HMPI: 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
E-mail: Alexey.Lastovetsky@ucd.ie, Manumachu.Reddy@ucd.ie

Abstract

The paper presents Heterogeneous MPI (HMPI), 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. This model allows for all the main features of the underlying parallel algorithm, which have an impact on its execution performance, such as 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 the description of the performance model, HMPI creates a group of processes executing the algorithm faster than any other group of processes. The most principal extensions to MPI are presented. Parallel simulation of the interaction of electric and magnetic fields and parallel matrix multiplication are used to demonstrate the features of the library.

1 Introduction

The standard MPI [1] 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 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, which is difficult because the processors may demonstrate different speeds for different applications due to differences in the set of instructions, 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 speed 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 HNOC much more difficult than across a dedicated cluster of workstations interconnected with a homogeneous high-performance communication network. Other 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 independently by its users. The first implication with the multi-user decentralized nature of HNOCs is the unstable performance characteristics of processors during the execution of a parallel program as the computers may be used for other computations and communications. The second implication is the 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 does not provide means for employment of 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. The only exception is the use of shared memory and TCP/IP in MPICH [2]. At the same time, there have been some research efforts to address this challenge implicitly, via

advanced non-standard implementations of the standard MPI specification (Nexus [3], Madeleine [4]).

Secondly, the standard MPI does not provide means for the writing of fault-tolerant parallel applications for HNOCs. There are some research efforts made recently to address this challenge such as the fault-tolerant MPI (FT-MPI) [5]. FT-MPI is a small set of extensions to MPI and its research implementation, which are aimed at the writing of message-passing programs that can survive failures. It offers the application programmer a range of recovery options other than just returning to some previous checkpoint.

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 HMPI (Heterogeneous MPI), aimed at efficient parallel computing on HNOCs, and its research implementation.

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

2 Outline of HMPI

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 speed 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 speeds 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 speeds 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 HMPI is to automate the process of selection of such a group of processes that executes the heterogeneous algorithm faster than any other group. HMPI allows the application programmers to describe a performance model of their implemented heterogeneous algorithm. This model allows for 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, and
- 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.

HMPI 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 [7]. 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 HMPI runtime system.

Having provided such a description of the performance model, application programmers can use a new operation, whose interface is shown below, to create a group that will 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,
        int param_count)
```

where ***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), and ***param_count*** is the number of parameters of the performance model. This function returns an HMPI handle to the group of MPI processes in ***gid***.

In HMPI, groups are not absolutely independent on 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 HMPI group.

During the creation of this group of processes, HMPI 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 model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

The algorithms used to solve the problem of selection of processes are discussed in [7]. The accuracy of the model of the executing network of computers depends upon the accuracy of the estimation of the actual speeds of processors. HMPI 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 that 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,

```
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 HMPI.

Another principal operation provided by HMPI 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, int param_count)
```

This function allows the application programmers to write such a parallel application that can follow 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 associated with the predefined communication universe **HMPI_COMM_WORLD** of HMPI.

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

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

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

After the initialization, application programmers can call any other HMPI 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 HMPI instead of **MPI_COMM_WORLD** of MPI.

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

The only group constructor operation provided by HMPI is the creation of the group using **HMPI_Group_create**, and the only group destructor operation provided by HMPI is

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

where ***gid*** is the HMPI 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 HMPI. This is because:

- Firstly, HMPI does not guarantee that groups composed using these operations can execute a logical unit of parallel algorithm faster than any other group of processes, and

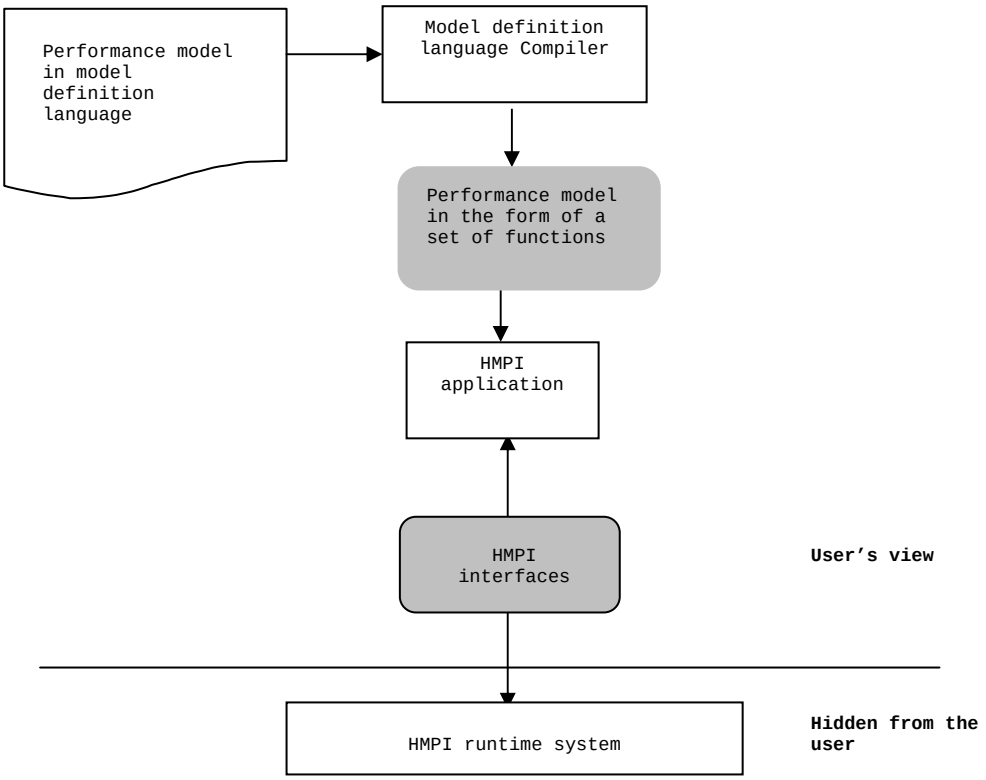

Figure 1. Development process of a HMPI application. To build HMPI applications, an application programmer describes a performance model using the model definition language, compiles the performance model description into a set of functions, writes the application using the HMPI interfaces to create groups of processes to execute the parallel algorithm.

- 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 HMPI apart from the group constructor and destructor are the following group accessors:

- **HMPI_Group_rank** to get the rank of the process in the HMPI group, and
- **HMPI_Group_size** to get the number of processes in this group.

The initialization of HMPI 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**,
- Execution of the parallel algorithm by the members of the group. At this point, control is handed over to MPI. MPI and HMPI are interconnected by 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 HMPI groups with **HMPI_Group_free**.
- Finalizing the HMPI runtime system by using operation

  HMPI_Finalize (**int** exitcode),

Note, that in general, the architecture of HMPI summarised in Figure 1 has similarities to the architectural framework of the CORBA specification [9].

3 Example of irregular HMPI application

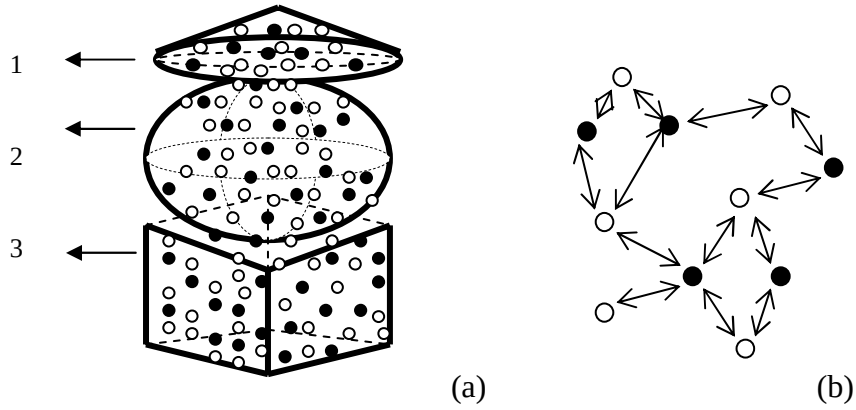

Figure 2. (a) The three dimensional object consists of three subbodies. In each subbody, the electric field value is represented as a white dot, an E node, and the magnetic field value represented by a black dot, an H node and (b) A bipartite graph showing dependencies between E and H nodes.

To explain how an application programmer can use HMPI to write a real-life irregular application, consider the EM3D application simulating the interaction of electric and magnetic fields on a three-dimensional object [11, 12]. 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 neighboring H nodes and vice versa. Thus, the dependencies between E and H nodes form a bipartite graph. In a bipartite graph, the vertices are decomposed into two disjoint sets such that no two vertices 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 Figure 2(a). A simple example of bipartite graph is shown in Figure 2(b).

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

At each step of the algorithm,

- For each of the E nodes in its sub body, if any of the neighboring 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 sub body;
- For each of the H nodes in its sub body, if any of the neighboring 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 sub body;

The most interesting fragments of the MPI version of this parallel application are shown in Figure 3.

We assume the one-process-per-processor configuration for this MPI application.

As shown in the MPI program in Figure 3, 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.

The HMPI version of this parallel application involves first describing the performance model of the

```
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();
}
```

Figure 3. The most principal code of the MPI program implementing the EM3D algorithm.

parallel algorithm. The definition of **Em3d** shown in Figure 4 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 ofabstract processors 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 abstract processor 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 abstract processors 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 processors. As a unit of measurement, the volume of computation performed by some benchmark code is used. In this particular case, it is assumed that the benchmark code computes 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.

The **link** declaration specifies the volumes of data to be transferred between the abstract processors at each step of the algorithm. Abstract processor 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 processor P_{remote} to processor P_{owner} at this step will be sent. The second nested **par** statement in the main **for** loop of the **scheme** declaration just specifies it. 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.
- Each processor then computes the new values for each of the nodes in its subbody. The processor will perform 100% of computations it should perform during this iteration. The **par**

```
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];
  };
```

Figure 4. Specification of the performance model of the em3d algorithm in the HMPI's performance definition language.

algorithmic patterns are used here to specify that all abstract processors 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 processor performs the same volume of computations, and the same volume of data is transferred between each pair of processors.

The most interesting fragments of the rest code of the HMPI parallel application are shown in Figure 5.

In the example shown in figure 5, the HMPI 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 HMPI runtime system using operation **HMPI_Finalize**.

On HNOCs, the running time of the HMPI program shown above will always be less than the running time of the corresponding MPI program. This is because an HMPI group of processes will always 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 HMPI group are chosen optimally taking into account all the main features of the underlying parallel algorithm, 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 sub-body thus depending on the number of nodes within a sub-body,
- 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 sub-bodies, and
- 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 HMPI 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 heterogeneous network.

It can also be seen from the MPI and HMPI 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.

```
int main(int argc, char **argv) {
   MPI_Comm em3dcomm;
   int i, me, k, E = 0, H = 1;
   HMPI_Group gid;
   void* model_params;
   int param_count;
   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, &param_count);
   if (HMPI_Is_host() || HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_Em3d,
                        model_params, param_count);
   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);
}
```

Figure 5. The most principal code of the HMPI program implementing the algorithm of EM3D.

4 Example of regular HMPI 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 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 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 efficient solving a regular problem is to reduce it to such an irregular problem, the structure of which is determined by the irregularity of underlying hardware rather than the irregularity of the problem itself.

Consider the problem of parallel matrix multiplication (MM) on HNOCs. The algorithm of execution of the matrix operation $C=A\times B$ on a HNOC is obtained by modification of the ScaLAPACK [8] 2D block-cyclic MM algorithm. The modification is that the heterogeneous 2D block-cyclic data distribution of [6] is used instead of the standard homogeneous data distribution. Thus, the heterogeneous algorithm of multiplication of two dense square $(n\times r)\times(n\times r)$ matrices A and B on an $m\times m$ grid of heterogeneous processors can be summarised as follows:

- Each element in A, B, and C is a square r×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 (l×r)×(l×r), where $m \le l \le n$. The generalized blocks are identically partitioned into 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 $r \times r$ block of matrix elements. The generalized block is a $l \times l$ square of $r \times r$ blocks.
  - First, the $l \times l$ square is partitioned into m vertical slices, so that the area of the j-th slice is proportional to $\sum_{i=1}^{m} s_{ij}$. It is supposed that blocks of the j-th slice will be assigned to processors of the j-th column in the $m \times m$ processor grid. Thus, at this step, we balance the load between processor columns in the $m \times m$ 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 i-th horizontal slice in the j-th vertical slice is proportional to s_{ij}. It is supposed that blocks of the i-th horizontal slice in the j-th 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 r×r block a_{ik} of the pivot column of matrix A is sent horizontally from the processor, which stores this block, to m-1 processors (see Figure 6);
  - Each r×r block b_{kj} of the pivot row of matrix B is sent vertically from the processor, which stores this block, to m-1 processors (see Figure 6);
- Each processor updates its rectangle in the C matrix with one block from the pivot row and one block from the pivot column.

The definition of **ParallelAxB** given in Figure 7 describes the performance model of this heterogeneous algorithm.

The network type **ParallelAxB** describing the algorithm has 6 parameters. Parameter **m** specifies the number of abstract processors along the row and along the column 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 $r \times r$ blocks. Parameter l is the size of a generalised block also measured in $r \times r$ block.

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

Parameter **h** specifies the heights of rectangle areas of a generalised 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 generalised 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 $r \times r$ blocks.

Note that **h[I][J][I][J]** specifies the height of R_{IJ}, and **h[I][J][K][L]** will be always 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 a $m \times m$ 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 generalised block assigned to processor P_{IJ}, and n_g is the total number of generalised 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 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 is given by n, 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 $r \times r$ matrices.

The **link** declaration specifies the volumes of data to be transferred between the abstract processors

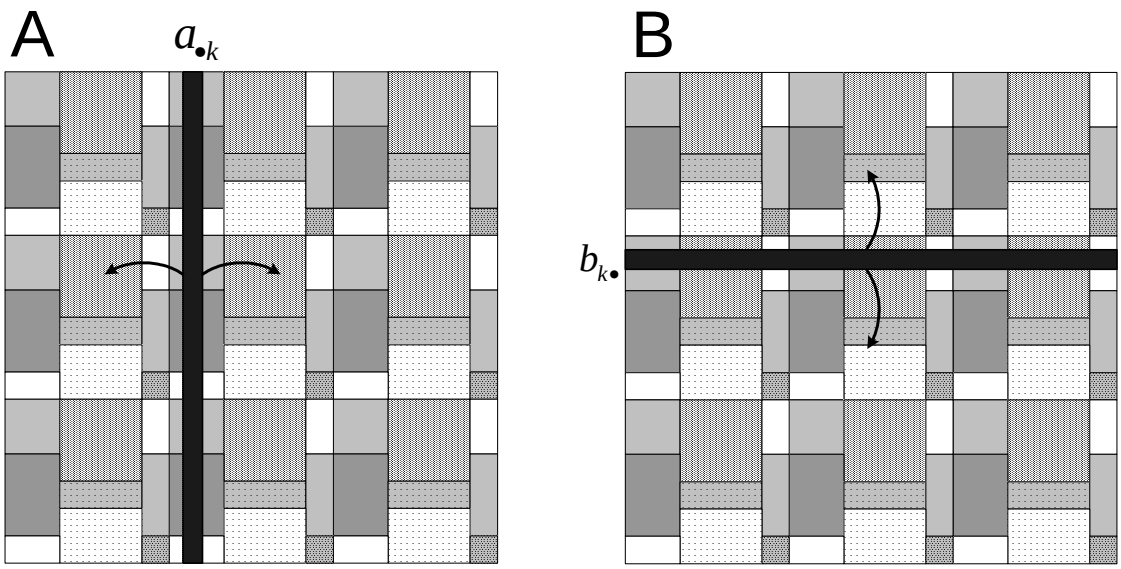

Figure 6. One step of the algorithm of parallel matrix-matrix multiplication based on heterogeneous two-dimensional block distribution of matrices A, B, and C. First, each $r \times r$ block of the pivot column $a_{\bullet k}$ of matrix A (shown shaded dark grey) is broadcast horizontally, and each $r \times r$ block of the pivot row $b_{k\bullet}$ of matrix B (shown shaded dark grey) is broadcast vertically.

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 generalised 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 $N_{IJKL} \times n_g$ $r \times r$ 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 generalised blocks. As 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 $r \times r$ block is given by **(r*r)*sizeof(double)**, 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 $r \times r$ blocks of matrix *B* to all other processors from column *J* of the processor grid. The total number of $r \times r$ 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 $r \times r$ 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 $r \times r$ blocks transferred from P_{IJ} to P_{KJ} will be $w_{IJ} \times \sqrt{n_g}$, where $\sqrt{n_g}$ is the number of generalised blocks along the row of $r \times r$ blocks. The total number of $r \times r$ 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 data that should be in total 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 it. 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 $r \times r$ 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 $r \times r$ 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 generalised block, which is communicated

```
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[I]*(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;
    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; )
      {
        GetProcessor(Arow, Acolumn, m, h, w, &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/(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; )
      {
        GetProcessor(Brow, Bcolumn, m, h, w, &Root);
        par(Receiver.I = 0; Receiver.I < m; Receiver.I++)
          if(Root.I != Receiver.I)
            (100/((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/n) %% [Current.I, Current.J];
    }
  };
};
```

Figure 7. Specification of the performance model of the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices in the HMPI's performance definition language.

```
int m, l;
int main(int argc, char** argv) {
   int optimal_generalised_block_size;
   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++) {
          time = HMPI_Timeof(&HMPI_Model_ParallelAxB,
                             model_params, param_count);
          if (time < min_time) {
             optimal_generalised_block_size = bsize;
             min_time = time;
          }
      }
   }
   …
   l = optimal_generalised_block_size;
   if (HMPI_Is_host() || HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_ParallelAxB,
                        model_params, param_count);
   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);
}
```

Figure 8. The most principal code of the HMPI program implementing the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices.

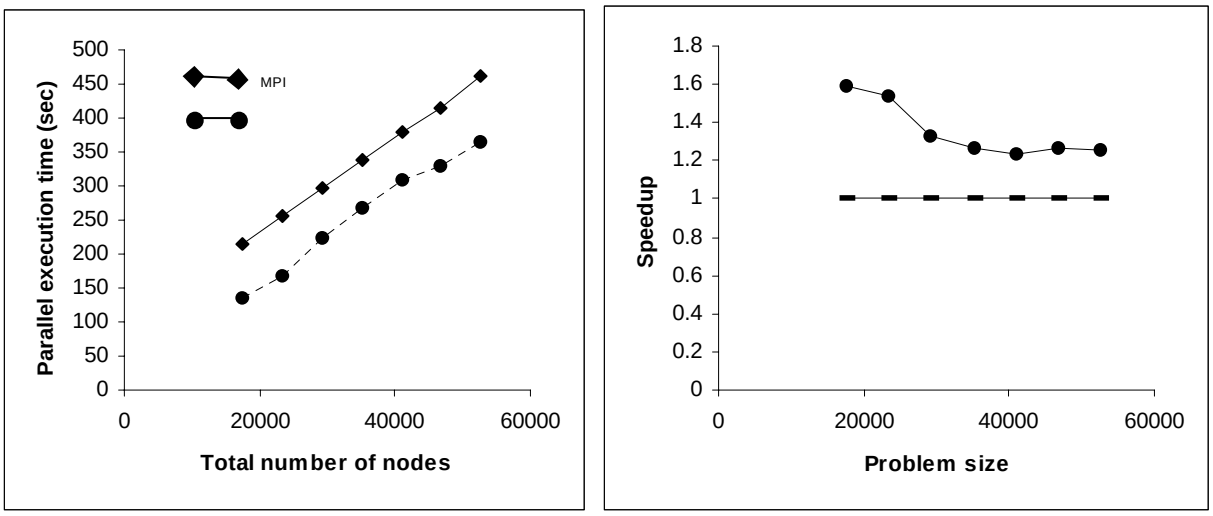

Figure 9. (a) Comparison of execution times of EM3D algorithm between HMPI and MPI. (b) The speedup of EM3D algorithm obtained using HMPI over MPI.

from P_{IJ} to P_{KL}, and $\sqrt{n_g}$ is the number of generalised blocks along the column of $r \times r$ blocks. The total number of $r \times r$ blocks of matrix *A*, which processor P_{IJ} sends to processor P_{KL}, is $N_{IJKL} \times n_g$. Therefore,

$$\frac{H_{IJKL} \times \sqrt{n_g}}{N_{IJKL} \times n_g} \times 100 = \frac{H_{IJKL} \times \sqrt{n_g}}{(H_{IJKL} \times w_{IJ}) \times n_g} \times 100$$

$$= \frac{1}{w_{IJ} \times \sqrt{n_g}} \times 100$$

percent of data that should be in total 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 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, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. 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 says it. 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 generalised block of a matrix are specified by its first two parameters.

The most interesting fragments of the rest code of the HMPI parallel application are shown in Figure 8.

In the example shown above, HMPI runtime system is initialised 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 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 HMPI runtime system using operation **HMPI_Finalize**.

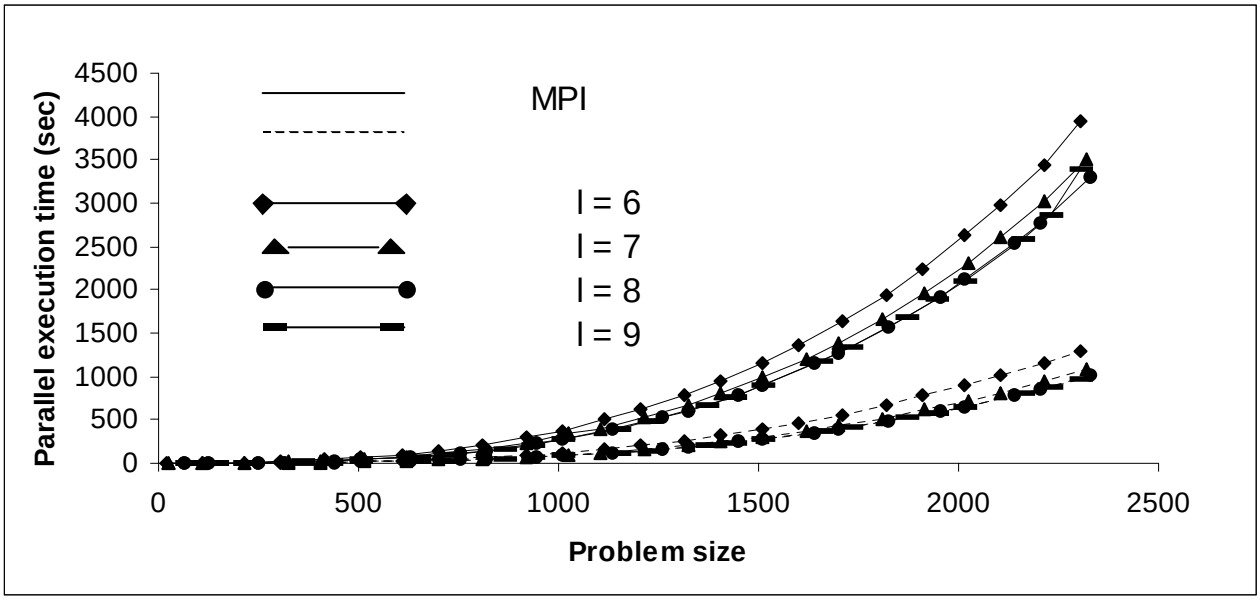

Figure 10. Comparison of execution times of MM algorithm between HMPI and MPI for different values of generalised block size.

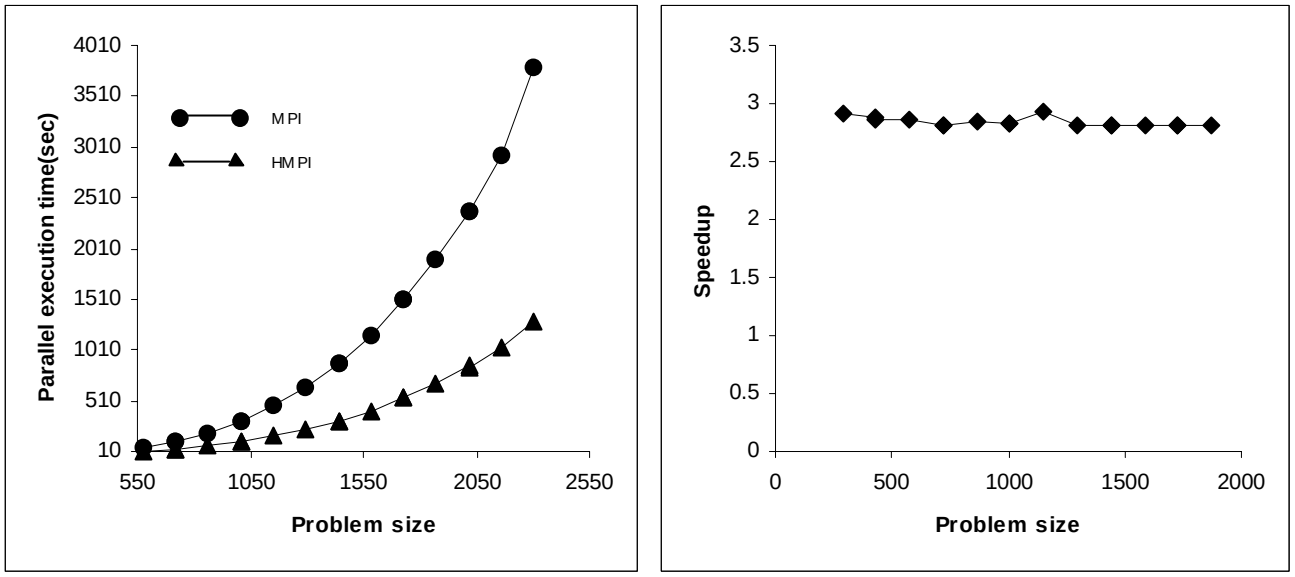

Figure 11. (a) Comparison of execution times of MM algorithm between HMPI and MPI. (b) The speedup of MM algorithm obtained using HMPI over MPI.

5 Experiments with HMPI

This section presents some results of experiments with the HMPI applications presented in Sections 3 and 4.

A small heterogeneous local network of 9 different Solaris and Linux workstations is used in the experiments for the EM3D algorithm. The speeds of the workstations demonstrated on the core computation of this algorithm, are 46, 46, 46, 46, 46, 46, 176, 106, and 9. Note that the figures give the average speeds measured at runtime during the experiments. The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers.

Figure 9(a) shows the comparison of the execution times of the HMPI application and the standard MPI

application executing EM3D algorithm. The experimental results are obtained by averaging the execution times over a number of experiments. One can see that the HMPI application is almost 1.5 times faster than the standard MPI one. Figure 9(b) demonstrates the speedup of the HMPI program over the MPI one.

All results are obtained for **r** = **l = 9**, which have appeared optimal. Figure 10 shows results for different values of generalised block sizes for the value of **r = 8**.

A small heterogeneous local network of 9 different Solaris and Linux workstations is used in the experiments for the MM algorithm. The speeds of the workstations demonstrated on the core computation of this algorithm, are 46, 46, 46, 46, 46, 46, 46, 106, and 9. Note that the figures give the average speeds measured at runtime during the experiments. The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers.

Figure 11(a) shows the comparison of the execution times of the HMPI application and the standard MPI application using homogeneous 2D block-cyclic data distribution. The experimental results are obtained by averaging the execution times over a number of experiments. One can see that the HMPI application is almost 3 times faster than the standard MPI one. Figure 11(b) demonstrates the speedup of the HMPI program over the MPI one.

6 Conclusion

We consider the HMPI 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 HMPI.

References

[1] Jack Dongarra, Steven Huss-Lederman, Steve Otto, Marc Snir, and David Walker. *MPI: The Complete Reference*. The MIT Press, 1996.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard", *Parallel Computing* 22(6), pp.789-828, 1996.

[3] I. Foster, J.Geisler, C. Kesselman, and S. Tuecke, "Managing Multiple Communication Methods in High-Performance Networked Computing Systems", *Journal on Parallel and Distributed Computing* 40(1), pp.35-48, 1997.

[4] O.Aumage, L.Bouge, and R.Namyst, "A Portable and Adaptative Multi-Protocol Communication Library for Multithreaded Runtime Systems", *Proc. 4th Workshop on runtime systems for Parallel Programming (RTSPP '00)*, Lect. Notes in Comp. Science 1800, pp.1136-1143, Cancun, Mexico, 2000.

[5] G. E. Fagg, A.Bukovsky, and J. Dongarra, "HARNESS and Fault Tolerant MPI", *Parallel Computing* 27(11), pp.1479-1495, 2001.

[6] A. Kalinov and A. Lastovetsky, "Heterogeneous Distribution of Computations
Solving Linear Algebra Problems on Networks of Heterogeneous Computers",
Journal of Parallel and Distributed Computing 64(4), pp.520-535, 2001.

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

[8] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, "ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers---Design Issues and Performance," *Computer Physics Comm.*, vol. 97. pp. 1-15, 1996.

[9] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.3. 1998.

[10] N.Mirenkov, A.Vazhenin, R.Yoshioka, T.Ebihara, T.Hirotomi, and T.Mirenkova, "Self-Explanatory Components: A New Programming Paradigm", *International Journal of Software Engineering and Knowledge Engineering* 11(1), 5-36, 2001.

[11] K. Yelick, C. Wen, S. Chakrabarti, E. Deprit, J. A. Jones, A. Krishnamurthy, "Portable Parallel Irregular Applications," Workshop on Parallel Symbolic Languages and Systems, Lecture Notes in Computer Science, 1995.

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

BIOGRAPHIES

Alexey Lastovetsky received the PhD degree from the Moscow Aviation Institute in 1986, and the Doctor of Science degree from the Russian Academy of Sciences in 1997. He is currently a lecturer in the Computer Science Department at University College Dublin, National University of Ireland. His main research interests are parallel and distributed programming languages and systems for heterogeneous environments.

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