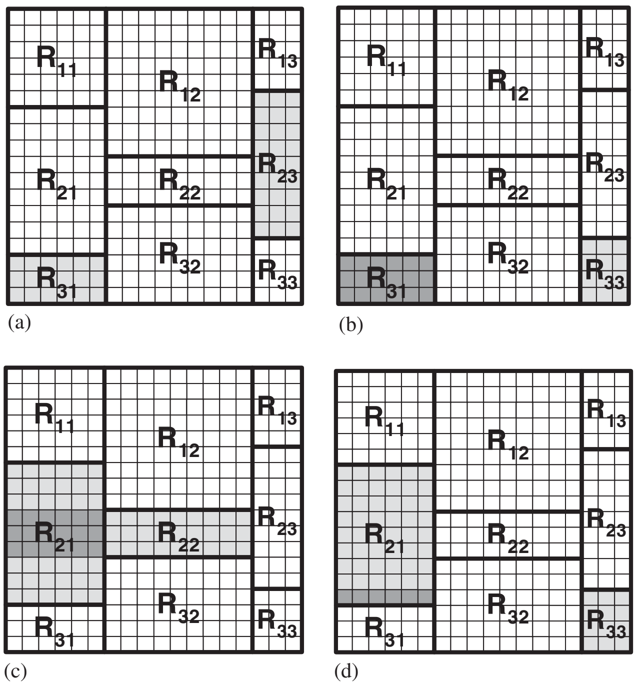

Fig. 12. Different combinations of rectangles in a generalized block. (a) No **r** × **r** block of rectangle R_{31} is a horizontal neighbour of rectangle R_{23}; therefore, $h[3][1][2][3] = 0$. (b) All **r** × **r** blocks of rectangle R_{31} are horizontal neighbours of rectangle R_{33}; $h[3][1][3][3] = 3$. (c) Neighbours of rectangle R_{22} in rectangle R_{21} make up a 3 × 6 rectangle area (shaded dark grey); $h[2][1][2][2] = 3$. (d) Neighbours of rectangle R_{33} in rectangle R_{21} make up the last row of this rectangle (shaded dark grey); $h[2][1][3][3] = 1$.

is given by **n** . Therefore the total volume of computation performed by abstract processor P_{IJ} will be **w[J] ∗h[I][J][I][J] ∗(n/l) ∗(n/l) ∗n** times bigger than the volume of computation performed by the code multiplying two **r** × **r** matrices.

The **link** declaration specifies the volumes of data to be transferred between the abstract processors during the execution of the algorithm. The first statement in this declaration describes communications related to matrix *A*. Obviously, abstract processors from the same column of the processor grid do not send each other elements of matrix *A*. Abstract processor P_{IJ} will send elements of matrix *A* to processor P_{KL} only if its rectangle R_{IJ} in a generalized block has horizontal neighbours of the rectangle R_{KL} assigned to processor P_{KL}. In that case, processor P_{IJ} will send all such neighbours to processor P_{KL}. Thus, in total processor P_{IJ} will send $N_{IJKL} \times n_g$ **r** × **r** blocks of matrix *A* to processor P_{KL}, where N_{IJKL} is the number of horizontal neighbours of rectangle R_{KL} in rectangle R_{IJ}, and n_g is the total number generalized blocks. N_{IJKL} is given by **w[J] ∗h[I][J][K][L]** , n_g is given by **(n/l) ∗(n/l)** , and the volume of data in one **r** × **r** block is given by **(r ∗r) ∗sizeof(double)** . Therefore the total volume of data transferred from processor P_{IJ} to processor P_{KL} will be given by **w[J] ∗h[I][J][K][L] ∗(n/l) ∗(n/l) ∗(r ∗r) ∗sizeof(double)** .

The second statement in the **link** declaration describes communications related to matrix *B*. Obviously, only abstract processors from the same column of the processor grid send each other elements of matrix *B*. In particular, processor P_{IJ} will send all its **r** × **r** blocks of matrix *B* to all other processors from column *J* of the processor grid. The total number of **r** × **r** blocks of matrix *B* assigned to processor P_{IJ} is given by **w[J] ∗h[I][J][I][J] ∗(n/l) ∗(n/l)** .

The **scheme** declaration describes **n** successive steps of the algorithm. At each step **k** ,

- A row of **r** × **r** blocks of matrix *B* is communicated vertically. For each pair of abstract processors P_{IJ} and P_{KJ} involved in this communication, P_{IJ} sends a part of this row to P_{KJ}. The number of **r** × **r** blocks transferred from P_{IJ} to P_{KJ} will be $w_{IJ} \times \sqrt{n_g}$, where $\sqrt{n_g}$ is the number of generalized blocks along the row of **r** × **r** blocks. The total number of **r** × **r** blocks of matrix *B*, which processor P_{IJ} sends to processor P_{KJ}, is $(w_{IJ} \times h_{IJ}) \times n_g$. Therefore, $\frac{w_{IJ}\times\sqrt{n_g}}{(w_{IJ}\times h_{IJ})\times n_g} \times 100 = \frac{1}{h_{IJ}\times\sqrt{n_g}} \times 100$ percent of all data that should be sent from processor P_{IJ} to processor P_{KJ} will be sent at the step. The first nested **par** statement in the main **for** loop of the **scheme** declaration just specifies this fact. The **par** algorithmic patterns are used to specify that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel;
- A column of **r** × **r** blocks of matrix *A* is communicated horizontally. If processors P_{IJ} and P_{KL} are involved in this communication so that P_{IJ} sends a part of this column to P_{KL}, then the number of **r** × **r** blocks transferred from P_{IJ} to P_{KL} will be $H_{IJKL} \times \sqrt{n_g}$, where H_{IJKL} is the height of the rectangle area in a generalized block, which is communicated from P_{IJ} to P_{KL}, and $\sqrt{n_g}$ is the number of generalized blocks along the column of **r** × **r** blocks. The total number of *r* × *r* blocks of matrix *A*, which processor P_{IJ}

```
int m, l, r, n;
int main(int argc, char** argv) {
   int optimal_generalised_block_size;
   int *w, *h; //Matrix partitioning parameters
   typedef struct {double *a; double *b; double *c; int r;}
           Recon_params;
   HMPI_Group gid;
   void *model_params;
   int param_count = 4+m+(m*m*m*m);
   double *a, *b, *c;
   HMPI_Init(argc, argv);
   if (HMPI_Is_member(HMPI_COMM_WORLD_GROUP)) {
      int output_p;
      Recon_params recon_params;
      Initialize(a, b, c, r, &recon_params);
      HMPI_Recon(&rMxM, &recon_params,  1, &output_p);
   }
   if (HMPI_Is_host()) {
      int bsize;
      double time, min_time=DBL_MAX;
      for (bsize = m; bsize < n; bsize++) {
          Partition_matrix(n, bsize, r, w, h);
          HMPI_Pack_model_parameters(p, q, n, bsize, r, w, h, model_params);
          time = HMPI_Timeof(&HMPI_Model_ParallelAxB,
                             model_params);
          if (time < min_time) {
             optimal_generalised_block_size = bsize;
             min_time = time;
          }
      }
   }
   …
   l = optimal_generalised_block_size;
   if (HMPI_Is_host()) {
      HMPI_Pack_model_parameters(p, q, n, l, r, w, h, model_params);
      HMPI_Group_create(&gid, &HMPI_Model_ParallelAxB, model_params);
   }
   if (HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_ParallelAxB, NULL);
   if (HMPI_Is_member(&gid)) {
      …
      MPI_Comm* grid_comm = (MPI_Comm*)HMPI_Get_comm(&gid);
      …
      // computations and communications are performed here
      // using standard MPI routines.
      //
      …
   }
   if (HMPI_Is_member(&gid)) HMPI_Group_free(&gid);
   HMPI_Finalize(0);
}
```

Fig. 13. The core of the HeteroMPI program implementing the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices.

sends to processor P_{KL}, is $N_{IJKL} \times n_{\mathrm{g}}$. Therefore, $\frac{H_{IJKL} \times \sqrt{n_{\mathrm{g}}}}{N_{IJKL} \times n_{\mathrm{g}}} \times 100 = \frac{H_{IJKL} \times \sqrt{n_{\mathrm{g}}}}{\left(H_{IJKL} \times w_{IJ}\right) \times n_{\mathrm{g}}} \times 100 = \frac{1}{w_{IJ} \times \sqrt{n_{\mathrm{g}}}} \times 100$ percent of all data that should be sent from processor P_{IJ} to processor P_{KL} will be sent at the step. The second nested **par** statement in the main **for** loop of the **scheme** declaration specifies this fact. Again, we use the **par** algorithmic patterns in this specification to stress that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel;

- Each abstract processor updates each of its **r** $\times$ **r** block of matrix C with one block from the pivot column and one block from the pivot row, so that each block c_{ij} $(i, j \in \{1, \ldots, n\})$ of matrix C will be updated to have the values $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. The processor performs the same volume of computation at each step of the algorithm. Therefore, at each of **n** steps of the algorithm the processor will perform $\frac{100}{n}$ percent of the volume of computations it performs during the execution of the algorithm. The third nested **par** statement in the main **for** loop of the **scheme** declaration just specifies this fact. The **par** algorithmic patterns are used here to specify that all abstract processors perform their computations in parallel.

Function **GetProcessor** is used in the **scheme** declaration to iterate over abstract processors that store the pivot row and the pivot column of **r** $\times$ **r** blocks. It returns in its last parameter the grid coordinates of the abstract processor storing the **r** $\times$ **r** block, whose coordinates in a generalized block of a matrix are specified by its first two parameters.

The most interesting fragments of the rest code of the HeteroMPI parallel application are shown in Fig. 13. The HeteroMPI runtime system is initialized using operation **`HMPI_Init`** . Then, operation **`HMPI_Recon`** updates the estimation of performances of processors using the serial multiplication of test matrices of size **`r`** × **`r`**. The computations performed by each processor mainly fall into the execution of calls to function **`rMxM`**.

The next block of code, executed by the host-processor, uses the operation **`HMPI_Timeof`** predicting the total time of execution of the parallel algorithm. This operation is used to calculate the optimal generalized block size, one of the parameters of the heterogeneous parallel algorithm. This is followed by the creation of a group of processes using operation **`HMPI_Group_create`** . The members of this group then perform the computations and communications of the heterogeneous parallel algorithm using standard MPI means. This is followed by freeing the group using operation **`HMPI_Group_free`** and the finalization of HeteroMPI runtime system using operation **`HMPI_Finalize`** .

6. Experiments with HeteroMPI

This section presents some results of experiments with the HeteroMPI applications presented in Sections 4 and 5. Note that the figures showing the performances of the computers in the tables in this section give the average speeds measured at runtime during the experiments. The computers used in the experiments are connected to a communication network, which is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers. The experimental results are obtained by averaging the execution times over a number of experiments.

A small heterogeneous local network of 9 different FreeBSD, Solaris, and Linux workstations shown in Table 1 is used in the experiments presented in Figs. 14–16. We measure the relative speeds with the core computation of the algorithm (updating of a matrix). Note that the relative speed does not depend on the size of problem for the wide range of matrix sizes used in our experiments. The relative speeds measured for this network are shown in Table 1. One can see that in the case of matrix–matrix multiplication, the fastest computer `afflatus` is almost 10 times faster than the slowest computer `csultra01`.

Fig. 14(a) shows the comparison of the execution times of the HeteroMPI application and the standard MPI application executing the EM3D algorithm. Fig. 14(b) demonstrates the speedup of the HeteroMPI program over the MPI one. In this case the HeteroMPI application is almost 1.7 times faster than the standard MPI one. Fig. 15(a) shows the comparison of the execution times of the MM algorithm between the HeteroMPI application and the standard MPI application using a homogeneous two-dimensional block-cyclic data distribution. Fig. 15(b) demonstrates the speedup of the HeteroMPI program over MPI. The results are obtained for the value of **`r`** equal to **16** and the optimal value of the size of generalized block **`l`** , which is equal to **96**. Using HeteroMPI the application is almost 8 times faster than using standard MPI.

Experiments shown in Fig. 16 compare the efficiency of the HeteroMPI application executing the MM algorithm on a network of nine heterogeneous workstations to the efficiency of the MPI application using homogeneous two-dimensional block-cyclic data distribution executed on a network of 9 identical workstations. The relative speeds of the workstations in the heterogeneous network are 26, 20, 14, 14, 14, 14, 14, 9, and 1. The workstations in the homogeneous network have a same relative speed of 14. The two sets share 5 workstations (of the speed 14) and belong to the same homogeneous communication segment of the local network. The sets were selected so that the aggregate performance of the processors of the heterogeneous network is practically the same as that of the homogeneous one. Thus we compare the efficiency demonstrated by the heterogeneous algorithm on the heterogeneous

Table 1
Specifications of the nine heterogeneous processors

Machine name (number of processors)	Architecture	CPU MHz	Main memory (MB)	Cache (KB)	Relative speed (m × m)
afflatus (1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor supporting HT† technology	2867	2048	1024	499
aries2 (1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor	2457	512	1024	384
Pg1 cluster01(2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269
Pg1 cluster02 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269
Pg1 cluster03 (1)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269
linserver (1)	Linus 2.4.20-20.9bigmem Intel(R) Xeon(TM)	2783	7748	512	172
csultra01 (1)	SunOS 5.8 sun4u sparc SUNW, Ultra-5_10	440	512	2048	46

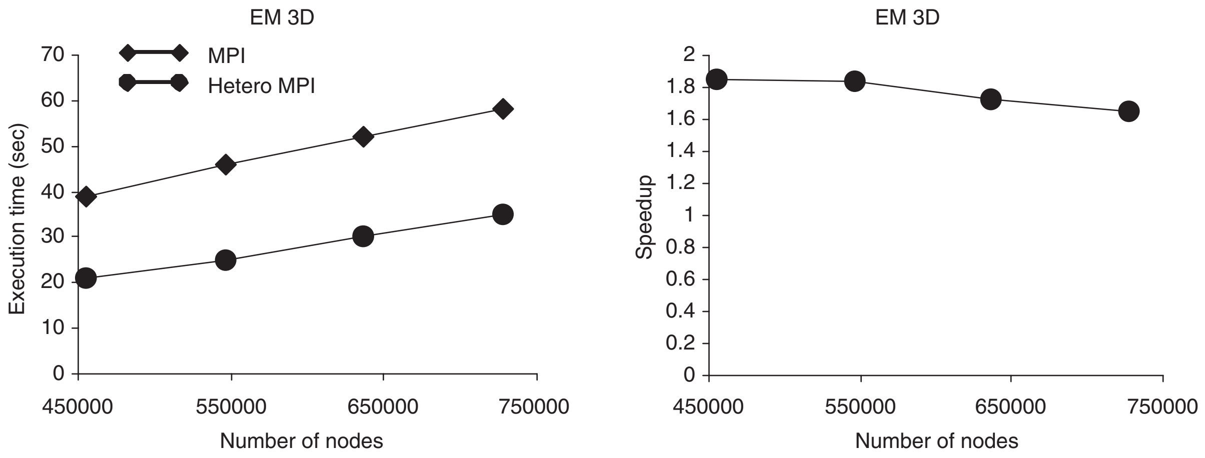

Fig. 14. Results obtained using the heterogeneous network of computers shown in Table 1; (a) comparison of execution times of the EM3D algorithm between HeteroMPI and MPI; (b) the speedup of EM3D algorithm obtained using HeteroMPI over MPI.

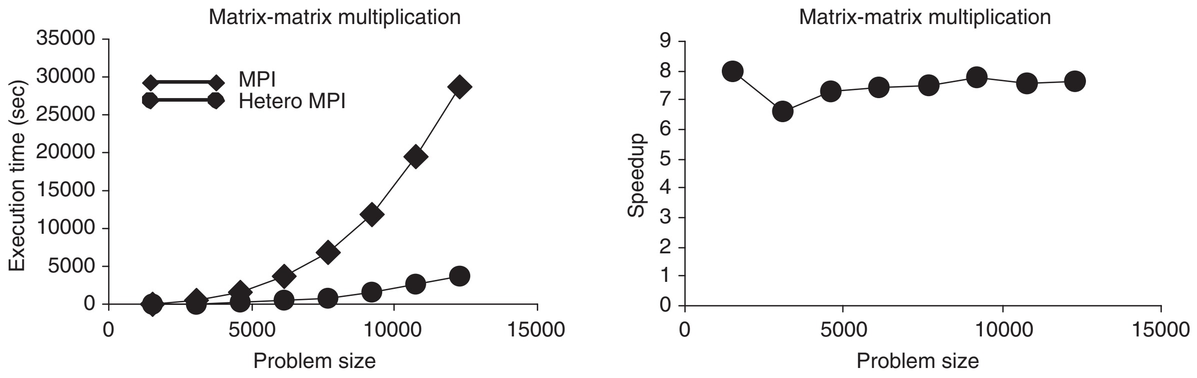

Fig. 15. Results obtained using the heterogeneous network of computers shown in Table 1; (a) a comparison of the execution times of the MM algorithm using HeteroMPI and MPI; (b) the speedup of the MM algorithm obtained using HeteroMPI over MPI.

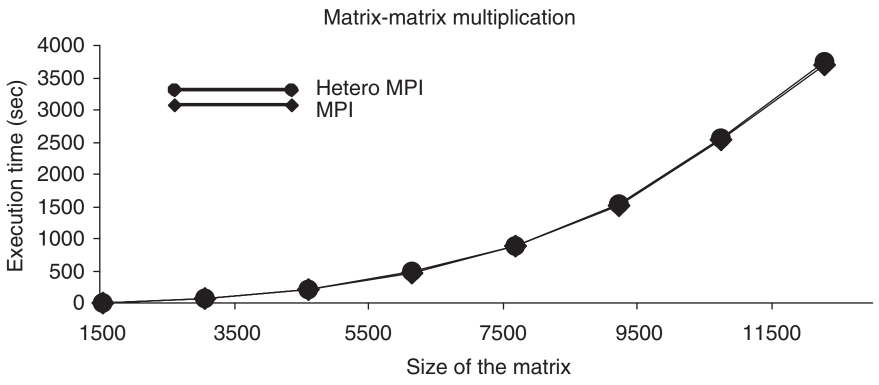

Fig. 16. Execution times of the HeteroMPI application on the heterogeneous network and the MPI application on the homogeneous network. The two networks have approximately the same aggregate power of processors and share the same (homogeneous) communication network.

network with the efficiency demonstrated by its homogeneous prototype on the homogeneous network having the same aggregate performance as the heterogeneous one. From the figure, it can be seen that the applications demonstrate practically the same speed, but each on its network. As the two networks are practically of the same power, we can conclude that the HeteroMPI application cannot perform better and its efficiency is close to optimal on such a heterogeneous network of computers. This approach to analysis of the performance of heterogeneous algorithms is presented in more detail in [17].

7. Related work

The section surveys related papers from the literature. The papers surveyed are mainly:

- papers presenting extensions to MPI and runtime systems that distribute data efficiently and automatically when there are changes in the application or the underlying environment;
- papers presenting implementations of MPI that adopt runtime adaptation schemes to find an efficient data distribution when workload and communication characteristics of a program change at runtime;
- papers presenting a combined approach of compile-time analysis, runtime load distribution, and cooperation of operating system scheduler for improved utilization of resources on HNOCs.

Dyn-MPI [22] extends MPI by providing specialized facilities for memory allocation, communication, and node participation. The key component of Dyn-MPI is its run-time system, which efficiently and automatically redistributes data on the fly when there are changes in the application or the underlying environment. Dyn-MPI also provides a facility for removing nodes from the computation when their participation degrades performance. However, converting an MPI program to a Dyn-MPI program can require major modifications for it to run efficiently on HNOCs. These include allocations functions for all potentially re-distributable arrays and determination of Deferred Regular Section Descriptors (DRSDs) for each array.

AMPI [4,18] is an implementation of a significant subset of MPI 1.1 Standard over CHARM++ [13]. AMPI utilizes the dynamic load-balancing capabilities of CHARM++ by associating a "user-level" thread with each CHARM++ migratable object. User's code runs inside this thread, so that it can issue blocking receive calls similar to MPI, and still present the underlying scheduler an opportunity to schedule other computations on the same processor. The runtime system keeps track of the computation loads of each thread as well as the communication graph between AMPI threads, and can migrate these threads in order to balance the overall load while simultaneously minimizing the communication overhead. However, converting an MPI program to an AMPI program can require major modifications for it to run efficiently on HNOCs. These include privatization of global variables, registering chunk data and providing a packing subroutine to the AMPI runtime system to pack the thread's data, and the migration decision has to be made by the user through a call to migration subroutine even though the actual migration of the chunk is done by the system's load balancing strategy.

Tern [15] is an implementation of a subset of the MPI standard for message passing parallel programming, and augments the MPI standard to provide for multithreaded execution. It provides the ability to transparently migrate threads between the nodes executing the parallel application thus achieving parallel program performance improvement through load balance and improved fault tolerance. However in a threaded environment, static and global variables will be shared among all threads executing in the same process and so need to be made thread-specific. Tern provides a compiler directive prefix that is required to be added to each global variable and static-local variable declaration. Tern exposes the policy mechanisms used to guide migration decisions to the user, allowing for customizable thread migration policies. Tern provides two mechanisms for safely migrating a user thread. The first mechanism allows the user to insert migration calls in the user thread. However this mechanism requires that the user knows exactly where to insert the migration calls. Alternatively, Tern runtime system migrates the thread depending on the policy mechanisms devised by the user.

CRAUL [21] is a runtime system that combines compile-time analysis, runtime load balancing and locality considerations, and cooperative scheduling support from the operating system for improved performance of parallel programs on HNOCs. However, the CRAUL compiler lacks the more complex translation mechanisms essential to extract parallelism from less easily analyzable loops. In such cases, the user needs to insert the required data structures manually into an already parallelized program. The problem of portability and reusability of the parallel code generated is not addressed. Also CRAUL does not provide features required to capture programmer's knowledge of an application to the level necessary to automatically provide an efficient implementation on a heterogeneous system. CRAUL shields the user from data distribution details but does not supply mechanisms for the user to guide the mapping process essential for exploiting his or her knowledge of the application.

8. Conclusion

In this paper, we have presented HeteroMPI, an extension of MPI for programming high-performance computations on heterogeneous networks of computers. The main idea of HeteroMPI is to automate the process of selection of a group of processes, which would execute the heterogeneous algorithm faster than any other group. HeteroMPI provides features that allow the user to carefully design their parallel applications that can run efficiently on HNOCs. The features that affect the efficiency of the process of selection are:

- The accuracy of the performance model designed by the application programmers to describe their implemented heterogeneous algorithm. The performance model definition language is used to describe their implemented heterogeneous algorithm. It provides comprehensive features to express many scientific parallel applications. These features allow the application programmers to design all types of performance models ranging from the simplest to the most complicated, and not the very accurate to the most accurate.

- The accuracy of **`HMPI_Recon`** . The accuracy of **`HMPI_Recon`** depends upon how accurately the benchmark code provided by the application programmers reflects the core computations of each phase of their parallel applications. If the benchmark code provided is an accurate measurement of the core computations in each phase, **`HMPI_Recon`** gives an accurate measure of the speeds.
- The accuracy of **`HMPI_Timeof`** . The accuracy of the estimation by **`HMPI_Timeof`** is dependent upon the following:
  - the accuracy of the performance model,
  - the quality of the heuristics designed for the set of parameters provided to the performance model,
  - the accuracy of the model of the executing network of computers. This depends on the accuracy of the measurements of the processor speeds given by **`HMPI_Recon`** and the communication model of the executing network of computers. Currently the communication model used in HeteroMPI runtime system is static. Future works would address the issue of efficiently updating the parameters of communication model at runtime.

Thus HeteroMPI provides all the features to the user to write portable and efficient parallel applications on HNOCs. Experimental results show that carefully designed HeteroMPI applications can show very good improvements in execution performance on HNOCs.

We consider the HeteroMPI as a step towards a future standard message-passing library for heterogeneous networks of computers. This library is viewed as such an extension of the standard MPI that combines the features of multi-protocol communication, fault tolerance, and the advanced support for efficient heterogeneous parallel computing, separately provided by the Nexus MPI, the FT-MPI, and the HeteroMPI.

References

[1] O. Aumage, L. Bouge, R. Namyst, A portable and adaptative multi-protocol communication library for multithreaded runtime systems, in: Proceedings of the Fourth Workshop on Runtime Systems for Parallel Programming (RTSPP ’00), Lecture Notes in Computer Science, vol. 1800, Springer, Cancun, Mexico, 2000, pp. 1136–1143.

[2] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y.S. Dandass, M. Apte, MPI/FT: architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing, in: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, Brisbane, Australia, May 2001.

[3] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, Y. Robert, A proposal for a heterogeneous cluster ScaLAPACK (Dense Linear Solvers), IEEE Trans. Comput. 50 (10) (October 2001) 1052–1070.

[4] M. Bhandarkar, L.V. Kale, E. de Sturler, J. Hoeflinger, object-based adaptive load balancing for MPI programs, in: Proceedings of the International Conference on Computational Science, San Francisco, CA, Lecture Notes in Computer Science, vol. 2074, Springer, May 2001, pp. 108–117.

[5] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK: a portable linear algebra library for distributed memory computers-design issues and performance, Comput. Phys. Commun. 97 (1996) 1–15.

[6] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, K. Yelick, Parallel Programming in Split-C, In: Proceedings of Supercomputing ’93, Portland, Oregon, November 1993, pp. 262–273.

[7] R. Dimitrov, A. Skjellum, An efficient MPI implementation for virtual interface architecture—enabled cluster computing, in: Proceedings of the third MPI Developer’s and User’s Conference, Atlanta, Georgia, March 1999, pp. 15–24.

[8] R. Dimitrov, Overlapping of communication and computation and early binding: fundamental mechanisms for improving parallel performance on clusters of workstations. Ph.D. Dissertation, Department of Computer Science, Mississippi State University, May 2001.

[9] J. Dongarra, S. Huss-Lederman, S. Otto, M. Snir, D. Walker, MPI: The Complete Reference, The MIT Press, Cambridge, MA, 1996.

[10] G.E. Fagg, A. Bukovsky, J. Dongarra, HARNESS and fault tolerant MPI, in: Parallel Comput. 27 (11) (October 2001) 1479–1495.

[11] I. Foster, J. Geisler, C. Kesselman, S. Tuecke, Managing multiple communication methods in high-performance networked computing systems, Journal Parallel Distrib. Comput. 40 (1) (January 1997) 35–48.

[12] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Comput. 22 (6) (September 1996) 789–828.

[13] L.V. Kalé, S. Krishnan, CHARM++: a portable concurrent object oriented system based on C++, in: Proceedings of OOPSLA’93, ACM Press, New York, 1993, pp. 91–108.

[14] A. Kalinov, A. Lastovetsky, Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers, Journal Parallel Distrib. Comput. 61 (4) (April 2001) 520–535.

[15] J. Ke, E. Speight, Tern: migrating threads in an MPI runtime environment, Technical Report, SCL-TR-2001-1016, Cornell, November, 2001.

[16] A. Lastovetsky, Adaptive parallel computing on heterogeneous networks with mpC, Parallel Comput. 28 (10) (October 2002) 1369–1407.

[17] A. Lastovetsky, R. Reddy, On performance analysis of heterogeneous parallel algorithms, Parallel Comput. 30 (11) (2004) 1195–1216.

[18] O. Lawlor, M. Bhandarkar, L.V. Kale, Adaptive MPI, Technical Report, TR 02-05, University of Illinois, 2002.

[19] S. Louca, N. Neophytou, A. Lachanas, P. Evripidou, MPI-FT: portable fault tolerance scheme for MPI, Parallel Process. Lett. 10 (4) (December 2000) 371–382.

[20] J. Marinho, J.G. Silva, WMPI—Message Passing Interface for Win32 Clusters, in: Proceedings of Fifth European PVM/MPI User’s Group Meeting, September 1998, pp. 113–129.

[21] U. Rencuzogullari, S. Dwarkadas, Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations, in Eighth ACM PPOPP, June 2001, pp. 72–81.

[22] D.B. Weatherly, D.K. Lowenthal, M. Nakazawa, F. Lowenthal, Dyn-MPI: supporting MPI on a nondedicated cluster of workstations, in: Proceedings of the 15th IEEE/ACM Supercomputing 2003 (SC’03), Phoenix, AZ, April 2003.

[23] K. Yelick, C. Wen, S. Chakrabarti, E. Deprit, J.A. Jones, A. Krishnamurthy, Portable parallel irregular applications, in: Workshop on Parallel Symbolic Languages and Systems, Lecture Notes in Computer Science, 1995.

Alexey Lastovetsky received a Ph.D. from the Moscow Aviation Institute in 1986, and a Doctor of Sciences degree from the Russian Academy of Sciences in 1997. His main research interests include algorithms, models and programming tools for high performance heterogeneous computing. He published over 60 technical papers in refereed journals and international conferences. He authored the monograph “Parallel computing on heterogeneous networks” published by Wiley in 2003. He is currently a lecturer in the Department of Computer Science at University College Dublin, National University of Ireland. At UCD, he also leads Heterogeneous Computing Laboratory. He is on the editorial boards of the research journals “Parallel Computing” and “Programming and Computer Software”.

Ravi Reddy is currently a Ph.D. student in the Department of Computer Science at University College Dublin, National University of Ireland. His main research interests are design of algorithms and tools for parallel and distributed computing systems.