Experimental Study of Six Different Parallel Matrix-Matrix Multiplication Applications for Heterogeneous Computational Clusters of Multicore Processors

Pedro Alonso[1], Ravi Reddy[2], Alexey Lastovetsky[2]

School of Computer Science and Informatics, University College Dublin
Technical Report UCD-CSI-2009-2
February, 2009

Abstract

In this document, we describe two strategies of distribution of computations that can be used to implement parallel solvers for dense linear algebra problems for Heterogeneous Computational Clusters of Multicore Processors (HCoMs). These strategies are called Heterogeneous Process Distribution Strategy (HPS) and Heterogeneous Data Distribution Strategy (HDS). They are not novel and have already been researched thoroughly. However, the advent of multicores necessitates enhancements to them.

We conduct experiments using six applications utilizing the various distribution strategies to perform parallel matrix-matrix multiplication (PMM) on a local HCoM. The first application calls ScaLAPACK PBLAS routine PDGEMM, which uses the traditional homogeneous strategy of distribution of computations. The second application is an MPI application, which utilizes HDS to perform the PMM. The application requires an input, which is the two-dimensional processor grid arrangement to use during the execution of the PMM. The third application is also an MPI application but that uses HPS to perform the PMM. The application requires two inputs, which are the number of threads to run per process and the two-dimensional process grid arrangement to use during the execution of the PMM. The fourth application is the HeteroMPI application using the HDS strategy. It calls the HeteroMPI group management routines to determine the optimal two-dimensional processor grid arrangement and uses it during the execution of the PMM. The fifth application is the HeteroMPI application using the HPS strategy. It calls the HeteroMPI group management routines to determine the optimal two-dimensional process grid arrangement given the number of threads per process is preconfigured and uses it during the execution of the PMM. The final application is the Heterogeneous ScaLAPACK application, which applies the HPS strategy and reuses the ScaLAPACK PBLAS routine PDGEMM. The number of threads to run per process must be preconfigured.

We then compare the results of execution of these six applications. The results reveal that the two strategies can compete with each other. The MPI applications employing HDS perform the best since they fully exploit the increased thread-level parallelism (TLP) provided by the multicore processors. However, for large problem sizes, the non-cartesian nature of the data distribution may lead to excessive communications that can be very expensive. For such cases, the HPS strategy has been shown to equal and even out-perform the HDS strategy. We also conclude that HeteroMPI is a valuable tool to implement heterogeneous parallel algorithms on HCoMs because it provides desirable features that determine optimal values of the algorithmic parameters such as the total number of processors and the 2D processor grid arrangement.

---

[1] Department of Information Systems and Computation, Polytechnic University of Valencia
(palonso@dsic.upv.es)
[2] School of Computer Science and Informatics, University College Dublin
(manumachu.reddy, alexey.lastovetsky)@ucd.ie

Contents

1 Introduction ... 3
2 Homogeneous ScaLAPACK Application Using PDGEMM ... 8
3 MPI Application Using HDS ... 9
4 HeteroMPI Application Using HDS ... 12
4.1 Determination of the Optimal Algorithmic Parameters ... 17
4.1.1 Data Distribution Blocking Factor ... 18
4.1.2 Two-dimensional Process Grid Arrangement ... 19
5 MPI Application Using HPS ... 21
6 HeteroMPI Application Using HPS ... 21
7 Heterogeneous ScaLAPACK Application Using HPS ... 21
8 Experimental Results ... 25
9 Conclusions ... 44
10 Summary and Future Work ... 45
References ... 46

1 Introduction

Parallel platforms employing multicores are becoming dominant systems in High Performance Computing (HPC). Almost 90% of the supercomputing systems in the Top500 list [1] are based on dual-core or quad-core architectures. This rapid widespread utilization of multicore processors is due to several factors [2]. First, system builders have encountered insurmountable physical barriers to further increases in processor clock speeds. These are excessive production of heat, consumption of power, and leakage of voltage. Multicore designs are a natural response to this situation. By putting multiple processor cores on a single die, architects can continue to increase the number of gates on a chip without increasing the power densities. Second, the pins that connect the processor to main memory have become a bottleneck, with both the rate of pin growth and the bandwidth per pin slowing down. Physical limits on the number of pins and bandwidth on a single chip mean that gap between processor performance and memory performance will get progressively worse. And finally, due to the aforementioned fundamental physical limitations, commodity off-the-shelf (COTS) processors which were used to build tera- and petascale systems, due to their economic viability, will be unlikely to deliver the capabilities that cutting-edge research applications require. This would imply that these systems would now be built using heterogeneous constellations of special purpose processing elements from different vendors. Examples include hardware accelerators, GPUs, FGPAs, and communication processors (NIC-processing, RDMA).

Therefore, to summarize, computers containing multicore processors will become ubiquitous soon and will be widely deployed in clusters purposely built to tackle the most challenging scientific and engineering problems. A cluster built from such computers, called the Heterogeneous Computational Cluster of Multicore Processors (HCoM), will be inherently heterogeneous due to the following reasons:

- Different computers in a HCoM may contain differing number of multicore processors from different vendors with different processing capabilities. Already, a computer built using heterogeneous multicore processors, for example, CELL BE, GPUs etc, has become a reality;
- Communications between multicore processors inside a computer will be faster compared to the communications between computers because they use shared memory, and therefore a HCoM should be visualized and modeled as a tree/multilevel hierarchy of interconnected sets of multicore processors.

Therefore, the heterogeneity can arise from two different sources:

- **Heterogeneity of the multicore processors.** The heterogeneity could be due to the multicore processors having different processing speeds. This could be because either they are from the same vendor but have different number of cores or they are from different hardware vendors or some of them are old and some new, the old obviously exhibiting slower processing speed;
- **Heterogeneity of the communications.** A HCoM should be visualized and modeled as a tree/multilevel hierarchy of interconnected sets of multicore processors. The communications between different computers connected by a common nework on the same level (for example, ethernet) are more expensive compared to communications between multicore processors inside a computer (typically using shared memory). The communication segments connecting different computers on different levels could also have differing latencies and bandwidths.

Therefore, the advent of multicores pose many challenges to writing parallel solvers for dense linear algebra problems for a HCoM. Addressing these challenges would entail redesign and

rewriting of parallel algorithms to take into account the increased TLP, and the hierarchical nature of communications, satisfying the criteria of fine granularity, as cores are associated with relatively small local memories, and asynchronicity, to hide the latency of access to memory. Algorithms hitherto considered unsuitable/unscalable for being communication-intensive or due to high computation-to-communication ratio (granularity) will therefore have to be revisited. These criteria can be satisfied when an algorithm can generate a collection of independent tasks, each having a high ratio of floating point calculations to data required, that is, all the tasks involved are of Level 3 BLAS.

These solvers must take into account the aforementioned heterogeneities and provide "scalable" parallelism where speedups obtained are proportional to the number of cores as one scales from 4-16-128 and more cores. They must be written using hybrid programming models, for example MPI [3] plus OpenMP/Pthreads [4] where the communications between multicore processors are performed using optimized MPI communication routines and the computations locally are performed using threads exploiting the increased TLP provided by the multicores. To state the obvious, these solvers must also be automatically tuned for a HCoM, which means that they must automate the following complex optimization tasks, which are also described in [5]:

- Determination of the accurate values of platform parameters such as speeds of the processors, latencies and bandwidths of the communication links connecting different pairs of processors;
- Using an efficient communication model that would reflect the hierarchical nature of communications and would accurately predict the time of different types of communications, for example p2p, broadcast, gather, scatter etc. between different sets of processors on different levels;
- Determination of the optimal values of algorithmic parameters such as data distribution blocking factor and two-dimensional processor grid arrangement to be used during the execution of the parallel linear algebra routines, and finally
- Efficient mapping of the processes executing the parallel algorithm to the computers of the HCoM.

There are two strategies of distribution of computations that can be used to implement parallel solvers for dense linear algebra problems for a HCoM [6]. These strategies are not novel and have already been researched thoroughly. However, the advent of multicores necessitates enhancements to them.

Heterogeneous Process Distribution Strategy (HPS): In this strategy, more than one process is executed per computer. Each process gets the same amount of data, for example a sub-matrix obtained from partitioning the matrix between the processes. The number of processes executed on the computer multiplied by the number of threads run per process is equal to the number of cores in the computer. It is assumed that the native compiler produces code that treats a process as a thread, that is, it does not differentiate between a process and a thread. So two processes executed with two threads each would be equivalent to a single process with four threads and would utilize four cores. So depending on the problem size, there is an optimal number of processes to be executed on a computer and an optimal number of threads to run per process, that is, there is an optimal (process, thread) combination. The local computations are performed using optimized threaded BLAS library.

This definition differs from the original definition, which can be summarized as follows:

- The whole computation is partitioned into a large number of equal chunks;
- Each chunk is performed by a separate process;

- The number of processes run by each processor is proportional to its speed.

Thus, while distributed evenly across parallel processes, data and computations are distributed unevenly over processors of the heterogeneous computational cluster so that each processor performs the volume of computations proportional to its speed.

The difference is that the number of processes executed per computer in the modified strategy is equal to the number of cores. Therefore, the condition of proportionality of the number of processes run per processor proportional to its speed is relaxed.

Heterogeneous Data Distribution Strategy (HDS): In this strategy, one process is executed per computer (the computer may have one or more processors). The volume of data allocated to a computer is proportional to the speed of the computer. The number of threads run per computer is equal to the number of cores in the computer. This is to ensure that all the cores are fully utilized during the execution of the program. There is an unstated assumption here that the native compiler on the computer generates optimized code that fully utilizes all the cores by producing a one-to-one mapping between the cores and the threads. However, it is likely that there is an optimal number of threads that may exceed the number of cores. This must be pre-determined using optimized sequential routines.

It should also be noted the terms computer/process/processor are used interchangably for applications using this strategy, for we consider a computer with one or more processors (multicore or not) as a single entity. That is, one process is executed per computer even though the computer may have one or more processors (multicore or not). The local computations are performed using optimized threaded BLAS library.

There are subtle differences from the original definition of this strategy, where a processor is considered the single entity and not a computer, which may have one or more processors. Traditionally, in this strategy, one process is executed per processor and data is distributed over the processes using heterogeneous block-cyclic distribution such that the volume of data allocated to a processor is proportional to its speed. In the modified definition, a computer is considered as a whole due to two reasons.

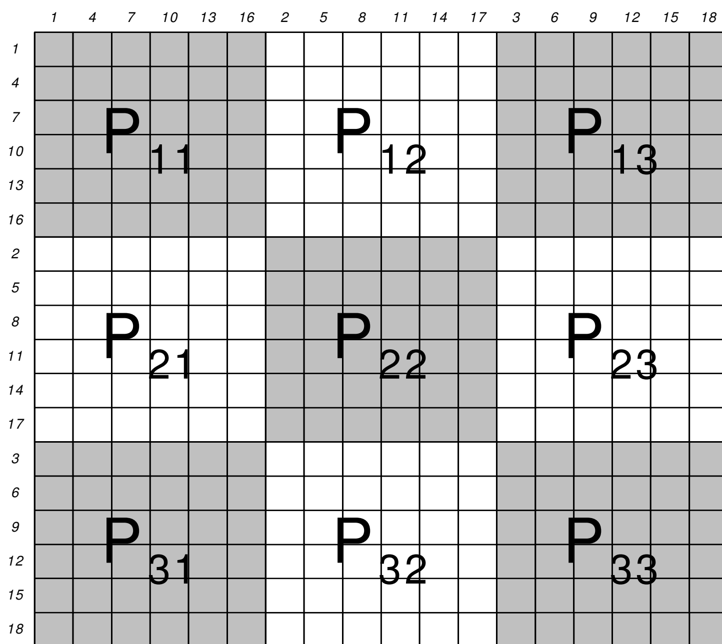
First, it is assumed that a computer solving a problem size given the number of threads set equal to the number of cores will complete the execution faster than some or all of its processors solving the same problem size. The difference is that in the first case, the solver is a sequential one employing optimized BLAS library with the number of threads set equal to the number of cores. In the second case, the solver is parallel because the problem is divided between the processors in a computer. In both the cases, the time of computations is the same. However, there are no communication overheads in the first case, whereas in the second case, there are communication overheads due to shared-memory communications between the processors. Notice that in both the cases the increased TLP of the cores can be exploited to the fullest extent by making sure that the number of processes multiplied by the number of threads executed per process is equal to the number of cores. This assumption will be invalid, of course, if the BLAS library performing the local computations is not optimized and not threaded.

Secondly, the solution space of 2D processor arrangements to evaluate will increase enormously if the (processor, thread) combinations need to be considered as is the case in the original definition of the strategy. Therefore, by treating a computer as a single entity, we eliminate this complexity.

The HPS strategy is a multiprocessing approach that is used to accelerate legacy parallel linear algebra programs on HCoMs. It allows the complete reuse of high-quality legacy parallel linear algebra software such as ScaLAPACK [7] on HCoMs with no redesign efforts and provides good

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}
2	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{11}	P_{11}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}
3	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{11}	P_{11}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}
4	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}
5	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}
6	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}
7	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}
8	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}
9	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}
10	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}
11	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}
12	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}
13	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}
14	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}
15	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}
16	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}	P_{11}	P_{12}	P_{13}
17	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}	P_{21}	P_{22}	P_{23}
18	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}	P_{31}	P_{32}	P_{33}

(a) Block cyclic distribution over 3x3 grid.



(b) Data distribution from process point-of-view.

Figure 1. A matrix with 18×18 blocks is distributed over a 3×3 process grid. The numbers on the left and on the top of the matrix represent indices of a row of blocks and a column of blocks, respectively. (a) The labelled squares represent blocks of elements, and the label indicates at which location in the process grid the block is stored – all blocks labelled with the same name are stored in the same process. Each shaded and unshaded area represents different generalised blocks. (b) Each processor has 6×6 blocks.

speedups. The Heterogeneous ScaLAPACK library [8], currently under development, uses this strategy and is built on top of HeteroMPI [9] and ScaLAPACK. It provides automatically tuned parallel linear algebra programs for HCoMs but most importantly performs all the aforementioned critical automations of the complex optimization tasks.

The rest of the document is organized as follows. In Section 2, we present the homogeneous application, which calls the ScaLAPACK PBLAS PDGEMM routine to perform the parallel matrix-matrix multiplication (PMM). We describe the ScaLAPACK outer-product algorithm used to perform the matrix product. In Section 3, we present the MPI application utilizing the HDS strategy. This application requires as input, the two-dimensional computer grid arrangement to use during the execution of the PMM. The parallel algorithm used for the execution of the matrix-matrix multiplication is a heterogeneous counterpart of the ScaLAPACK outer-product algorithm. In Section 4, we present the HeteroMPI application using the HDS strategy to perform the PMM. This application detects the optimal two-dimensional computer grid arrangement using the HeteroMPI group management functions. In Section 5, we present the MPI application utilizing the HPS strategy. This application requires as inputs, the number of threads to run per process and the two-dimensional process grid arrangement to use during the execution of the PMM. In Section 6, we present the HeteroMPI application using the HPS strategy to perform the PMM. This application requires as input, the number of threads to run per process. It detects the optimal two-dimensional process grid arrangement using the HeteroMPI group management functions. In Section 7, we describe the Heterogeneous ScaLAPACK application employing the multiprocessing HPS strategy and which calls the ScaLAPACK PBLAS PDGEMM routine to perform the PMM. The number of threads per process must also be preconfigured for this application. In Section 8, we present the experimental results comparing the performance of the six applications. We complete this document with conclusions and future work.

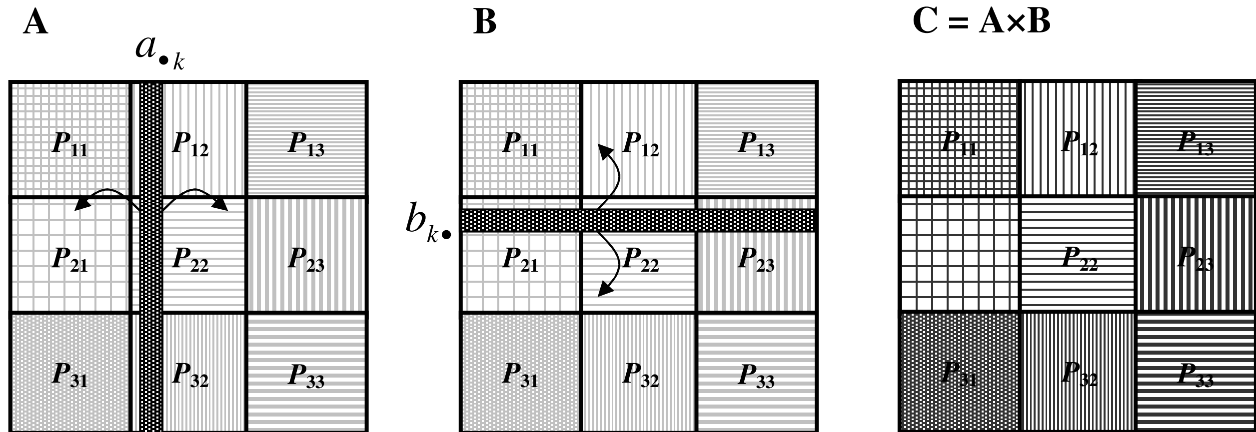


Figure 2. One step of the algorithm of parallel matrix-matrix multiplication (from the process point-of-view). First, the pivot column $a_{\bullet k}$ of *b*×*b* blocks of matrix A (emitting the curly arrows) is broadcast horizontally, and the pivot row $b_{k\bullet}$ of *b*×*b* blocks of matrix B (emitting the curly arrows) is broadcast vertically. Then, each *b*×*b* block c_{ij} of matrix C is updated, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$.

2 Homogeneous ScaLAPACK Application Using PDGEMM

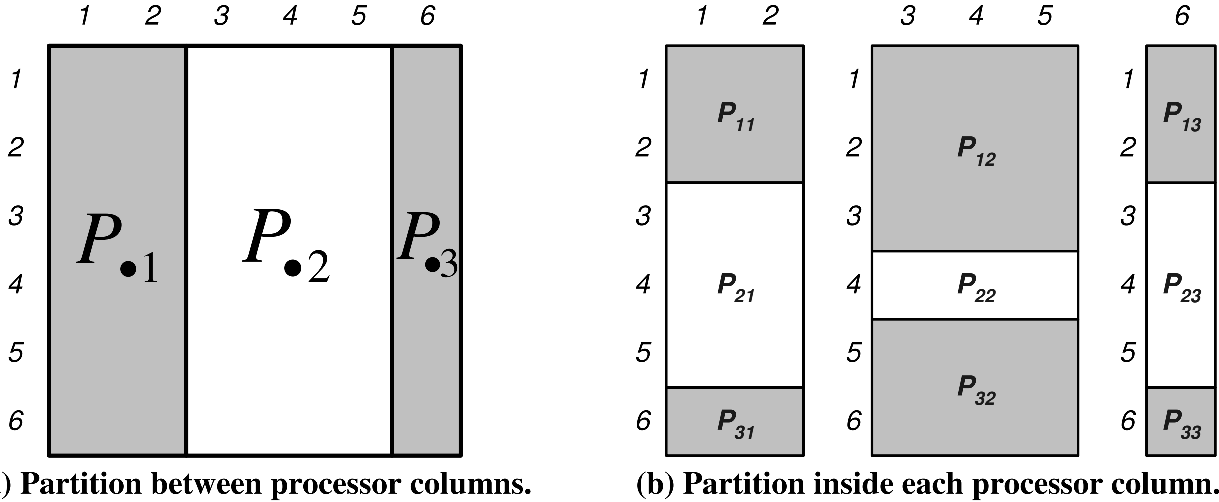
This section presents the ScaLAPACK outer-product algorithm used in the parallel matrix-matrix multiplication (PMM) of two dense $n \times n$ matrices *A* and *B* on a 2D process grid of size $p \times q$, $P_{ij}, \forall i \in [1, p] \wedge j \in [1, q]$. The routine performing the PMM is PDGEMM. The algorithm can be summarized as follows:

- The matrices *A*, *B* and *C* are distributed over the processes using the *two-dimensional block cyclic distribution* illustrated in Figure 1. Figure 1(a) shows the distribution from the matrix point-of-view. Figure 1(b) shows the distribution from the process point-of-view. The distribution scheme can be summarized as follows:
  - Each element in *A*, *B*, and *C* is a square $b \times b$ block. The unit of computation is the updating of one block, i.e., a matrix multiplication of size *b*;
  - The blocks are scattered in a cyclic fashion along both dimensions of the $p \times q$ process grid, so that for all $i, j \in \{1, \ldots, \frac{n}{b}\}$ blocks a_{ij}, b_{ij}, c_{ij} will be mapped to process P_{IJ} so that $I = (i-1) \bmod p + 1$ and $J = (j-1) \bmod q + 1$.
- The algorithm consists of $\frac{n}{b}$ steps. Figure 2 shows the operation of the algorithm from the process point-of-view. As shown in the figure, there is one-to-one mapping between the squares and the processes. Each process is responsible for computing its *C* square. At each step $k = \left(1, \ldots, \frac{n}{b}\right)$ of the algorithm,
  - The pivot column $a_{\bullet k}$ is owned by the column of processes $\{P_{iK_1}\}_{i=1}^{p}$ where $K_1 = (k-1) \bmod q + 1$ and the pivot row $b_{k\bullet}$ is owned by the row of processes $\{P_{K_2 j}\}_{j=1}^{q}$, where $K_2 = (k-1) \bmod p + 1$;
  - Each process P_{iK_1} (for all $i \in \{1, \ldots, p\}$) horizontally broadcasts its part of the pivot column $a_{\bullet k}$ to processes $P_{i\bullet}$ (see Figure 2);
  - Each process $P_{K_2 j}$ (for all $j \in \{1, \ldots, q\}$) vertically broadcasts its part of the pivot row $b_{k\bullet}$ to processes $P_{\bullet j}$ (see Figure 2);
  - Each process P_{ij} receives the corresponding part of the pivot column and pivot row and uses them to update each $b \times b$ block of its *C* square (implemented using a level-3 BLAS call) (see Figure 2).

Thus, after $\frac{n}{b}$ steps of the algorithm, each block c_{ij} of matrix *C* will be

$$c_{ij} = \sum_{k=1}^{\frac{n}{b}} a_{ik} \times b_{kj} \,,$$

i.e., $C = A \times B$.



(a) Partition between processor columns. **(b) Partition inside each processor column.**

Figure 3. Example of two-step distribution of a 6×6 square 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 processors 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 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** $0.05:0.08:0.03 \approx 2:3:1$**.**

3 MPI Application Using HDS

This section presents the PMM application multiplying matrix A and matrix B, where A, B, and C are dense matrices of size $m \times k$, $k \times n$, and $m \times n$ matrix elements respectively on a 2D heterogeneous processor grid of size $p \times q$, $P_{ij}, \forall i \in [1, p] \wedge j \in [1, q]$. We use dense square matrices and a 2D heterogeneous processor grid of size 3×3 for illustration purposes in Figure 4. Each matrix element is a square block of size $b \times b$. The heterogeneous parallel algorithm [10] used to compute this matrix product is a modification of the ScaLAPACK outer-product algorithm. It should be noted the terms computer/process/processor are interchangable, for we consider a computer with one or more processors (multicore or not) as a single entity. That is, one process is executed per computer even though the computer may have one or more processors (multicore or not). The algorithm can be summarized as follows:

- The matrices A, B, and C are divided into rectangles such that there is one-to-one mapping between the rectangles and the processors, and the area of each rectangle is proportional to the speed of the processor owning it. The procedure of data distribution invokes the data partitioning algorithm [6,10], which determines the optimal 2D column-based partitioning of

a rectangular area/matrix of size $m \times n$ on a 2D heterogeneous processor grid of size $p \times q$. The area is partitioned into uneven rectangles so that they are arranged into a 2D grid of size $p \times q$ and the area of a rectangle is proportional to the speed of the processor owning it. The inputs to the procedure are

- The rectangular area of size $m \times n$;
- The 2D processor grid of size $p \times q$, $P_{ij}, \forall i \in [1, p] \wedge j \in [1, q]$;
- The single number speeds of the processors, $s_{ij}, \forall i \in [1, p] \wedge j \in [1, q]$.

The output is the heights and the widths of the rectangles, $(r_{ij}, c_{ij}), \forall i \in [1, p] \wedge j \in [1, q]$. The procedure can be summarized as follows:

- First, the area $m \times n$ is partitioned into q vertical slices, so that the area of the j-th slice is proportional to $\sum_{i=1}^{p} s_{ij}$ (see Figure 3(a)). It is supposed that blocks of the j-th slice will be assigned to processors of the j-th column in the $p \times q$ processor grid. Thus, at this step, the load *between* processor columns in the $p \times q$ processor grid is balanced, 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 p horizontal slices, so that the area of the i-th horizontal slice in the j-th vertical slice is proportional to s_{ij} (see Figure 3(b)). 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, the load of processors *within* each processor column is balanced independently.

- The algorithm consists of $\frac{n}{b}$ steps. At each step $k = \left(1, \ldots, \frac{n}{b}\right)$ of the algorithm,
  - The pivot column $a_{\bullet k}$ is owned by the column of processors $\{P_{iK_1}\}_{i=1}^{p}$ and the pivot row $b_{k\bullet}$ is owned by the row of processors $\{P_{K_2 j}\}_{j=1}^{q}$, where expressions for K_1 and K_2 are involved;
  - Each processor P_{iK_1} (for all $i \in \{1, \ldots, p\}$) horizontally broadcasts its part of the pivot column $a_{\bullet k}$ to processors (horizontal neighbors) $P_{i\bullet}$ (see Figure 4). For example, the horizontal neighbors of P_{32} shown in Figure 4 are $\{P_{21}, P_{31}, P_{23}, P_{33}\}$;
  - Each processor $P_{K_2 j}$ (for all $j \in \{1, \ldots, q\}$) vertically broadcasts its part of the pivot row $b_{k\bullet}$ to processors $P_{\bullet j}$ (see Figure 4);
  - Each processor P_{ij} receives the corresponding part of the pivot column and pivot row and uses them to update each $b \times b$ block of its C square (implemented using a level-3 BLAS call) (see Figure 4).

Thus, after $\frac{n}{b}$ steps of the algorithm, each block c_{ij} of matrix C will be

$$c_{ij} = \sum_{k=1}^{\frac{n}{b}} a_{ik} \times b_{kj} ,$$

i.e., $C = A \times B$.

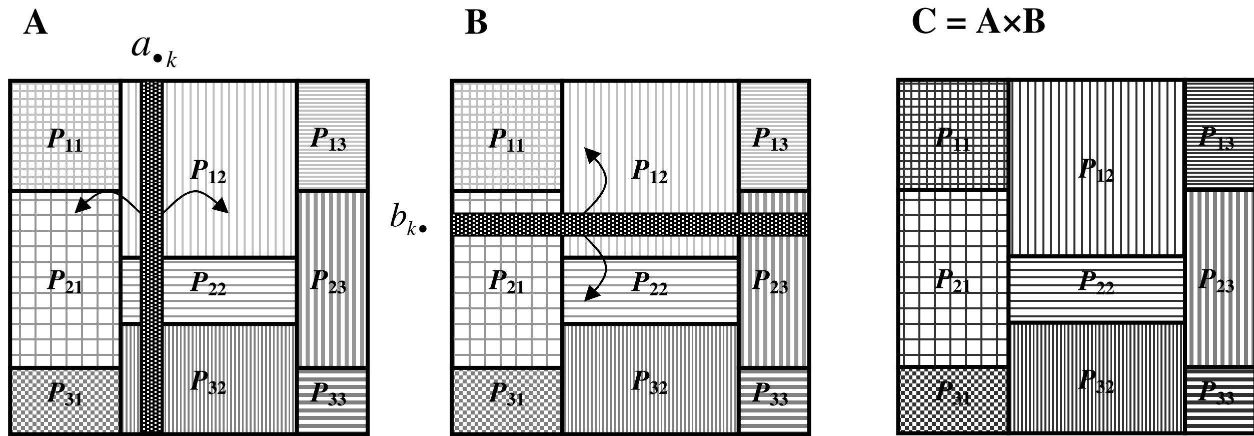


Figure 4. One step of the algorithm of parallel matrix-matrix multiplication employing a 2D heterogeneous processor grid of size 3×3. Matrices A, B, and C are partitioned such that the area of the rectangle is proportional to the speed of the processor owning it. First, each b×b block of the pivot column $a_{\bullet k}$ of matrix A (emitting the curly arrows) is broadcast horizontally, and each b×b block of the pivot row $b_{k\bullet}$ of matrix B (emitting the curly arrows) is broadcast vertically. Then, each *b*×*b* block c_{ij} of matrix C is updated, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$.

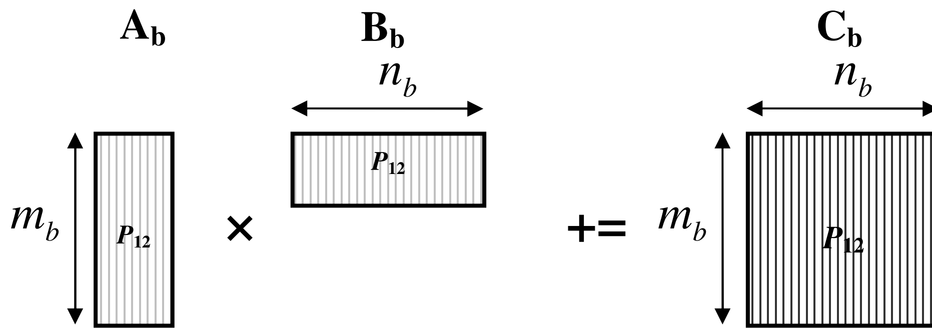


Figure 5. The computational kernel (shown here for processor P_{12} for example) performs a matrix update of a dense matrix C_b of size m_b×n_b using A_b of size m_b×1 and B_b of size 1×n_b. The matrix elements represent b×b matrix blocks.

For this application, the core computational kernel performs a matrix update of a matrix C_b of size $m_b \times n_b$ using A_b of size $m_b \times 1$ and B_b of size $1 \times n_b$ as shown in Figure 5. The size of the problem is represented by two parameters, m_b and n_b. The total number of matrix elements stored and processed on each processor is $(m_b \times n_b + m_b + n_b)$. We use a combined computation unit, which is made up of one addition and one multiplication, to express the volume of computation. Therefore, the total number of computation units (namely, multiplications of two $b \times b$ matrices) performed during the execution of the benchmark code will be approximately equal to $m_b \times n_b$. The absolute speed of the processor exposed by the application when solving the problem of size (m_b, n_b) can be calculated as $m_b \times n_b$ divided by the execution time of the matrix update.

The column-based matrix partitioning algorithm is based on the traditional performance model, which represents the speed of a processor by a constant positive number and computations are distributed amongst the processors such that their volume is proportional to this speed of the processor. However, this model is less accurate in the presence of paging. The functional performance model of heterogeneous processors [11] has proven to be more realistic than the traditional model because it integrates many important features of heterogeneous processors such as the processor heterogeneity, the heterogeneity of memory hierarchy, and the effects of paging. So we use a distributed iterative data partitioning algorithm, called DIPA-2D [12], which employs a 2D grid of heterogeneous processors and their FPMs to partition the matrices. It has three features distinguishing it from the traditional heterogeneous parallel and distributed algorithms employing a 2D grid of heterogeneous processors. Firstly, instead of accepting pre-built continuous speed functions of the processors, it uses as input a computational kernel whose execution time on a processor for a given number of computational chunks can be used to calculate the absolute speed of the processor. Secondly, it is a distributed algorithm by nature, and finally, it builds and uses a partially estimated FPM instead of the full FPM to provide optimal data distribution.

4 HeteroMPI Application Using HDS

This section presents the PMM application, which is composed of two parts. First, it calls the HeteroMPI routines to determine the optimal values of the algorithmic parameters, which are the total number of computers and the 2D computer grid arrangement to be used in the execution of the PMM. Second, the computers, which are members of the optimal 2D computer grid, perform the heterogeneous PMM explained in the preceding section. It should be noted the terms computer/process/processor are interchangable, for we consider a computer with one or more processors (multicore or not) as a single entity. That is, one process is executed per computer even though the computer may have one or more processors (multicore or not).

To summarize, HeteroMPI is an extension of MPI for programming high-performance computations on heterogeneous computational clusters (HCCs). The main idea of HeteroMPI is to automate the process of selection of a group of processes, which would execute the parallel algorithm faster than any other group.

The first step in this process of automation is the writing/specification of the performance model of the parallel algorithm, in this case, the performance model of the heterogeneous PMM algorithm. Performance model is a tool supplied to the programmer to specify her high-level knowledge of the application that can assist in finding the most efficient implementation on HCCs. This model allows description of all the main features of the underlying parallel algorithm that impact the execution performance of parallel applications on HCCs. 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.

```
/* 1 */  #define   HMPI_RECT_INDEX(a, b, c, d, p, q)  (a*p*q*q+b*p*q+c*q+d)
/* 2 */  nettype heteropmm(int p, int q, int n, int b, int bp, int bq, int w[p*q*p*q],
/* 3 */                  int h[p*q*p*q]) {
/* 4 */     coord I=p, J=q;
/* 5 */     node {I>=0 && J>=0: bench*((double)(n/b)*
/* 6 */          ((double)(w[HMPI_RECT_INDEX(I, J, I, J, p, q)]*b*bp)/(double)n)*
/* 7 */          ((double)(h[HMPI_RECT_INDEX(I, J, I, J, p, q)]*b*bq)/(double)n) );};
/* 8 */     link (K=p, L=q) {
/* 9 */         I>=0 && J>=0 && J!=L && (h[HMPI_RECT_INDEX(I, J, K, L, p, q)]>0) : length*
/* 10 */        ( w[HMPI_RECT_INDEX(I, J, I, J, p, q)]*h[HMPI_RECT_INDEX(I, J, K, L, p, q)]
/* 11 */               *(b*b)*sizeof(double)) [I, J]->[K, L];
/* 12 */        I>=0 && J>=0 && I!=K : length*( w[HMPI_RECT_INDEX(I, J, I, J, p, q)]
/* 13 */        *h[HMPI_RECT_INDEX(I, J, I, J, p, q)]
/* 14 */               *(b*b)*sizeof(double)) [I, J] -> [K, J];
/* 15 */     };
/* 16 */     parent[0,0];
/* 17 */     scheme {
/* 18 */         int i, j, k, CurrentI, CurrentJ, RootI, RootJ, ReceiverI, ReceiverJ, PivotI,
/* 19 */             PivotJ;
/* 20 */         for (k = 0; k < (n/b); k++) {
/* 21 */            /*
/* 22 */            * P(i,k) broadcasts a(i,k) to p(i,*) horizontally.
/* 23 */            * The processes holding the pivot column would be
/* 24 */            * (*, RootJ);
/* 25 */            */
/* 26 */            int RootJ = q-1;
/* 27 */            int cumj  = w[HMPI_RECT_INDEX(0, 0, 0, 0, p, q)];
/* 28 */            for (j = 1; j < q; j++) {
/* 29 */               if (k < cumj) {
/* 30 */                  RootJ = j-1;
/* 31 */                  break;
/* 32 */               }
/* 33 */               cumj += w[HMPI_RECT_INDEX(0, j, 0, j, p, q)];
/* 34 */            }
/* 35 */            par (RootI = 0; RootI < p; RootI++) {
/* 36 */               for (ReceiverI = 0; ReceiverI < p; ReceiverI++) {
/* 37 */                  for (ReceiverJ = 0; ReceiverJ < q; ReceiverJ++) {
/* 38 */                     if (RootJ != ReceiverJ) {
/* 39 */                        int ncommon = h[HMPI_RECT_INDEX(RootI, RootJ, ReceiverI,
/* 40 */                                                        ReceiverJ, p, q)];
/* 41 */                        int nsendf = w[HMPI_RECT_INDEX(RootI, RootJ, RootI, RootJ,
/* 42 */                                                       p, q)];
/* 43 */                        if ((ncommon > 0) && (nsendf > 0))
/* 44 */                           (100./nsendf) %% [(RootI), (RootJ)] ->
/* 45 */                                            [(ReceiverI), (ReceiverJ)];
/* 46 */                     }
/* 47 */                  }
/* 48 */               }
/* 49 */            }
/* 50 */            /*
/* 51 */            * P(k,j) broadcasts a(k,j) to p(*,j) vertically.
/* 52 */            */
/* 53 */            par (ReceiverJ = 0; ReceiverJ < q; ReceiverJ++) {
/* 54 */                /*
/* 55 */                * The processes holding the pivot row would be
/* 56 */                * (RootI, *);
/* 57 */                */
/* 58 */                int RootI = p-1; int RootJ = ReceiverJ;
/* 59 */                int cumi  = h[HMPI_RECT_INDEX(0, j, 0, j, p, q)];
/* 60 */                for (i = 1; i < p; i++) {
/* 61 */                   if (k < cumi) {
/* 62 */                      RootI = i-1;
/* 63 */                      break;
/* 64 */                   }
/* 65 */                   cumi += h[HMPI_RECT_INDEX(i, j, i, j, p, q)];
/* 66 */                }
```

Figure 6. The performance model of the heterogeneous PMM written in the performance model definition language (first part).

```
/* 67 */                for (ReceiverI = 0; ReceiverI < p; ReceiverI++) {
/* 68 */                   if (RootI != ReceiverI) {
/* 69 */                      int ncommon = w[HMPI_RECT_INDEX(RootI, RootJ, ReceiverI, RootJ,
/* 70 */                                                      p, q)];
/* 71 */                      int nsendf = h[HMPI_RECT_INDEX(RootI, RootJ, RootI, RootJ,
/* 72 */                                                     p, q)];
/* 73 */                      if ((ncommon > 0) && (nsendf > 0))
/* 74 */                         (100./nsendf) %% [(RootI), (RootJ)] ->
/* 75 */                                          [(ReceiverI), (RootJ)];
/* 76 */                   }
/* 77 */                }
/* 78 */            }
/* 79 */            /*
/* 80 */            *  Perform local update
/* 81 */            */
/* 82 */            par (CurrentI = 0; CurrentI < p; CurrentI++)
/* 83 */                par (CurrentJ = 0; CurrentJ < q; CurrentJ++)
/* 84 */                   (100./(n/b)) %% [CurrentI, CurrentJ];
/* 85 */         }
/* 86 */      };
/* 87 */   };
```

Figure 7. The performance model of the heterogeneous PMM written in the performance model definition language (second part).

HeteroMPI provides a dedicated performance model definition language (PMDL) for writing this performance model. The model and the PMDL are borrowed from the mpC programming language [13,14]. The PMDL compiler compiles the performance model written in PMDL to generate a set of functions, which make up the algorithm-specific part of the HeteroMPI runtime system. These functions are called by the mapping algorithms of HeteroMPI runtime system to estimate the execution time of the parallel algorithm.

A typical HeteroMPI application contains HeteroMPI group management function calls to create and destroy a HeteroMPI group, and the execution of the computations and communications involved in the execution of the parallel algorithm employed in the application by the members of the group. The principal group constructor routine **`HMPI_Group_auto_create`** determines the optimal number of processes and the optimal process grid arrangement, and thereby, relieving the application programmers from having to specify these parameters during the execution of the parallel application.

During the creation of a group of processes, the HeteroMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the HCC. It should be noted that this problem of mapping, in general, is NP-complete. The mapping algorithms used to solve the problem of selection of processes are explained in [9,14]. The solution 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 compilation of the performance model written in PMDL;
- 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 [14]. This model takes into account the material nature of communication links and their heterogeneity.

The performance model definition in PMDL of the heterogeneous parallel algorithm (presented in the preceding section) used to perform the PMM is shown in Figure 6 and Figure 7.

Lines 1-2 is a header of the performance model declaration. It introduces the name of the performance model **heteropmm**. It takes as input 8 parameters. Parameters **p** and **q** represent the number of processes along the row and the column of the 2D process grid arrangement respectively. Parameter **n** is the size of square matrices *A*, *B*, and *C*. Parameter **b** specifies the size of a square block of matrix elements, the updating of which is the unit of computation of the algorithm. Parameters **bp** and **bq** are used in the execution of the benchmark code, which performs a matrix update of a matrix C_b of size **(n/bp)×(n/bq)** using A_b of size **(n/bp)×(b)** and B_b of size **(b)×(n/bq)**. The value of the parameter **bp** is the square root of the total number of processes available for computation. The value of parameter **bq** is equal to total number of processes divided by **bp**.

Vector parameter **w** specifies the widths of the rectangles of assigned to different processes of the **p×q** grid. It logically represents a four-dimensional array of size **[p][q][p][q]**. The width of the rectangle assigned to process P_{IJ} is given by element **w[HMPI_RECT_INDEX(I, J, I, J, p, q)]** of the parameter, logically representing the element **w[I][J][I][J]**. All widths are measured in **b×b** blocks. Parameter **h** specifies the heights of rectangles of matrix *A*, which are horizontally communicated between different pairs of processes. Let R_{IJ} and R_{KL} be the rectangles of matrix *A* assigned to processes P_{IJ} and P_{KL} respectively. Then, **h[I][J][K][L]** gives the height of the rectangle of R_{IJ}, which is required by process P_{KL} to perform its computations. All heights are measured in **b×b** blocks. The possible combinations are illustrated in [9].

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

The **node** declaration associates the processes with this coordinate system to form a **p×q** process grid. It also describes the absolute volume of computation to be performed by each of the processes. The total volume of computations performed by each process in the benchmark code is **(n/bp)×(n/bq)×b**. The total volume of computation performed by each process P_{IJ} at each step of the parallel algorithm is **w[I][[J][I][J]*h[I][J][I][J]*(b)*(b)*(b)**. Since there are **n/b** steps in the parallel algorithm, the total volume would be **w[I][[J][I][J]*h[I][J][I][J]*(b)*(b)*(b)*(n/b)**. The expression in the node declaration describes the ratio of this total volume to the total volume of computations involved in the execution of the benchmark code.

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

```
int main(int argc, char** argv) {
   int opt_p, opt_q, *model_params, nd, *dp;
   HMPI_Group gid;
   // Initialize HeteroMPI runtime
   HMPI_Init(&argc, &argv);
   // Refresh the speeds of the processors
   int nc = HMPI_Get_number_of_computers();
   int bp = sqrt(nc); int bq = nc/bp;
   int output_p, input_p[4] = {n, b, bp, bq};
   HMPI_Recon(&dgemm_benchmark, input_p, 4, &output_p);
   // Create a HeteroMPI group of MPI processes
   if (HMPI_Is_host()) {
      // The performance model parameters are filled
      model_params[0] = p;
      model_params[1] = q;
      model_params[2] = n;
      model_params[3] = b;
      …
   }
   HMPI_Group_pauto_create(&gid, &HMPI_Model_heteropmm,
                           model_params);
   if (HMPI_Is_member(&gid)) {
      HMPI_Group_topology(&gid, &nd, &dp);
      opt_p = dp[0];
      opt_q = dp[1];
      MPI_Comm *pmmcomm = HMPI_Get_comm(&gid);

      // Heterogeneous PMM algorithm is performed here
      // calling standard MPI routines using MPI communicator 'pmmcomm'

      HMPI_Group_free(&gid);
   }
   HMPI_Finalize(0);
}
```

Figure 8. The main fragments of HeteroMPI program. It shows the usage of function `HMPI_Group_pauto_create`, which detects the optimal 2D processor grid arrangement to execute the heterogeneous PMM algorithm.

The second statement in the **`link`** declaration describes vertical communications related to matrix B. Obviously, only processes from the same column of the process grid send each other elements of matrix B. In particular, process P_{IJ} will send all its **`b×b`** blocks of matrix B to all other processes from column J of the process grid. The total number of **`b×b`** blocks of matrix B assigned to process P_{IJ} is given by **`w[I][J][I][J]*h[I][J][I][J]*(n/l)*(n/l)`**.
The **`scheme`** declaration describes **`n/b`** successive steps of the algorithm. At each step **`k`**,

- A column of **`b×b`** blocks of matrix A is communicated horizontally. If processes 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 **`b×b`** blocks transferred from P_{IJ} to P_{KL} will be **`H[I][J][K][L]`**, which is the height of the rectangle area that is communicated from P_{IJ} to P_{KL}. The total number of **`b×b`** blocks of matrix A, which process P_{IJ} sends to process P_{KL}, is **`w[I][J][I][J]*h[I][J][K][L]`**. Therefore,

`h[I][J][K][L]/(w[I][J][I][J]*h[I][J][K][L])*100=(100./ w[I][J][I][J])` percent of all data that should be sent from process P_{IJ} to process P_{KL} will be sent at the step. The 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 processes is carried out in parallel;

- A row of **b×b** blocks of matrix B is communicated vertically. For each pair of processes P_{IJ} and P_{KJ} involved in this communication, P_{IJ} sends a part of this row to P_{KJ}. The number of **b×b** blocks transferred from P_{IJ} to P_{KJ} will be `w[I][J][I][J]`. The total number of **b×b** blocks of matrix B, which process P_{IJ} sends to process P_{KJ}, is `w[I][J][I][J]*h[I][J][I][J]`. Therefore, `w[I][J][I][J]/(w[I][J][I][J]*h[I][J][I][J])*100=(100./ h[I][J][I][J])` percent of all data that should be sent from process P_{IJ} to process P_{KJ} will be sent at the step;
- Each process updates each of its **b×b** block of matrix C with one block from the pivot column and one block from the pivot row, so that each block c_{ij} $(i, j \in \{1,\ldots,\frac{n}{b}\})$ of matrix C will be updated to have the values $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. The process performs the same volume of computation at each step of the algorithm. Therefore, at each of `n/b` steps of the algorithm the process will perform `(100./(n/b))` 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 processes perform their computations in parallel.

The most interesting fragments of the rest code of the HeteroMPI parallel application are shown in Figure 8. The HeteroMPI runtime system is initialised using operation **HMPI_Init**. Then, the operation **HMPI_Recon** refreshes the speeds of processors. All the processors peform the benchmark code, `dgemm_benchmark`, which performs a matrix update of a matrix C_b of size `(n/bp)×(n/bq)` using A_b of size `(n/bp)×(b)` and B_b of size `(b)×(n/bq)`. This is followed by the creation of a HeteroMPI group of MPI processes using the operation **HMPI_Group_pauto_create**. The second parameter to this operation, **HMPI_Model_heteropmm**, is a handle generated from the compilation of the performance model, **heteropmm**, by the PMDL compiler. The members of this group then perform the computations and communications of the heterogeneous PMM 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**.

4.1 Determination of the Optimal Algorithmic Parameters

This section describes how the optimal values of the algorithmic parameters are determined. These are the data distribution blocking factor `(b)` and the 2D processor grid arrangement `(p,q)`.

4.1.1 Data Distribution Blocking Factor

The HeteroMPI function **`HMPI_Timeof`**, as shown below, is used to determine the optimal value of the data distribution blocking factor.

```
int b, opt_b;
double time, min_time = DBL_MAX; void *model_params;
for (b = 1; b <= n; b++) {
   // Fill the parameters to the performance model
   model_params[0] = p;  model_params[1] = q;
   …
   HMPI_Recon(&dgemm_benchmark, model_params);
   if (HMPI_Is_host())
      time = HMPI_Timeof(&HMPI_Model_heteropmm, model_params);
   if (time < min_time) {
      opt_b = b; min_time = time;
   }
}
```

The function `HMPI_Timeof` is used to estimate the execution time of the algorithm on the underlying hardware without its real execution. 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 of HeteroMPI. The parameters to this function are the handle, **`HMPI_Model_heteropmm`**, generated from the compilation of the performance model of the heterogeneous PMM algorithm, and the parameters to this performance model. As one can see from the code snippet, this estimation is performed for each possible set of values to the parameters to the performance model. Using the execution time predicted for each set, the optimal value can be determined, which would be the one resulting in minimum estimated execution time.

The estimation is based on the performance model of the heterogeneous PMM algorithm and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the heterogeneous PMM algorithm. The function **`HMPI_Recon`** is used to dynamically update the estimation of processor speeds at runtime. This is a collective operation and must be called by all the processes running in the application. The performance model of the executing network of computers is summarized as follows:

- The performance of each processor is characterized by the execution time of the same serial code (takes place during the execution of **`HMPI_Recon`**)
  - The serial code is provided by the application programmer;
  - It is supposed that the code is representative for the computations performed during the execution of the application;
  - The code is performed at runtime in the points of the application specified by the application programmer. Thus, the performance model of the processors provides current estimation of their speed demonstrated on the code representative for the particular application.

  In this case, the serial code, *hscal_dgemm_benchmark*, performs a local DGEMM update of **(`N/bp`)×`b`** and **`b`×(`N/bq`)** matrices where **`b`** is the data distribution blocking factor;
- The communication model [14] is seen as a hierarchy of homogeneous communication layers. Each is characterized by the latency and bandwidth. Unlike the performance model of processors, the communication model is static, a shortcoming that would be addressed in our

future work. Its parameters are obtained during the initialization of the HeteroMPI runtime and are not refreshed later.

The estimation procedure is explained in detail in [14] and is summarized here. The time of execution for each mapping, μ:*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 M^K possible mappings have to be probated to find the best one (here, 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 $M \times K$ possible mappings instead of M^K.

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 [12]. This is the core of the entire mapping algorithm determining its accuracy and efficiency. It takes into account material nature and heterogeneity of both processors and network equipment. It relies on fairly allocating processes to processors in a shared-memory multiprocessor normally implemented by operating systems for processes of the same priority (HeteroMPI processes are just the case). At the same time, it proceeds from the pessimistic point of view when estimating workload of different processors of that multiprocessor. Estimation of communication cost by the rule is sensitive to scalability of the underlying network technology. It treats differently communication layers serializing data packages and supporting their parallel transfer. The most typical and widely used collective communication operations are treated specifically to provide better accuracy of the estimation of their execution time. An important advantage of the rule is its relative simplicity and effectiveness. The effectiveness is critical because the algorithm is supposed to be multiply executed at runtime.

4.1.2 Two-dimensional Process Grid Arrangement

The HeteroMPI function **`HMPI_Group_pauto_create`** is used to determine the optimal values for the total number of processes, the number of process rows, the number of process columns, and efficient mapping of the processes to the executing computers of the network. Again the terms process/processor/computer are interchangable. Its operation is shown below:

```
int i, k, pa, p, opt_p, q, opt_q, *a, terminate = 0;
```

```
double t, mint=DBL_MAX; void *model_params = NULL;
// The host process
if (HMPI_Is_host()) {
   // The total number of processes available for computation
   int np = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
   for (k=np; k>=1; k--) {
      // The possible two dimensional process arrangements (p,q)
      HMPI_Get_2D_process_arrangements(k, &pa, &a);
      // For each two dimensional process arrangement (p,q)
      for (i=0; i<pa; i++) {
         // Fill the parameters to the performance model
         // Estimate the execution time for each process arrangement (p,q)
         t = HMPI_Timeof(&HMPI_Model_heteropmm, model_params);
         if (t < mint)
            // A better process arrangement found, continue the algorithm
            terminate = 0; mint = t; opt_p = p; opt_q = q;
      }
      if (terminate) { break; }
      terminate=1;
   }
   // Create a HeteroMPI group of processes with the optimal values of
   // algorithmic parameters
   model_params[0] = opt_p; // Optimal value of p
   model_params[1] = opt_q; // Optimal value of q
   model_params[2] = n;
   model_params[3] = b;     // Optimal value of b
   …
}
// Create a HeteroMPI group of processes
HMPI_Group_create(gid, model_params);
return HMPI_SUCCESS;
```

The algorithm can be summarized as follows: The number of steps of the algorithm is represented by the variable, **k**. For **k**=1, the total number of processes available for computation, **np**, is determined. All the possible two-dimensional process arrangements, **(p,q)**, whose product is **np**, are obtained using the function, **HMPI_Get_2D_process_arrangements**. For example, if the total number of processes available is 25, then the possible two dimensional process arrangements are {(1,25), (5,5), (25,1)}.

Each such process arrangement, **(p,q)**, is filled into the array of parameters to the performance model of the heterogeneous PMM algorithm. The function call **HMPI_Timeof** is then invoked to estimate the execution time of the algorithm. One of the inputs to this function call is the handle, **HMPI_Model_heteropmm**, which 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 of the heterogeneous PMM algorithm. 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 for that process arrangement. The selection process is described in detail in [9,14]. It is based on the performance model of the heterogeneous PMM algorithm and the performance model of the executing network of computers, which reflects the state of the network just before the execution of the heterogeneous PMM algorithm. During the selection process, the HeteroMPI runtime system searches for some approximate solution that can be found in some reasonable interval of time by probation of a subset of all possible mappings. From the execution times predicted for all

the possible process arrangements, the process arrangement, `(opt_p,opt_q)`, that results in the least estimated time of execution of the algorithm is determined.

For the next step, the total number of processes is decremented by one. The possible two-dimensional process grid arrangements are again obtained and evaluated using the function `HMPI_Timeof`. The algorithm continues if a process arrangement is found for which the estimated execution time is less than the estimated execution time of the process arrangement `(opt_p,opt_q)` determined in the previous step. Otherwise the algorithm terminates.

The optimal values of the blocking factor and the 2D process grid arrangement `(opt_p,opt_q)` are then passed as performance model parameters to the function call, `HMPI_Group_create`, which creates a HeteroMPI group of MPI processes that participate in the execution of the PMM application. This function call is a collective operation and must be called by all the processes available for computation in the predefined communication universe of HeteroMPI.

Heuristics are used to reduce the number of process arrangements evaluated. For example, one-dimensional arrangements are not evaluated in the case of matrix-matrix multiplication on ethernet where it can be proved using simple formulas [15] that they perform poorly compared to two-dimensional arrangements and do not minimize the objective function for networks with either sequential communications or parallel communications.

5 MPI Application Using HPS

This application requires two inputs, which are the number of threads per process and the two-dimensional process arrangement. It uses homogeneous distribution of computations, that is, each process gets the same amount of data. It reuses the code of the MPI application utilizing the HDS strategy.

6 HeteroMPI Application Using HPS

This application reuses the code of the HeteroMPI application utilizing the HDS strategy with some exceptions. It uses homogeneous distribution of computations, that is, each process gets the same amount of data. The number of threads per process must be preconfigured. This is due to a shortcoming in HeteroMPI, which is the feature that would detect the optimal (process, thread) combination in the HPS strategy.

7 Heterogeneous ScaLAPACK Application Using HPS

This section presents the Heterogeneous ScaLAPACK application, which utilizes the HPS strategy. Before we describe the application, we present an overview of Heterogeneous ScaLAPACK software.

The flowchart of the main routines of Heterogeneous ScaLAPACK package is shown in Figure 9. The high-level building blocks of Heterogeneous ScaLAPACK are HeteroMPI and ScaLAPACK. The principal routines in Heterogeneous ScaLAPACK package are the context creation functions for the ScaLAPACK routines (which include the PBLAS routines as well).

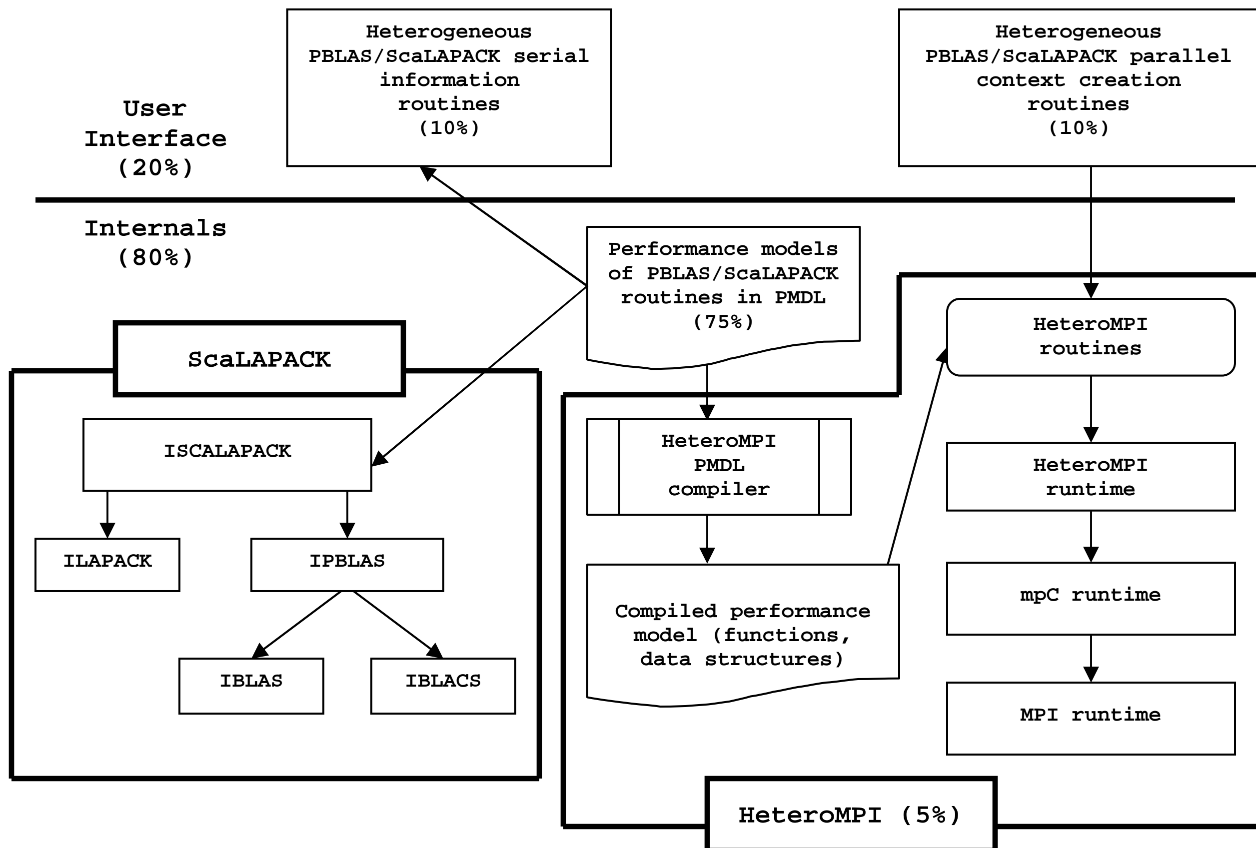


Figure 9. Flow of the Heterogeneous ScaLAPACK context creation routine call. The percentages give the breakup of Heterogeneous ScaLAPACK development efforts.

There is a context creation function for each and every ScaLAPACK routine. It provides a context for the execution of the ScaLAPACK routine but most importantly, performs the critical work of automating the difficult optimization tasks.

All the context creation routines have names of the form **`hscal_pxyyzzz_ctxt`**. The second letter, **`x`**, indicates the data type. For example, **`d`** would mean double precision real data. The next two letters, **`yy`**, indicate the type of matrix (or of the most significant matrix). For example, **`ge`** would represent a general matrix. The last three letters **`zzz`** indicate the computation performed. For example, the context creation function for the PDGEMM routine has an interface, which is shown below:

```
int hscal_pdgemm_ctxt(char* transa, char* transb, int * m, int *
n, int * k, double * alpha, int * ia, int * ja, int * desca, int
* ib, int * jb, int * descb, double * beta,int * ic, int * jc,
int * descc, int * ictxt)
```

This function call returns a handle to a HeteroMPI group of processes in **`ictxt`** and a return value of **`HSCAL_SUCCESS`** on successful execution. It differs from the ScaLAPACK PDGEMM call in the following ways:

- It returns a context but does not actually execute the PDGEMM routine;

- The input arguments are the same as for the PDGEMM call except
  - The matrices *A*, *B*, and *C* containing the data are not passed as arguments;
- The output arguments differ as follows:
  - An extra return argument, **`ictxt`**, which contains the handle to a group of MPI processes that is subsequently used in the actual execution of the PDGEMM routine;
  - A return value of **`HSCAL_SUCCESS`** indicating successful execution or otherwise an appropriate error code.

The function call is a collective operation and must be called by all the processes running in the Heterogeneous ScaLAPACK application. The context contains a handle to a HeteroMPI group of processes, which tries to execute the ScaLAPACK routine faster than any other group of processes. This context can be reused in multiple calls of the same routine or any routine that uses similar parallel algorithm as PDGEMM. The reader is referred to the Heterogeneous ScaLAPACK programmer's manual for more details of the user interface.

The Heterogeneous ScaLAPACK context creation/destruction routines call interface functions of HeteroMPI runtime system (the main routines being **`HMPI_Recon`**, **`HMPI_Timeof`**, **`HMPI_Group_auto_create`**). The Heterogeneous ScaLAPACK information functions calculate the total number of computations (arithmetical operations) performed by each process and the total number of communications in bytes between a pair of processes involved in the execution of the homogeneous ScaLAPACK routine. These routines are serial and can be called by any process. They do not actually execute the corresponding ScaLAPACK routine but just calculate the total number of computations and communications involved. The block ISCALAPACK ('I' standing for instrumented) represents the instrumented code of ScaLAPACK, which reuses the existing code base of ScaLAPACK completely. The instrumentations made to it are (a) Wrapping of the parallel regions of the code (ScaLAPACK and PBLAS routines) in mpC par loops recognized by the PMDL compiler and (b) Replacement of the serial BLAS computation routines and the BLACS communication routines by calls to information functions, which return the number of arithmetical operations performed by each process and number of communications in bytes between different pairs of processes respectively.

The first step in the implementation of the context creation routine for a Heterogeneous ScaLAPACK routine is the writing of its performance model using the PMDL. The performance model definitions contain the instrumented code components. The HeteroMPI compiler compiles this performance model to generate a set of functions. During the creation of the context, the mapping algorithms of HeteroMPI runtime system calls these functions to estimate the execution time of the ScaLAPACK routine.

The full performance model definition of PDGEMM can be studied from the file **`/PBLAS/SRC/pm_pdgemm.mpc`** in the Heterogeneous ScaLAPACK package. Figure 10 shows the essential steps involved in calling the ScaLAPACK PDGESV routine in a Heterogeneous ScaLAPACK program. The Heterogeneous ScaLAPACK runtime is initialized using the operation **`hscal_init`**. The heterogeneous PDGEMM context is obtained using the context constructor routine **`hscal_pdgemm_ctxt`**. The function call **`hscal_in_ctxt`** returns a value of 1 for the processes chosen to execute the PDGEMM routine, otherwise 0. Then the homogeneous ScaLAPACK PDGEMM routine is executed. The heterogeneous PDGEMM context is freed using the context destructor operation **`hscal_free_ctxt`**. When all the computations have been completed, the program is exited with a call to **`hscal_finalize`**, which finalizes the Heterogeneous ScaLAPACK runtime.

```
int main(int argc, char **argv) {
      int nprow, npcol, pdgemmctxt, myrow, mycol, c__0 = 0;
/* Problem parameters */
      char *TRANSA, *TRANSB;
      int  *M, *N, *K, *IA, *JA, *DESCA, *IB, *JB, *DESCB, *IC, *JC,
           *DESCC;
      double *ALPHA, *A, *B, *BETA, *C;
/* Initialize the Heterogeneous ScaLAPACK runtime */
      hscal_init(&argc, &argv);
/* Get the heterogeneous PDGEMM context */
      hscal_pdgemm_ctxt(TRANSA, TRANSB, M, N, K, ALPHA, IA, JA, DESCA,
                        IB, JB, DESCB, BETA, IC, JC, DESCC, &pdgemmctxt);
      if (!hscal_in_ctxt(&pdgemmctxt))
         hscal_finalize(c__0);
/* Retrieve the process grid information */
      Cblacs_gridinfo(pdgesvctxt, &nprow, &npcol, &myrow, &mycol);
/* Initialize the array descriptors for the matrices A, B, and C */
      descinit_(DESCA, …, &pdgemmctxt);  /* for Matrix A */
      descinit_(DESCB, …, &pdgemmctxt);  /* for Matrix B */
      descinit_(DESCC, …, &pdgemmctxt);  /* for Matrix C */
/* Distribute matrices on the process grid using user-defined pdmatgen */
      pdmatgen_(&pdgemmctxt, …); /* for Matrix A */
      pdmatgen_(&pdgemmctxt, …); /* for Matrix B */
      pdmatgen_(&pdgemmctxt, …); /* for Matrix C */
/* Call the PBLAS ‘pdgemm’ routine */
      pdgemm_(TRANSA, TRANSB, M, N, K, ALPHA, A, IA, JA, DESCA, B, IB,
              JB, DESCB, BETA, C, IC, JC, DESCC);
/* Release the heterogeneous PDGESV context */
      hscal_free_ctxt(&pdgemmctxt);
/* Finalize the Heterogeneous ScaLAPACK runtime */
      hscal_finalize(c__0);
   }
```

Figure 10. Heterogeneous ScaLAPACK program employing ScaLAPACK PDGEMM.

The PDGEMM context constructor routine **`hscal_pdgemm_ctxt`** is the main function automating the difficult optimization tasks of parallel programming on HCCs. They are the determination of the accurate values of the platform parameters such as the speeds of the processors and the latencies and bandwidths of the communication links connecting different pairs of processors, the optimal values of the algorithmic parameters such as the 2D process grid arrangement and efficient mapping of the processes executing the parallel algorithm to the executing nodes of the HCC. The execution of the context creation routine consists of the following steps:

1. Refreshing of the speeds of the processors using the HeteroMPI routine **`HMPI_Recon`**. A benchmark code representing the core computations involved in the execution of the PDGEMM routine is provided to this function call to accurately estimate the speeds of the processors. For example in this case, the benchmark code provided is a local GEMM update of **`(n/bp)×b`** and **`b×(n/bq)`** matrices where **`b`** is the optimal data distribution blocking factor. The optimal value of the blocking factor is determined using the HeteroMPI routine **`HMPI_Timeof`** (this is now performed during the installation of the Heterogeneous ScaLAPACK package);

2. Creation of a HeteroMPI group of MPI processes using the HeteroMPI's group constructor routine **HMPI_Group_pauto_create**. One of the inputs to this function call is the handle, which 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 of the ScaLAPACK routine. During this function call, the HeteroMPI runtime system detects the optimal process arrangement as well as solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network. The selection process is described in detail in [9, 14]. It is based the performance model of the ScaLAPACK routine and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the ScaLAPACK routine;
3. The handle to the HeteroMPI group is passed as input to the HeteroMPI routine **HMPI_Get_comm** to obtain the MPI communicator. This MPI communicator is translated to a BLACS handle using the BLACS routine **Csys2blacs_handle**;
4. The BLACS handle is then passed to the BLACS routine **Cblacs_gridinit**, which creates the BLACS context. This context is returned in the output parameter.

The Heterogeneous ScaLAPACK program uses the multiprocessing HPS strategy, which allows more than one process involved in its execution to be run on each processor. The number of processes to run on each processor during the program startup is determined automatically by the Heterogeneous ScaLAPACK command-line interface tools. During the creation of a HeteroMPI group in the context creation routine, the mapping of the parallel processes in the group is performed such that the number of processes running on each processor is as proportional to its speed as possible. 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 as proportional to its speed as possible. At the same time, the mapping algorithm invoked tries to arrange the processors along a 2D grid so as to optimally load balance the work of the processors.

8 Experimental Results

A small local heterogeneous cluster (Rosebud) consisting of multicore computers, SMPs, and single-processor workstations is used in the experiments. The specifications of this cluster are shown in Table 1. *rosebud01* and *rosebud02* are single-processor workstations. *rosebud03* and *rosebud04* are SMPs with two processors each. *rosebud05* and *rosebud06* are computers with four Itanium dual-core processors. *rosebud07* and *rosebud08* are computers with two Itanium dual-core processors. All the computers are running Linux OS. *rosebud01* to *rosebud04* have 32-bit OS whereas the rest have 64-bit OS. The communication network is based on 1 Gbit Ethernet. The software used is OpenMPI-1.2.8, ScaLAPACK-1.8.0, HeteroMPI-1.2.0, Heterogeneous ScaLAPACK-1.0.6-BETA, and Intel MKL toolkit, which provides an optimized BLAS library using OpenMP. The compiler used on all these machines is the Intel *icc* (version 9.1).

For the applications utilizing the HDS strategy, the number of threads configured per computer is equal to the number of cores on the computer. This is to ensure that all the cores are fully utilized during the execution of the application. For SMP machines, the number of threads configured per computer is equal to the number of processors. For the single-processor workstations, the number of threads is set to 1. The absolute speeds of the computers is obtained by performing a local DGEMM update of two matrices 2048×99 and 99×2048 where

Computer	Total main memory (kB)	No. of processors	No. of cores	HDS	
				No. of threads	Absolute speed (Mflops)
rosebud01	1035492	1	-	1	2295
rosebud02	1035688	1	-	1	2295
rosebud03	3635424	2	-	2	6515
rosebud04	3635424	2	-	2	6515
rosebud05	8240240	4	8	8	34600
rosebud06	8240512	4	8	8	34600
rosebud07	8240528	2	4	4	19130
rosebud08	8240672	2	4	4	19130

Table 1. Specifications of the eight computers in the Rosebud cluster.

99 is the optimal blocking factor. These speeds are shown in million flop/s. The heterogeneity of the network due to the heterogeneity of the computers is calculated as the ratio of the absolute speed of the fastest computer to the absolute speed of the slowest computer. As one can see, *rosebud05/rosebud06* is the fastest computer and *rosebud01/rosebud02* is the slowest computer. The heterogeneity in this case ≈ 15. If we exclude the computers *rosebud01* and *rosebud02*, the heterogeneity would be ≈ 5.

Table 2 shows the results of using different number of threads during the execution of sequential matrix-matrix multiplication application on different computers. The application calls optimized level-3 BLAS routine 'dgemm' to perform the matrix-matrix multiplication. There are two trends that can be observed in the execution performance. The first trend concerns problem sizes before the computer starts paging. For these problem sizes, the execution performance of the applications reduces when the number of threads exceeds the number of processors (single-processor or SMP) or the number of cores (multicore computers) on the computer. Further study must be done to conclude whether this is an intrinsic property of the multicore systems and is applicable for wide range of applications or whether the operating system/compiler can be tuned to enable execution of large number of threads that increases the execution performance of the applications. The second trend relates to the execution performance in the area of paging. It can be concluded that there is no definite rule to use for the optimal number of threads except that the number of threads to run per process must be greater than the number of processors or the number of cores.

Table 3 and Figure 11 shows the execution of two solvers on the computer rosebud06 performing the same matrix-matrix multiplication. The solver '*Threads*' is sequential. It calls the local optimized level-3 BLAS routine 'dgemm' to perform the matrix-matrix multiplication. During the execution of this solver, the number of threads is set equal to the number of cores (=8). The solver '*Processes*' adopts the original HDS strategy where one process is executed per processor and data is distributed over the processes using heterogeneous block-cyclic distribution such that the volume of data allocated to a processor is proportional to its speed. The number of threads is set to 1. The figures demonstrate that the sequential solver is more efficient than the the parallel solver. This justifies our modifications to the original HDS strategy, which is that a

Size of the matrix (n)	rosebud01/rosebud02 (Number of threads)						
	1	**2**	**4**	**8**	**10**	**12**	**16**
594	**0.2**	0.2	0.2	0.3	0.3	0.3	0.3
1188	**1.6**	1.8	1.8	1.9	1.9	2	2
2376	**13**	20	15	16	17	19	19
4752	**100**	101	104	107	109	109	111
7128	697	635	507	508	**467**	512	512
9504	4892	**3273**	3647	9600	10671	17752	6506

(a)

Size of the matrix (n)	rosebud03/rosebud04 (Number of threads)						
	1	**2**	**4**	**8**	**10**	**12**	**16**
594	0.16	**0.1**	0.16	0.20	0.20	0.20	0.24
1188	1.27	**0.71**	0.75	0.76	0.79	0.9	0.97
2376	7.8	**4.23**	4.56	5	5	4.7	5
4752	60	**32**	32	33	35	35	37
7128	202	**105**	107	108	108	111	118
9504	620	**249**	261	267	298	267	298

(b)

Size of the matrix (n)	rosebud05/rosebud06 (Number of threads)						
	1	**2**	**4**	**8**	**10**	**12**	**16**
594	0.08	0.05	0.03	**0.02**	0.12	0.12	0.13
1188	0.65	0.34	0.19	**0.12**	0.18	0.18	0.21
2376	5	2.59	1.36	**0.74**	0.96	0.92	0.81
4752	39	20	10	**5**	7	6	6
7128	132	67	35	**18**	22	18	18
9504	314	158	80	**42**	50	42	42
11880	613	308	155	**81**	97	81	82
14256	1057	530	268	**138**	166	139	140
16632	1681	843	427	**217**	264	221	226
19008	2705	1473	859	730	465	415	**395**
21384	5824	3641	2248	1755	1443	**1431**	1587

(c)

Size of the matrix (n)	rosebud07/rosebud08 (Number of threads)						
	1	**2**	**4**	**8**	**10**	**12**	**16**
594	0.09	0.05	**0.03**	0.12	0.12	0.12	0.12
1188	0.66	0.34	**0.19**	0.23	0.27	0.24	0.29
2376	5	2.6	**1.3**	1.43	1.7	1.45	1.55
4752	40	20	**10**	10	10	10	10
7128	133	67	**34**	35	35	35	35
9504	315	158	**80**	82	82	82	82
11880	614	308	**156**	160	160	160	161
14256	1058	533	**269**	274	275	274	274
16632	1684	845	**427**	428	438	437	440
19008	2724	1370	772	738	**680**	701	688
21384	5276	3035	1918	**1407**	1539	1474	1478

(d)

Table 2. The execution times for different number of threads. The application performs sequential matrix-matrix multiplication of three square matrices of size $n \times n$ using the optimized BLAS library provided by Intel MKL.

Size of the matrix (n)	Processes		Threads
	2×4	4×2	
594	0.38	0.27	0.034
1188	0.16	0.15	0.12
2376	0.99	0.98	0.75
4752	7.06	7.14	5.37
7128	23.22	22.72	17.84
9504	52.88	52.56	41.7
11880	104.17	99.64	81.43
14256	176.58	173.60	138.55
16632	284.25	271.93	216.37
19008	773.70	830.09	646.34

Table 3. Execution of matrix-matrix multiplication on rosebud06. The solver 'Threads' is sequential whereas 'Processes' is parallel. The number of threads set in the sequential solver and the number of processes used in the parallel solver is 8.

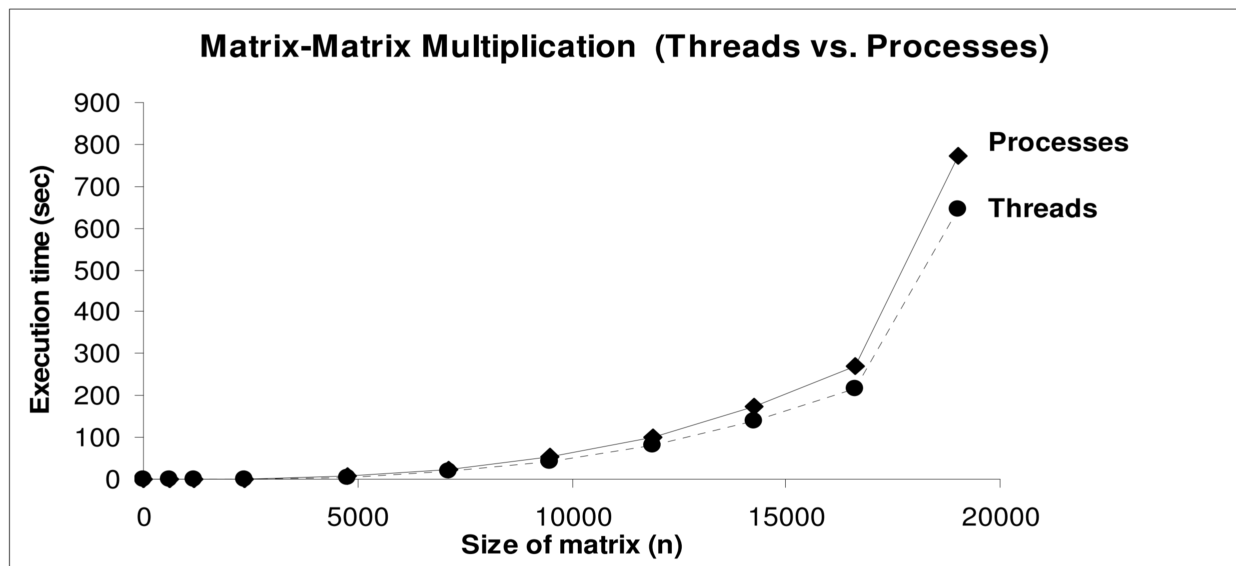
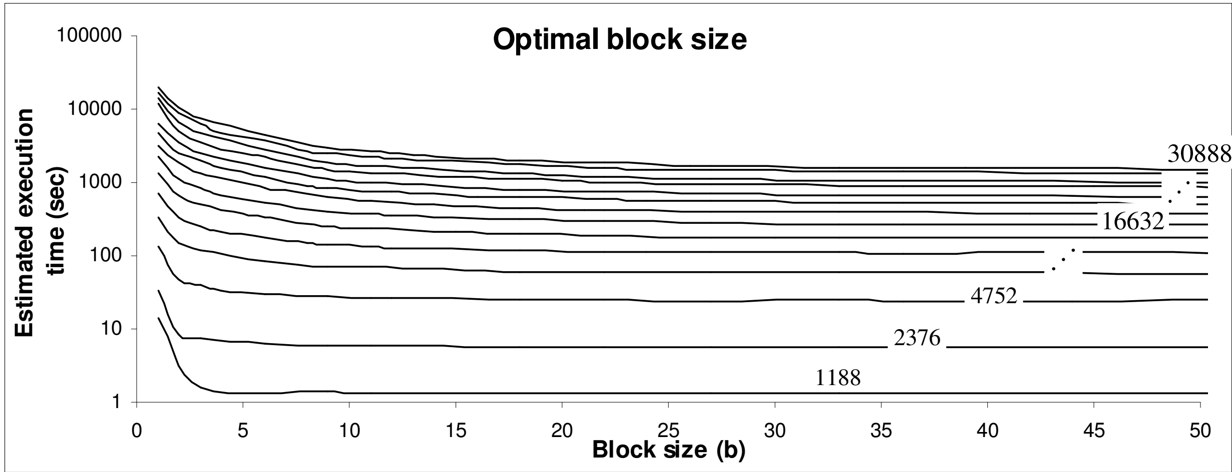


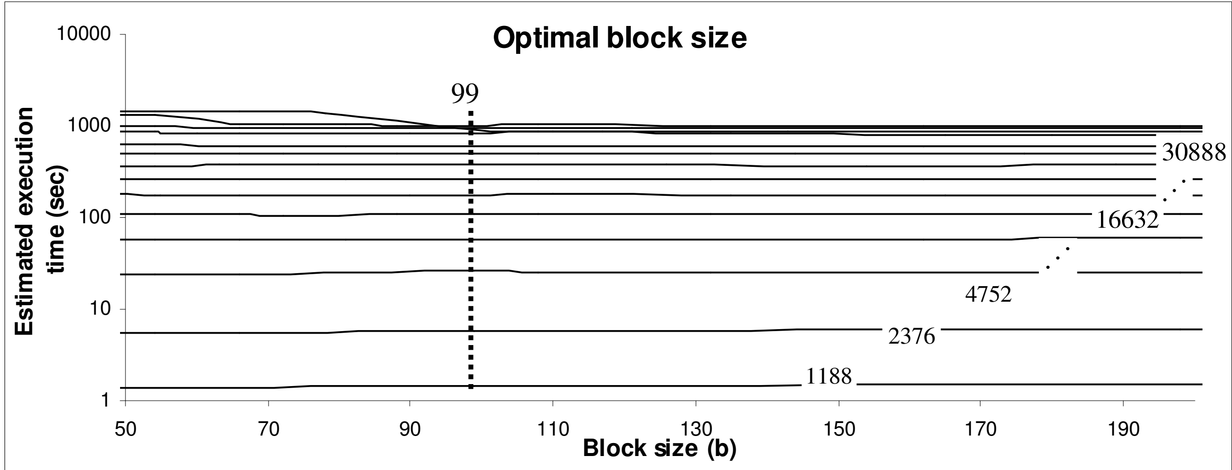
Figure 11. The solver using threads is efficient than the parallel solver on a computer performing the same matrix-matrix multiplication.

computer must be considered as a single entity instead of its processors for distribution of computations.

It would be worthwhile to present some notes on the procedure to find the optimal blocking factor. One approach investigated in [15] determines the optimal blocking factor for each computer, which can be a drawback. This is because a single value of the blocking factor must be used as input to the routines in the legacy linear algebra packages (this is a interface requirement). Modifying this approach to determine the single optimal value to use in the parallel application is not a trivial exercise. The procedure explained in section 4 determines the



(a)



(b)

Figure 12. Optimal block size estimated by the Heterogeneous ScaLAPACK library for various problem sizes (1188≤*n*≤30888). The execution times are on a log scale.

optimal value that minimizes the total execution time of the parallel algorithm. The MPI-HDS application is run using the 2D processor arrangement (p,q) of 4×2 for the range of problem sizes (1188≤n≤30888). The optimal values of the blocking factor are in the range (54≤b≤144) as shown in Figure 12. We use the value of 99. This procedure is executed separately and before the execution of all the parallel applications. So ideally this procedure must be executed during the installation of the software (as is done in the case of Heterogeneous ScaLAPACK).

For all the experimental results shown in the tables, the 2D process/processor/computer arrangements are shown below the tables. The processes/processors/computers are arranged in the grid in decreasing order of their speeds along each process row and along each process column. For example, a 2×4 computer grid arrangement containing all the computers shown in Table 1 will have the computers arranged as follows:

{rosebud05, rosebud06, rosebud07, rosebud08, rosebud04, rosebud03, rosebud02, rosebud01}

Similarly, a 3×8 process grid arrangement involving the computers {rosebud05, rosebud06, rosebud07, rosebud08} shown in Table 1, the number of processes run per computer being 8, 8, 4, 4, respectively, will have the processes arranged as follows:

Size of the matrix (n)	MPI-HDS								MPI-HDS (Best)
	1	2×1	3×1	4×1	2×2	3×2	4×2	8×1	
594	**0.03**	0.15	0.23	0.36	0.14	0.20	0.27	0.7	0.03 (1)
1188	**0.12**	0.55	0.95	1.4	0.55	0.80	1	3	0.12 (1)
2376	**1**	2	4	6	2	3	4	24	1 (1)
4752	**5**	11	17	24	11	13	18	78	5 (1)
7128	**18**	28	43	57	26	31	51	146	18 (1)
9504	**42**	57	81	114	50	62	86	249	42 (1)
11880	**81**	99	134	179	84	99	125	383	81 (1)
14256	139	155	203	264	**133**	167	184	551	133 (2×2)
16632	216	222	291	370	**191**	214	283	750	191 (2×2)
19008	646	313	398	501	**250**	320	353	987	250 (2×2)
21384	-	424	525	649	**350**	385	461	1260	350 (2×2)
23760	-	588	692	827	**453**	498	623	1575	453 (2×2)
26136	-	798	870	1029	**577**	670	773	1926	577 (2×2)
28512	-	2280	1065	1261	**756**	852	928	2320	756 (2×2)
30888	-	4305	1881	1520	1016	**1006**	1172	2750	1006 (3×2)
33264	-	-	-	-	1618	**1203**	1605	-	1203 (3×2)
35640	-	-	-	-	3264	**2091**	3053	-	2091 (3×2)
38016	-	-	-	-	4878	**3423**	5238	-	3423 (3×2)
40392	-	-	-	-	8879	**5575**	7638	-	5575 (3×2)

Table 4. MPI application utilizing the HDS strategy. "-" indicates very large execution times not useful for analysis.

2D computer grid arrangements:

No. of threads run per computer shown in brackets

rosebud01 (1), rosebud02 (1), rosebud03 (2), rosebud04 (2), rosebud05 (8), rosebud06 (8), rosebud07 (4), rosebud08 (4)

1: {rosebud05 | rosebud06}
2×1: {rosebud05, rosebud06}
3×1: {rosebud05, rosebud06, rosebud07}
4×1: {rosebud05, rosebud06, rosebud07, rosebud08}
2×2: {rosebud05, rosebud06, rosebud07, rosebud08}
3×2: {rosebud05, rosebud06, rosebud07, rosebud08, rosebud03, rosebud04}
4×2: {rosebud05, rosebud06, rosebud07, rosebud08, rosebud03, rosebud04, rosebud01, rosebud02}
8×1: {rosebud05, rosebud06, rosebud07, rosebud08, rosebud03, rosebud04, rosebud01, rosebud02}

{rosebud05, rosebud05, rosebud05, rosebud05, rosebud05, rosebud05, rosebud05, rosebud05, rosebud06, rosebud06, rosebud06, rosebud06, rosebud06, rosebud06, rosebud06, rosebud06, rosebud07, rosebud07, rosebud07, rosebud07, rosebud08, rosebud08, rosebud08, rosebud08}

The experimental results of the MPI-HDS application are shown in Table 4. The computers used in the different 2D computer grid arrangements are shown at the bottom of the table. The data distribution used is column-based [6,10]. The number of threads configured to run per

Size of the matrix (n)	MPI-HDS						MPI-HDS (Best)
	2×2		3×2		4×2		
	DIPA-2D Time (sec)	Execution time (sec)	DIPA-2D Time (sec)	Execution time (sec)	DIPA-2D Time (sec)	Execution time (sec)	
28512	11 (4)	745	12 (3)	852	32 (10)	896	756 (2×2)
30888	10 (3)	1006	14 (3)	1006	18 (3)	1155	1016 (2×2)
33264	11 (3)	1607	19 (4)	1203	220 (10)	1285	1222 (3×2)
35640	13 (3)	3251	19 (3)	2091	810 (5)	2143	2110 (3×2)
38016	15 (3)	4863	30 (5)	3423	1196 (3)	3442	3453 (3×2)
40392	22 (4)	8857	29 (4)	5575	1365 (5)	5673	5604 (3×2)

Table 5. Heterogeneous MPI application utilizing the HDS strategy for large problem sizes. IPADL2D is the distributed iterative 2D data partitioning algorithm used to distribute the matrices.

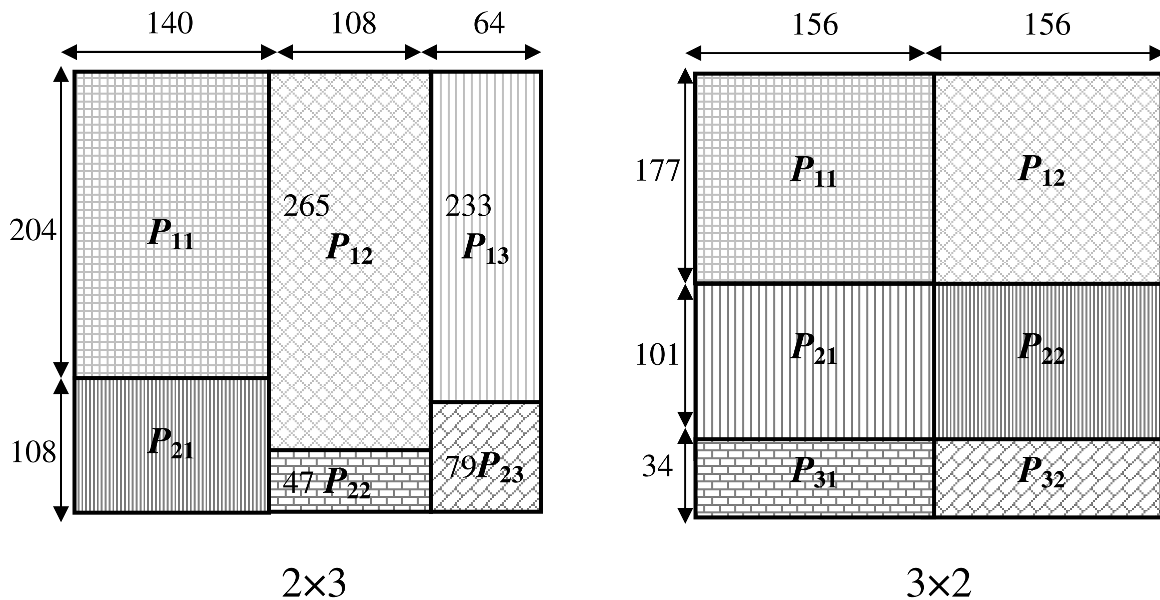
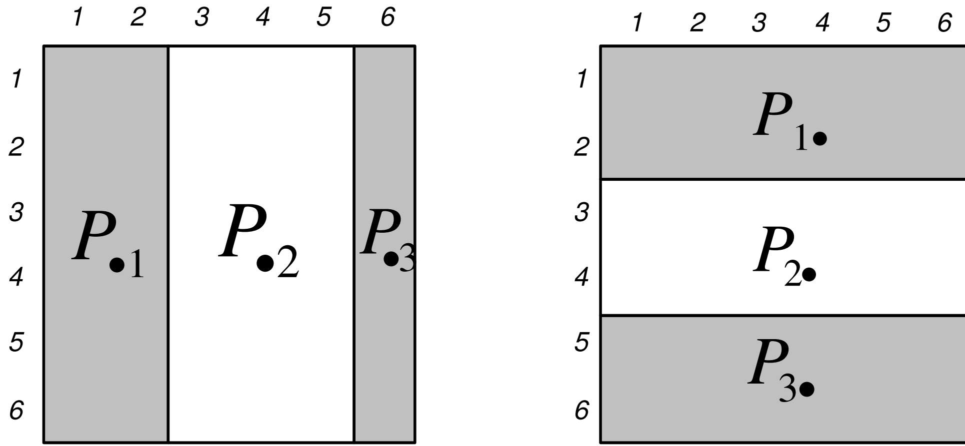
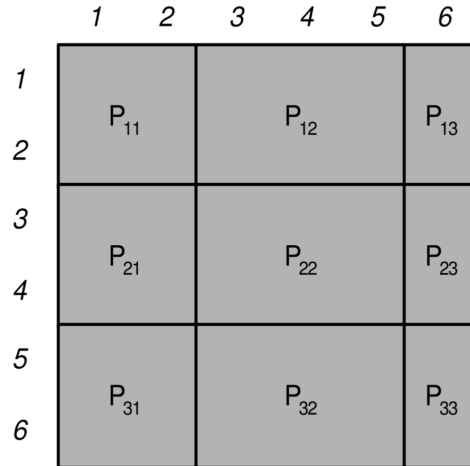


Figure 13. The column-based distribution for the arrangement 3×2 is actually cartesian whereas that of the arrangement 2×3 is non-cartesian. The number of communications (horizontal+vertical) at each step of the PMM for the arrangements {2×3, 3×2} is {24,18}. The matrix elements are square blocks of size 99×99.

computer is shown in the brackets. The final column shows the best execution times and the 2D computer grid arrangements, which varies with the problem size. As one can see, only the multicore computers and the SMPs are used in the best configurations for all problem sizes. The slow single-processor workstations *rosebud01* and *rosebud02* do not figure at all.



(a) Partition between processor columns. **(b) Partition between processor rows.**



(c) Final Partition.

Figure 14. Example of two-step Cartesian distribution of a 6×6 generalized block over a 3×3 processor grid. The relative speed of processors is given by matrix $s=\begin{pmatrix}0.11 & 0.17 & 0.05\\ 0.17 & 0.09 & 0.08\\ 0.05 & 0.25 & 0.03\end{pmatrix}$. (a) At the first step, the 6×6 square is distributed in a one-dimensional block fashion over processors columns of the 3×3 processor grid in proportion $0.33:0.51:0.16 \approx 2:3:1$. (b) At the second step, the 6×6 square is distributed in a one-dimensional block fashion over processors rows of the 3×3 processor grid in proportion $0.33:0.34:0.33 \approx 2:2:2$. (c) Final partition.

The experimental results in Table 5 show in detail the execution of the MPI-HDS application. It shows the execution times for large problem sizes where one or more computers start paging, which entails the use of sophisticated distributed iterative data partitioning algorithms [12] to determine the optimal data distribution. The computers *rosebud01* and *rosebud02* start paging for problem sizes (n>30000). The execution of the MPI-HDS application consists of two parts.

Size of the matrix (n)	MPI-HDS (COLUMN-BASED distribution) (p=3,q=2)	MPI-HDS (COLUMN-BASED distribution) (p=2,q=3)	MPI-HDS (CARTESIAN distribution) (p=3,q=2)	MPI-HDS (CARTESIAN distribution) (p=2,q=3)
594	0.20	0.30	0.20	0.20
1188	0.80	1	0.80	0.77
2376	3	4	3	3
4752	13	20	13	15
7128	31	49	31	37
9504	62	93	61	72
11880	99	145	100	121
14256	167	211	151	181
16632	214	294	214	254
19008	320	395	292	342
21384	385	513	386	450
23760	498	651	498	575
26136	670	814	649	727
28512	852	995	780	901
30888	1006	1211	951	1101
33264	1203	1434	1171	-
35640	2091	2551	1900	-
38016	3423	4448	3402	-
40392	5575	6478	5335	-

Table 6. The execution times of MPI-HDS applications using COLUMN-BASED and CARTESIAN distributions of the matrices. The 2D computer arrangement used in the execution is (*p*,*q*). "-" indicates failure of one or more processors.

Firstly, all the computers execute the DIPA-2D data partitioning algorithm to partition the matrices and secondly, they perform the PMM. The parameters to the DIPA-2D algorithm are the problem size (n), the 2D computer grid arrangement (p,q), and the termination criterion ε=0.05. Each computer arrangement has two columns. The first column shows the execution time of the DIPA-2D algorithm (first part of the parallel application). The total number of iterations of DIPA-2D is shown in the brackets. This also represents the total number of executions of the benchmark code/computational kernel. The second column shows the execution time of the PMM (second part of the parallel application). The final column shows the best execution times and the 2D computer grid arrangements.

The interesting computer arrangement to observe in this table is 4×2 where all the computers of the network are used including the slow ones *rosebud01* and *rosebud02*. It can be seen that the DIPA-2D algorithm spends large execution times to determine the optimal data distribution. This is because the slow computers *rosebud01* and *rosebud02* start paging as the problem size n exceeds 30000 and spend large times executing the initial benchmark. The optimal data distribution determined however does not include the slow computers and as a result the processor arrangement used in the PMM (second part of the parallel application) is 3×2. One can

Size of the matrix (n)	MPI-HDS (Best)		HeteroMPI-HDS			
	(p×q)	Execution time (sec)	Predicted (p×q)	Predicted execution time (sec)	Group creation time (sec)	Actual Time (sec)
594	1×1	0.03	1×1	0.04	0.1	0.15
1188	1×1	0.12	1×1	0.18	0.4	0.52
2376	1×1	1	1×1	1	1	2
4752	1×1	5	1×1	6	2	7
7128	1×1	18	1×1	19	3	21
9504	1×1	42	1×1	44	4	46
11880	1×1	81	1×1	85	5	86
14256	2×2	133	2×2	131	6	140
16632	2×2	191	2×2	191	8	201
19008	2×2	250	2×2	263	8	260
21384	2×2	350	2×2	347	9	360
23760	2×2	453	2×2	452	11	467
26136	2×2	577	2×2	576	11	593
28512	2×2	756	2×2	757	12	790
30888	3×2	1006	3×2	1010	13	1048
33264	3×2	1203	3×2	1225	15	1460
35640	3×2	2091	3×2	2167	15	2993
38016	3×2	3423	3×2	3552	17	4766
40392	3×2	5575	3×2	5773	18	7156

Table 7. HeteroMPI application utilizing the HDS strategy.

see that the execution times do not differ significantly from the execution times shown in the column for the processor arrangement 3×2.

It is also observed that in the case of this network, the 2D computer grid arrangements {3×2, 4×2} perform better than {2×3, 2×4} respectively. The column-based data distribution for the two 2D computer grid arrangements {2×3, 3×2} is shown in Figure 13 for the problem size n=30888. The slow computers {*rosebud01*, *rosebud02*} are not included in the arrangements{2×3, 3×2}. The reason why the column-based data distribution for 3×2 is efficient is because the matrix data distribution is actually cartesian (each processor has only two neighbors, one horizontal and one vertical). The column-based matrix data distribution for 2×3 is not cartesian, which results in more communications than in the case of the arrangement 3×2.

We performed further experiments to investigate the two matrix data distributions {column-based, cartesian}. The cartesian distribution of the matrices is illustrated in Figure 14. The results shown in Table 6 shows the execution times of MPI-HDS applications using column-based and cartesian distributions of the matrices. The best performing 2D computer arrangements are {3×2,

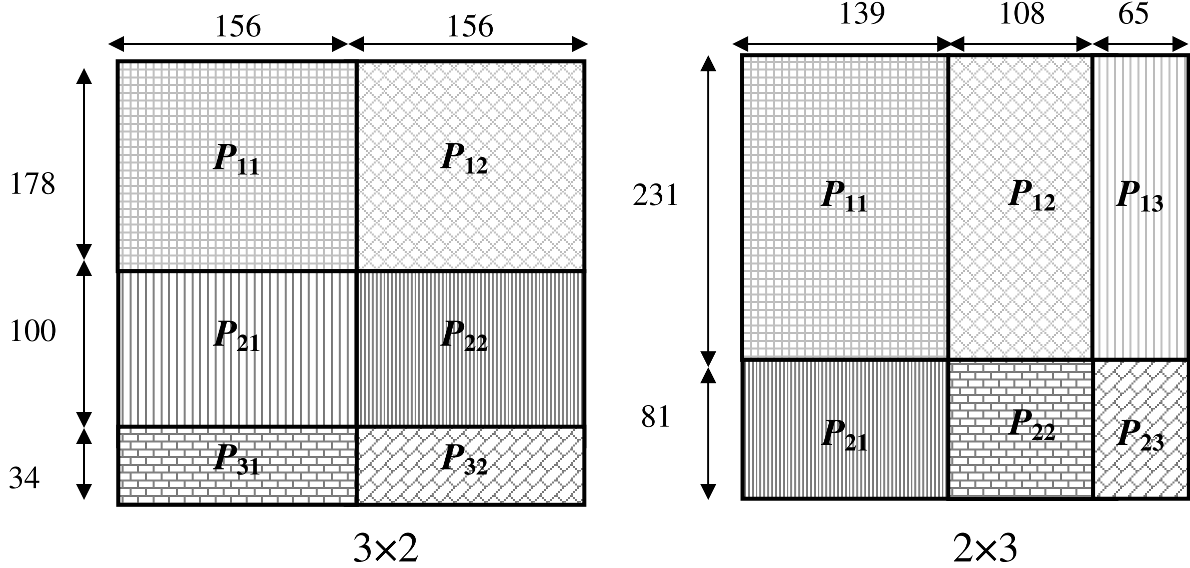


Figure 15. The cartesian data distributions for the arrangements 2×3 and 3×2. The matrix elements are square blocks of size 99×99. Process P_{11} is allocated large number of matrix elements in the case of 2×3.

3×2} with the respective matrix data distributions being {column-based, cartesian}. In fact, if we look closely at the matrix data distributions, the data distribution for the best performing 2D computer arrangement {3×2, column-based} is actually cartesian. The 2D computer grid arrangement {2×3, column-based} performs poorly due to non-cartesian distribution resulting in large number of communications. The 2D computer grid arrangement {2×3, cartesian} performs poorly due to load imbalance resulting from non-optimal matrix data distribution. The load imbalance is so large in this case that it results in failure of processors for problem sizes n>30888. Figure 15 shows the cartesian distributions for the 2D computer grid arrangements {2×3, 3×2} for the problem size n=30888. The respective allocations to the computer rosebud05 (P_{11}) are {231×139,178×156} square blocks of size 99×99. The speeds used for the calculation of the allocations are shown in Table 1. The large allocation in the case of the arrangement 2×3 results in the failure of the computer. So in some cases, it is very difficult to achieve the proportionality (volume of data to the speed of the processors) in the case of cartesian data distribution. Figure 14 illustrates the problem where true proportionality is not achieved.

Therefore, there are two observations that can be made from these experimental results. Firstly, there is an optimal 2D computer grid arrangement (also optimal total number of computers), which varies with the problem size. Secondly, the DIPA-2D algorithm must be used to determine the optimal data distribution as it employs functional performance model of heterogeneous processors, which has proven to be more realistic than the traditional models.

The experimental results highlight the importance of a tool that can provide features that can determine the optimal values of the algorithmic parameters such as the total number of computers and the 2D computer grid arrangement. HeteroMPI is one such tool. The experimental results of the HeteroMPI-HDS application are shown in Table 7. The second column shows the best execution times and the 2D computer grid arrangements from the MPI-HDS application.

These are compared with the predictions of HeteroMPI. The results of the HeteroMPI-HDS application in the third column are organized as follows. The first column shows the optimal 2D

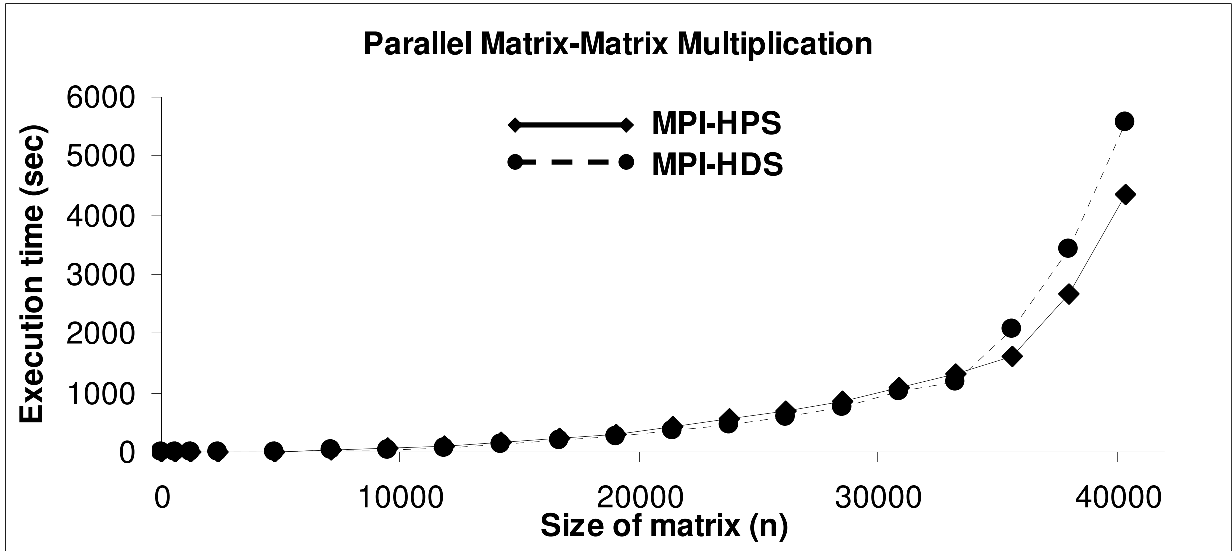
Size of the matrix (n)	MPI-HPS									MPI-HPS (Best)
	2×4 (1 t)	2×2 (2 t)	2×2 (4 t)	2×2* (4 t)	2×8 (1 t)	2×4 (2 t)	3×8 (1 t)	3×4 (2 t)	3×2 (4 t)	
594	0.27	**0.04**	0.5	0.14	1.64	0.14	1.64	0.25	0.25	0.04 (2×2, 2 t)
1188	0.15	**0.15**	0.5	0.5	0.66	0.5	0.66	0.92	0.92	0.15 (2×2, 2 t)
2376	1	**1**	2.4	2.3	2.5	2.3	5	4	4	1 (2×2, 2 t)
4752	7	**7**	11	11	12	11	19	16	16	7 (2×2, 2 t)
7128	23	**22**	28	27	30	27	48	39	41	22 (2×2, 2 t)
9504	53	**52**	56	53	60	53	99	81	80	52 (2×2, 2 t)
11880	104	100	97	**92**	105	95	160	127	130	92 (2×2, 4 t)
14256	177	172	154	**149**	169	149	226	192	199	149 (2×2, 4 t)
16632	284	276	232	**221**	250	223	308	259	282	221 (2×2, 4 t)
19008	773	746	325	**305**	350	311	408	364	385	305 (2×2, 4 t)
21384	-	-	443	**413**	460	431	530	478	505	413 (2×2, 4 t)
23760	-	-	582	**545**	615	572	682	620	647	545 (2×2, 4 t)
26136	-	-	809	**706**	949	824	853	784	822	706 (2×2, 4 t)
28512	-	-	1876	**870**	2286	2185	1043	965	1006	870 (2×2, 4 t)
30888	-	-	3401	**1090**	4232	3898	1266	1179	1226	1090 (2×2, 4 t)
33264	-	-	-	**1320**	-	-	2077	1857	1763	1320 (2×2, 4 t)
35640	-	-	-	**1597**	-	-	3288	3257	2596	1597 (2×2, 4 t)
38016	-	-	-	**2654**	-	-	5045	5702	4738	2654 (2×2, 4 t)
40392	-	-	-	**4355**	-	-	7427	7191	6430	4355 (2×2, 4 t)

Table 8. MPI application utilizing the HPS strategy. "-" indicates very large execution times not useful for analysis.

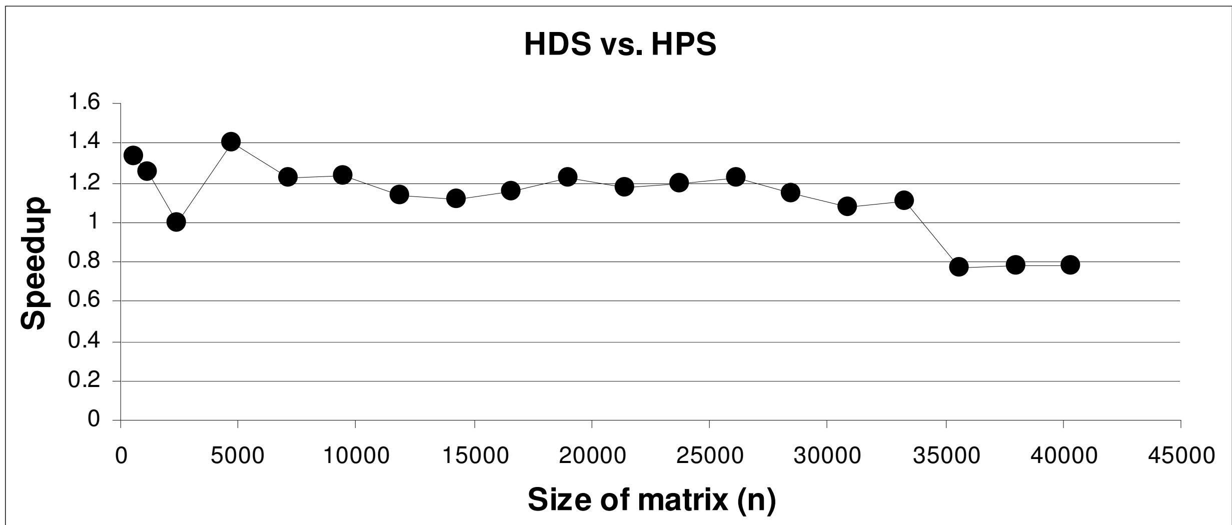
Process Arrangements:

No. of processes run per computer and the number of threads per process shown in brackets
2×4: rosebud05 (8,1) | rosebud06 (8,1)
2×2: rosebud05 (4,2) | rosebud06 (4,2)
2×2: rosebud05 (2,4) + rosebud06 (2,4)
2×2*: rosebud05 (1,4) + rosebud06 (1,4) + rosebud07 (1,4) + rosebud08 (1,4)
2×8: rosebud05 (8,1) + rosebud06 (8,1)
2×4: rosebud05 (4,2) + rosebud06 (4,2)
3×2: rosebud05 (2,4) + rosebud06 (2,4) + rosebud07 (1,4) + rosebud08 (1,4)
3×4: rosebud05 (4,2) + rosebud06 (4,2) + rosebud07 (2,2) + rosebud08 (2,2)
3×8: rosebud05 (8,1) + rosebud06 (8,1) + rosebud07 (4,1)+ rosebud08 (4,1)

computer grid arrangement predicted by HeteroMPI. The second column shows the predicted time of execution of the PMM. The third column shows the time taken by HeteroMPI group constructor routine to evaluate all the the possible 2D computer grid arrangements and to arrive at the best 2D computer grid arrangement. The final column shows the total execution time of the HeteroMPI-HDS application. This includes the time taken to execute the DIPA-2D algorithm and the time taken to determine the best 2D computer grid arrangement. As one can see, when it comes to 2D computer grid arrangements, the predictions of HeteroMPI are spot-on and the predictions of the execution times are accurate within 10%.



(a)



(b)

Figure 16. (a) The execution times of the MPI applications employing the HDS and the HPS strategies. (b) The speedup of the MPI application utilizing the HDS strategy over the MPI application utilizing the HPS strategy.

The experimental results in Table 8 show the execution times of the MPI application using the HPS strategy. The final column shows the best execution times and the optimal (process,thread) combinations, which vary with the problem size. There are interesting conclusions that can be drawn from the results. First, there is an optimal number of threads to run per process and an optimal number of processes to run per computer, that is, there is an optimal (process, thread) combination. As the problem size increases, it can be seen that the optimal number of threads per process increases from 2 to 4. For example, for the problem sizes {26136,28512,30888} and the process arrangements in the columns {2×2 (4t), 2×8 (1t), 2×4 (2t)}, the optimal number of threads per process has progressed from 1 to 4. This is because of fewer communications.

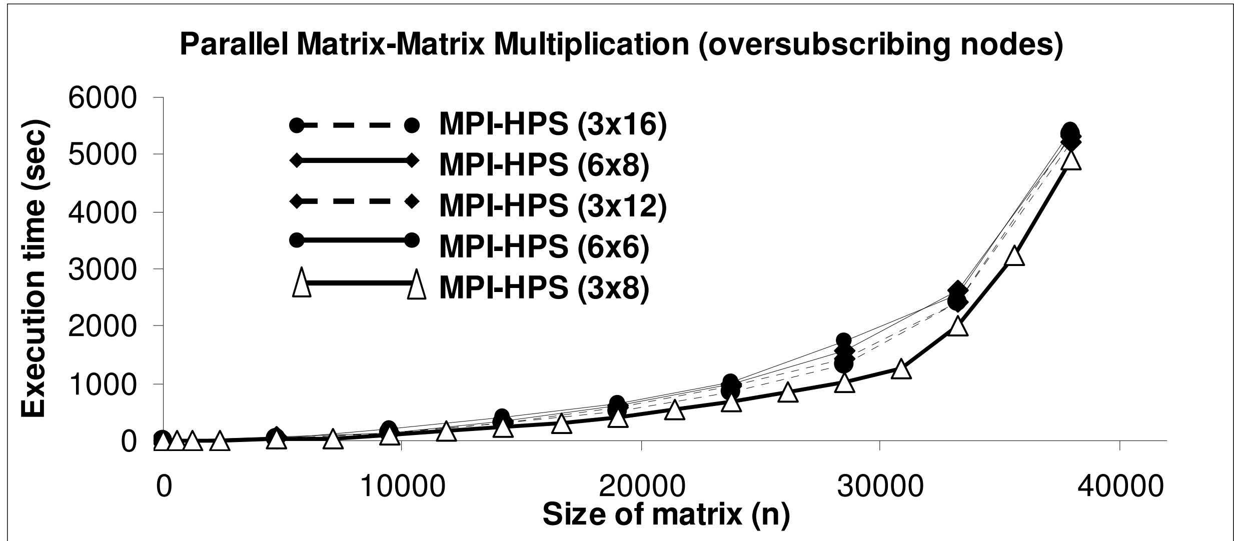


Figure 17. The execution times of the MPI application employing the HPS strategy. The multicore computers are oversubscribed, that is, the number of processes run on the computer is more than the number of cores.

Process Arrangements:

No. of processes run per computer shown in brackets:
3×16: rosebud05(16), rosebud06(16), rosebud07(8), rosebud08(8)
6×8: rosebud05(16), rosebud06(16), rosebud07(8), rosebud08(8)
3×12: rosebud05(12), rosebud06(12), rosebud07(6), rosebud08(6)
6×6: rosebud05(12), rosebud06(12), rosebud07(6), rosebud08(6)
3×8: rosebud05(8), rosebud06(8), rosebud07(4), rosebud08(4)

Similar results are observed for the problem sizes {33264,35640,38016,40392} for the process arrangements in the columns {3×8 (1t), 3×4 (2t), 3×2 (4t)}.

Figure 16 shows the speedup of the MPI-HDS application over the MPI-HPS application. The speedup calculated is the ratio of the execution time of the MPI-HPS application to the execution time of the MPI-HDS application. The results reveal that the two strategies can compete with each other. For the range of problem sizes ($n \le 35640$), the MPI applications employing HDS perform the best since they fully exploit the increased thread-level parallelism (TLP) provided by the multicore processors. However, for large problem sizes, the non-cartesian nature of the data distribution (where each processor has more than four neighbors) may lead to excessive communications that can be very expensive. For such cases, the HPS strategy has been shown to out-perform the HDS strategy.

Figure 17 shows the effect of oversubscribing the multicore computers, that is, the effect of running more processes than the number of cores in the computer. The number of threads per process on each computer is set to 1. One can notice the best performing 2D process grid arrangement (3×8) is the one where the number of processes run on each computer is equal to the number of cores.Figure 18 shows the effect of oversubscribing the multicore computers and the SMPs, that is, the effect of running more processes than the number of cores/processors in the computer respectively. The number of threads per process on each computer is set to 1. The figures show that once the number of processes exceed the number of cores/processors in the

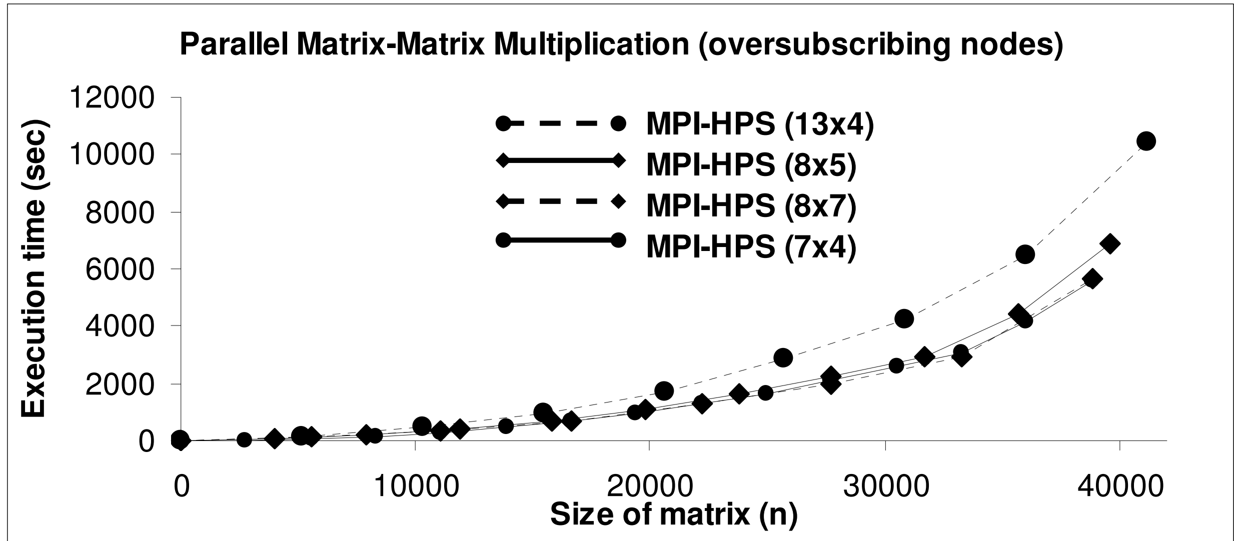


Figure 18. The execution times of the MPI application employing the HPS strategy. The multicore and the SMP machines are oversubscribed, that is, the number of processes run on the computer is more than the number of cores and processors respectively.

Process Arrangements:

No. of processes run per computer shown in brackets:
13×4: rosebud05(16), rosebud06(16), rosebud07(8), rosebud08(8), rosebud03(2), rosebud04(2)
8×5: rosebud05(12), rosebud06(12), rosebud07(6), rosebud08(6), rosebud03(2), rosebud04(2)
8×7: rosebud05(16), rosebud06(16), rosebud07(8), rosebud08(8), rosebud03(4), rosebud04(4)
7×4: rosebud05(8), rosebud06(8), rosebud07(4), rosebud08(4), rosebud04(2), rosebud03(2)

computer, the execution performance of the application goes down. However, in the region of paging for the problem sizes (n≤40000), the execution performance does converge and so the computers can be oversubscribed. But there still remains a problem of how many processes to run per computer and which 2D process grid arrangement to use. The problem becomes more complicated when threads are taken into account. This necessitates the importance of a tool, which can determine the optimal values of these algorithmic parameters automatically.

The experimental results of the HeteroMPI-HPS application are shown in Table 9. The second column shows the best execution times and the 2D process grid arrangements from the MPI-HPS application. These are compared with the predictions of HeteroMPI. At the moment, the number of threads must be pre-configured. Our future work would include enhancing HeteroMPI to determine the optimal (process, thread) combination in the HPS strategy.

The results of the HeteroMPI-HPS application in each of the main columns (3-5) are organized as follows. The first sub-column shows the optimal 2D process grid arrangement predicted by HeteroMPI. The second sub-column shows the estimated time of execution of the PMM. The third sub-column shows the time taken by HeteroMPI group constructor routine to evaluate all the the possible 2D process grid arrangements and to arrive at the best 2D process grid arrangement. The final sub-column shows the total execution time of the HeteroMPI-HPS application. This includes the time taken to execute the DIPA-2D algorithm and the time taken to determine the best 2D process grid arrangement.

Size of Size of the matrix (n)	HPS (Best)	HeteroMPI-HPS (1t)				HeteroMPI-HPS (2t)				HeteroMPI-HPS (4t)				HeteroMPI -HPS (Best)
		Pre (p× q)	Pre time (s)	GC time (s)	Act time (s)	Pre (p× q)	Pre time (s)	GC time (s)	Act time (s)	Pre (p× q)	Pre time (s)	GC time (s)	Act time (s)	
594	0.04 (2×2, 2 t)	2×15	0.11	3	0.22	**2×2**	**0.13**	**0.22**	**0.03**	2×2	0.17	0.13	0.15	0.25 (2×2, 2 t)
1188	0.15 (2×2, 2 t)	4×2	0.39	4	0.17	**2×2**	**0.31**	**0.47**	**0.15**	2×2	0.5	0.22	0.6	0.61 (2×2, 2 t)
2376	1 (2×2, 2 t)	3×4	2	8	2.84	**2×2**	**1.2**	**0.92**	**0.99**	2×2	3	0.33	3	2 (2×2, 2 t)
4752	7 (2×2, 2 t)	3×4	9	12	13	**2×2**	**6.4**	**1.8**	**8**	2×2	9	0.58	11	4 (2×2, 2 t)
7128	22 (2×2, 2 t)	3×4	26	18	35	**2×2**	**20**	**2.5**	**23**	2×2	20	0.88	28	26 (2×2, 2 t)
9504	52 (2×2, 2 t)	4×4	52	24	102	**2×2**	**32**	**4**	**53**	2×2	40	1.2	56	58 (2×2, 2 t)
11880	92 (2×2, 4 t)	5×4	91	33	186	**2×4**	**65**	**5**	**96**	2×2	70	1.5	97	104 (2×4, 2 t)
14256	149 (2×2, 4 t)	4×4	145	36	221	**2×4**	**95**	**6**	**149**	2×3	133	2	170	157 (2×4, 2 t)
16632	221 (2×2, 4 t)	6×4	218	48	480	**2×4**	**188**	**8**	**225**	2×2	169	3.2	233	236 (2×4, 2 t)
19008	305 (2×2, 4 t)	6×4	302	58	647	**3×4**	**259**	**9**	**316**	2×2	262	2.7	324	329 (2×4, 2 t)
21384	413 (2×2, 4 t)	6×4	407	66	824	**2×4**	**293**	**9**	**421**	2×2	328	3	442	435 (2×4, 2 t)
23760	545 (2×2, 4 t)	6×4	531	76	1064	**2×4**	**391**	**10**	**566**	2×2	488	3.5	583	572 (2×4, 2 t)
26136	706 (2×2, 4 t)	6×4	689	81	1304	**3×4**	**519**	**11**	**787**	2×2	575	4	808	806 (3×4, 2 t)
28512	870 (2×2, 4 t)	6×4	850	84	1573	**3×4**	**720**	**14**	**963**	2×2	760	4.7	987	986 (3×4, 2 t)
30888	1090 (2×2, 4 t)	6×4	1053	103	1890	**3×4**	**829**	**14**	**1181**	3×2	957	5	1226	1205 (3×4, 2 t)
33264	1326 (2×2, 4 t)	6×4	1285	121	2657	3×4	1089	25	1795	**2×2**	**1011**	**13**	**1326**	1668 (2×2, 4 t)
35640	1601 (2×2, 4 t)	6×4	1571	136	3714	3×4	1277	35	2870	**2×2**	**1161**	**25**	**1601**	2123 (2×2, 4 t)
38016	2667 (2×2, 4 t)	6×4	1846	163	5303	3×4	1512	49	5637	**2×2**	**1411**	**39**	**2667**	3642 (2×2, 4 t)
40392	4360 (2×2, 4 t)	6×4	2216	182	6669	3×4	1771	66	7146	**2×2**	**1924**	**56**	**4360**	5537 (2×2, 4 t)

Table 9. HeteroMPI application utilizing the HPS strategy. The number of threads must be pre-configured.

No. of threads run per process shown in brackets
HeteroMPI (1t):
rosebud01 (1), rosebud02 (1), rosebud03 (1), rosebud04 (1), rosebud05 (1), rosebud06 (1), rosebud07 (1), rosebud08 (1)
HeteroMPI (2t):
rosebud01 (1), rosebud02 (1), rosebud03 (2), rosebud04 (2), rosebud05 (2), rosebud06 (2), rosebud07 (2), rosebud08 (2)
HeteroMPI (4t):
rosebud01 (1), rosebud02 (1), rosebud03 (2), rosebud04 (2), rosebud05 (4), rosebud06 (4), rosebud07 (4), rosebud08 (4)

It can be observed that the estimation of the execution times and the optimal 2D process arrangement of the PMM are not accurate. One of the reasons could be the inaccuracy of the communication model used for the shared-memory communications between the processes inside a computer. This issue is currently under investigation.

The experimental results in Table 10 show the execution times of the ScaLAPACK application. The slow computers *rosebud01* and *rosebud02* are not used in the execution of the

Size of the matrix (n)	ScaLAPACK-HPS (seconds)								ScaLAPACK-HPS (Best)
	3×2 (1 t)	**2×8 (1 t)**	**3×8 (1t)**	**3×4 (2t)**	**3×2 (4 t)**	**7×4 (1t)**	**7×2 (2t)**	**4×2 (4t)**	
594	0.3	**0.3**	0.41	0.3	0.4	0.48	0.53	0.5	0.3 (2×8, 1t)
1188	1.3	**1.2**	1	1.2	1.5	1.3	1.4	1.5	1.2 (2×8, 1t)
2376	6	**3.3**	3.45	4.5	5.4	4.8	8.7	7.3	3.3 (2×8, 1t)
4752	30	**12**	15	20	25	24	35	40	12 (2×8, 1t)
7128	77	47	**34**	43	63	63	97	119	34 (3×8, 1t)
9504	158	103	**64**	77	116	110	185	148	64 (3×8, 1t)
11880	288	197	**110**	133	197	175	241	227	110 (3×8, 1t)
14256	467	321	**163**	203	301	247	279	285	163 (3×8, 1t)
16632	712	526	**245**	302	433	349	460	570	245 (3×8, 1t)
19008	1021	704	**344**	425	599	478	621	675	344 (3×8, 1t)
21384	1426	1041	**478**	568	782	634	813	794	478 (3×8, 1t)
23760	1905	1315	**610**	750	1023	830	1067	1260	610 (3×8, 1t)
26136	2509	1799	**827**	980	1314	1122	1512	1428	827 (3×8, 1t)
28512	-	-	**1020**	1213	1606	1272	1730	1764	1020 (3×8, 1t)
30888	-	-	**1290**	1822	1974	1834	2277	2110	1290 (3×8, 1t)
33264	-	-	**1819**	2045	2714	2012	3400	-	1819 (3×8, 1t)
35640	-	-	2412	2615	3623	**2410**	3364	-	2410 (7×4, 1t)
38016	-	-	14221	7416	6001	**3347**	-	-	3347 (7×4, 1t)
40392	-	-	**-**	12159	9244	**8224**	-	-	8224 (7×4, 1t)

Table 10. ScaLAPACK application using the HPS strategy. "-" indicates very large execution times not useful for analysis.

Process Arrangements (ScaLAPACK):

No. of processes run per computer and the number of threads per process shown in brackets
3×2: rosebud05(1,1), rosebud06(1,1), rosebud07(1,1), rosebud08(1,1), rosebud03(1,1), rosebud04(1,1)
2×8: rosebud05(8,1), rosebud06(8,1)
3×8: rosebud05(8,1), rosebud06(8,1), rosebud07(4,1), rosebud08(4,1)
3×4: rosebud05(4,2), rosebud06(4,2), rosebud07(2,2), rosebud08(2,2)
3×2: rosebud05(2,4), rosebud06(2,4), rosebud07(1,4), rosebud08(1,4)
7×4: rosebud05(8,1), rosebud06(8,1), rosebud07(4,1), rosebud08(4,1), rosebud04(2,1), rosebud03(2,1)
7×2: rosebud05(4,2), rosebud06(4,2), rosebud07(2,2), rosebud08(2,2), rosebud04(1,2), rosebud03(1,2)
4×2: rosebud05(2,4), rosebud06(2,4), rosebud07(1,4), rosebud08(1,4), rosebud04(1,2), rosebud03(1,2)

ScaLAPACK application. The application performing the worst is the ScaLAPACK application is the one that uses one process per computer and one thread per process. The results for this combination are only shown for problem sizes (n<28512) because the processors start paging severely beyond these problem sizes. Again, it is observed that as the problem size increases, the optimal number of threads per process increases from 1 to 4.

The experimental results in Table 11 show the execution times of the Heterogeneous ScaLAPACK application. All the computers are used in the execution of the application. The slow computers *rosebud01* and *rosebud02* are however not picked during the execution of the PMM. At the moment, the number of threads must be preconfigured for the Heterogeneous

Size of the matrix (n)	Heterogeneous ScaLAPACK (1t)				Heterogeneous ScaLAPACK (2t)				Heterogeneous ScaLAPACK (4t)			
	Pre (p×q)	Pre time (s)	Ctxt time (s)	Act time (s)	Pre (p×q)	Pre time (s)	Ctxt time (s)	Act time (s)	Pre (p×q)	Pre time (s)	Ctxt time (s)	Act time (s)
594	**5×6**	**0.68**	**6**	**0.4**	2×8	1	1.5	0.6	3×3	1.4	0.9	0.3
1188	**5×6**	**1**	**15**	**1.6**	4×4	1	1.3	1.3	3×3	3	1	1.2
2376	**5×6**	**1.8**	**21**	**6**	3×5	1.6	1.8	6	2×5	5	1.6	6
4752	**5×6**	**3.8**	**25**	**28**	2×7	3.1	2.5	23	2×4	10	1.7	25
7128	**3×9**	**10**	**33**	**77**	2×6	6	4	49	2×4	17	3	63
9504	**2×14**	**19**	**47**	**80**	3×4	14	7	99	2×4	33	6	105
11880	**2×14**	**37**	**68**	**119**	3×4	25	12	163	2×3	56	7	183
14256	**4×7**	**60**	**86**	**246**	2×6	41	21	297	2×3	81	8	285
16632	**2×14**	**98**	**90**	**380**	2×6	62	26	432	2×3	114	10	424
19008	**4×7**	**140**	**107**	**422**	2×6	93	31	541	2×3	156	12	571
21384	**4×7**	**198**	**130**	**558**	2×6	132	36	572	2×3	209	14	764
23760	**4×7**	**271**	**154**	**789**	2×6	184	38	915	2×3	275	17	1018
26136	**2×14**	**358**	**162**	**923**	2×6	240	45	1116	2×3	354	19	1219
28512	**2×14**	**460**	**192**	**1117**	2×6	314	51	1391	2×3	444	22	1572
30888	**2×14**	**591**	**213**	**1557**	2×6	502	56	1690	2×3	559	32	1936
33264	**2×14**	**758**	**612**	**1966**	2×6	626	66	2175	2×3	691	345	2591
35640	**4×7**	**922**	**829**	**2053**	2×6	775	382	2840	2×3	858	642	3445
38016	**4×7**	**1105**	**1377**	**2785**	2×6	957	954	4664	3×2	1065	1094	4919
40392	**4×7**	**1326**	**1645**	**4851**	2×6	1137	1308	7190	3×2	1219	1249	7780

Table 11. Heterogeneous ScaLAPACK applications using the HPS strategy. The number of threads must be pre-configured.

No. of threads run per process shown in brackets
Heterogeneous ScaLAPACK (1t):
rosebud01 (1), rosebud02 (1), rosebud03 (1), rosebud04 (1), rosebud05 (1), rosebud06 (1), rosebud07 (1), rosebud08 (1)
Heterogeneous ScaLAPACK (2t):
rosebud01 (1), rosebud02 (1), rosebud03 (2), rosebud04 (2), rosebud05 (2), rosebud06 (2), rosebud07 (2), rosebud08 (2)
Heterogeneous ScaLAPACK (4t):
rosebud01 (1), rosebud02 (1), rosebud03 (2), rosebud04 (2), rosebud05 (4), rosebud06 (4), rosebud07 (4), rosebud08 (4)

ScaLAPACK application. Given that the number of threads per process is preconfigured, the Heterogeneous ScaLAPACK runtime would then determine the optimal number of processes (also optimal 2D process arrangement). Our future work would involve enhancements to Heterogeneous ScaLAPACK to determine the optimal (process, thread) combination. The results

Size of the matrix (n)	MPI-HDS	HeteroMPI-HDS	MPI-HPS	HeteroMPI-HPS	ScaLAPACK	Heterogeneous ScaLAPACK
594	**0.03**	0.15	0.04	0.25	0.3	0.4
1188	**0.12**	0.52	0.15	0.61	1.2	1.6
2376	**1**	2	1	2	3.3	6
4752	**5**	7	7	4	12	28
7128	**18**	21	22	26	34	77
9504	**42**	46	52	58	64	80
11880	**81**	86	92	104	110	119
14256	**133**	140	149	157	163	246
16632	**191**	201	221	236	245	380
19008	**250**	260	305	329	344	422
21384	**350**	360	413	435	478	558
23760	**453**	467	545	572	610	789
26136	**577**	593	706	806	827	923
28512	**756**	790	870	986	1020	1117
30888	**1006**	1048	1090	1205	1290	1557
33264	**1203**	1460	1320	1668	1819	1966
35640	2091	2993	**1597**	2123	2410	2053
38016	3423	4766	**2654**	3642	3347	2785
40392	5575	7156	**4355**	5537	8224	4851

Table 12. Execution times of all the applications.

of the application in each of the main columns (2-4) are organized as follows. The first sub-column shows the predicted 2D process grid arrangement. The second sub-column shows the predicted time of execution of the PMM. The third sub-column shows the context creation time, which includes the execution of the benchmark code to refresh the speeds of the processors, and the time taken to evaluate all the the possible 2D process grid arrangements. The fourth sub-column shows the actual execution time of the application.

Table 12 shows the results of all the six applications. Table 13 shows the speedup of the best-performing parallel application over the application employing serial matrix-matrix multiplication. The application is run on a single-processor workstation (rosebud01/rosebud02), an SMP (rosebud03/rosebud04), and a computer with cores (rosebud05/rosebud06). The number of threads for the single-processor workstation is set to 1. The single-processor workstations and the SMPs fail for problem sizes $n \geq 9504$.

Consider the case when the number of threads set for the SMP machines and multicore computers is 1. The maximum speedup achieved in the case of SMPs is 15, which is almost linear considering the fact that the number of processors in the network is 16 (excluding the slow processors *rosebud01* and *rosebud02*). The maximum speedup achieved in the case of multicores computers is 17. This is sub-linear since the number of cores in the network is 24.

Size of the matrix (n)	Speedup (rosebud01/rosebud02)	Speedup (rosebud03/rosebud04)		Speedup (rosebud05/rosebud06)	
		# threads = 1	# threads = 2	# threads = 1	# threads = 8
594	7.9	5.3	5.6	2.6	1
1188	14.8	10.5	5	5.4	1
2376	13	7.8	4.3	5	1
4752	20.3	12	6.3	8	1
7128	49.8	11.2	5.9	7.3	1
9504	113.3	14.8	5.9	7.5	1
11880	-	-	-	7.5	1
14256	-	-	-	8	1.04
16632	-	-	-	8.8	1.14
19008	-	-	-	10.8	2.3
21384	-	-	-	16.6	5.4
23760	-	-	-	-	10

Table 13. Speedups of the best-performing application employing the PMM over the application employing sequential matrix-matrix multiplication. "-" indicates failure of the processor.

Consider the case when the number of threads set for the SMP machines and multicore computers is equal to the number of processors and the number of cores respectively. The maximum speedup achieved in the case of SMPs is 5, which is sub-linear considering the fact that the number of processors used in the sequential application is 2 and the number of processors in the network is 16 (excluding the slow processors *rosebud01* and *rosebud02*). The maximum speedup achieved in the case of multicore computers is 10 where the number of cores used in the sequential application is 8 and the number of cores in the network is 24.

9 Conclusions

The conclusions to be drawn from these results are the following:

a. The HDS strategy is the best strategy to use since it allows to fully exploit the increased thread-level parallelism (TLP) provided by the multicore processors. However, for large problem sizes, the non-cartesian nature of the data distribution may lead to excessive communications that can be very expensive. For such cases, the HPS strategy has been shown to outperform it;
b. HeteroMPI is a valuable tool to implement heterogeneous parallel algorithms on HCoMs using the HDS strategy owing to several reasons. It accurately predicts the execution time of

the parallel algorithm, accurately detects the optimal values of the algorithmic parameters such as the total number of processors and the 2D processor grid arrangement;

c. The software (HeteroMPI, Heterogeneous ScaLAPACK) must be enhanced to provide accurate predictions of the optimal (process, thread) combination in the case of the HPS strategy. Since Heterogeneous ScaLAPACK is built on the top of HeteroMPI, the feature only needs to be added to HeteroMPI;
d. No package is currently available using HDS strategy. A package based on this strategy requires great effort implementing the routines and writing the associated performance models. HeteroScaLAPACK is however completely implemented except for the eigenvalue solvers. It uses the legacy ScaLAPACK and its associated performance models are written. In addition, the HeteroMPI-HPS strategy followed by HeteroScaLAPACK have been shown to be quite competitive to the HDS strategy in single core processor clusters [16,17,18] as well as multicore processor clusters;
e. HeteroMPI has been proven to be a valuable tool in single core processor clusters [9] by oversubscribing a fast processor with a number of processes proportional to its speed. From the analysis of the results of this paper, this can be accomplished too in multicore processor clusters but with one limitation: the total number of processes per node cannot be larger than the number of cores of the node, otherwise, the performance falls down drastically due to resource contention of the concurrent processes and loss of some aggregate computational computation power of the heterogeneous cluster. More investigation needs to be done with incoming growth in the number of cores per node.

10 Summary and Future Work

In this document, we present enhancements to two pre-existing strategies of distribution of computations for Heterogeneous Computational Clusters of Multicore Processors (HCoMs). These strategies are called Heterogeneous Process Distribution Strategy (HPS) and Heterogeneous Data Distribution Strategy (HDS) and are used to implement parallel solvers for dense linear algebra problems.

We perform experiments using six applications utilizing the various distribution strategies to perform parallel matrix-matrix multiplication (PMM) on a local HCoM. We then compare the results of execution of these six applications. The results reveal that the two strategies can compete with each other. The MPI applications employing HDS perform the best since they fully exploit the increased thread-level parallelism (TLP) provided by the multicore processors. However, for large problem sizes, the non-cartesian nature of the data distribution may lead to excessive communications that can be very expensive. For such cases, the HPS strategy has been shown to out-perform the HDS strategy. We also conclude that HeteroMPI is a valuable tool to implement heterogeneous parallel algorithms on HCoMs because it provides features that determine optimal values of the algorithmic parameters such as the total number of processors and the 2D processor grid arrangement.

Our future work would involve addition of features to the software (HeteroMPI, Heterogeneous ScaLAPACK) to determine the optimal (process, thread) combination in the HPS strategy. We would also look at improvements to the communication models in these softwares, which would accurately predict the the time of different types of communications, for example, point-to-point, broadcast, gather, scatter etc., between different sets of processors on different

levels. We would then study one of the main linear algebra problems, which are dense systems of linear equations, least squares problems, and eigenvalue problems.

Acknowledgement

Pedro Alonso wishes to acknowledge the support provided by Vicerrectorado de Investigación, Desarrollo e Innovación de la Universidad Politécnica de Valencia, and Generalitat Valenciana.

References

[1]. Ranking of supercomputers according to the LINPACK benchmark. http://www.top500.org.
[2]. J. Dongarra, D. Gannon, G. Fox, and K. Kennedy, "The Impact of Multicore on Computational Science Software," CTWatch Quarterly, Volume 3, No. 1, February 2007.
[3] The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/.
[4] An API for multi-platform shared-memory parallel programming in C/C++ and Fortran. http://openmp.org/wp/.
[5] A. Lastovetsky, "Scientific Programming for Heterogeneous Systems - Bridging the Gap between Algorithms and Applications," Proceedings of the 5th International Symposium on Parallel Computing in Electrical Engineering (PARELEC 2006), Bialystok, Poland, IEEE Computer Society Press, pp. 3-8, 13-17 Sept 2006.
[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, Volume 61, No. 4, pp.520-535, April 2001.
[7] Scalable LAPACK. http://www.netlib.org/scalapack/.
[8] Heterogeneous ScaLAPACK. http://hcl.ucd.ie/project/HeteroScaLAPACK/.
[9] A. Lastovetsky and R. Reddy, "HeteroMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers," Journal of Parallel and Distributed Computing (JPDC), Volume 66, No. 2, pp.197-220, Elsevier, 2006. http://hcl.ucd.ie/project/HeteroMPI/.
[10] A. Lastovetsky and R. Reddy, "On Performance Analysis of Heterogeneous Parallel Algorithms," In Parallel Computing, Volume 30, No. 11, pp.1195-1216, 2004.
[11] A. Lastovetsky and R. Reddy, "Data Partitioning with a Functional Performance Model of Heterogeneous Processors", In International Journal of High Performance Computing Applications, Volume 21, No. 1, pp. 76-90, SAGE Publications, 2007.
[12]. A. Lastovetsky and R. Reddy, "Efficient Distributed Algorithms of Optimal Data Partitioning for Parallel Computing on Heterogeneous Processors Based on Partial Estimation of their Functional Performance Models," Technical Report, University College Dublin, 2009.
[13] A. Lastovetsky, D. Arapov, A. Kalinov, and I. Ledovskih, "A Parallel Language and Its Programming System for Heterogeneous Networks," Concurrency: Practice and Experience, Volume 12, No. 13, pp.1317-1343, November 2000.
[14] A. Lastovetsky, "Adaptive Parallel Computing on Heterogeneous Networks with mpC," Parallel Computing, Volume 28, No. 10, pp.1369-1407, October 2002.
[15] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix Multiplication on Heterogeneous Platforms", IEEE Transactions on Parallel and Distributed Systems, Volume 12, No. 10, pp.1033-1051, 2001.

[16] P. Alonso and A. M. Vidal, “Cauchy-like system solution on multicore platforms,” Workshop on State-of-the-Art in Scientific and Parallel Computing (Para 2008), May 13-16, NTNU, Trondheim, Norway.
[17] R. Reddy, A. Lastovetsky, and P. Alonso, “Heterogeneous PBLAS: Optimization of PBLAS for Heterogeneous Computational Clusters,” In Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008), pp. 73-80, IEEE Computer Society Press.
[18] R. Reddy, A. Lastovetsky, and P. Alonso, “Scalable Dense Factorizations for Heterogeneous Computational Clusters,” In Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC 2008), pp. 49-56, IEEE Computer Society Press.
[19] R. Reddy, A. Lastovetsky, and P. Alonso, “Parallel solvers for dense linear systems for heterogeneous computational clusters,” In Proceedings of the 23rd International Parallel and Distributed Processing Symposium (IPDPS 2009).