Distributed Data Partitioning for Heterogeneous Processors Based on Partial Estimation of their Functional Performance Models

Alexey Lastovetsky, Ravi Reddy

School of Computer Science and Informatics, University College Dublin,
Belfield Dublin 4, Ireland
{Alexey.Lastovetsky, Manumachu.Reddy}@ucd.ie

Abstract. The paper presents a new data partitioning algorithm for parallel computing on heterogeneous processors. Like traditional functional partitioning algorithms, the algorithm assumes that the speed of the processors is characterized by speed functions rather than speed constants. Unlike the traditional algorithms, it does not assume the speed functions to be given. Instead, it uses a computational kernel to estimate the speed functions of the processors for different problem sizes during its execution. This makes the algorithm distributed as its execution involves all the heterogeneous processors. The algorithm does not construct the complete speed function for each processor but rather builds and uses their partial estimates sufficient for optimal data distribution with a given accuracy. The low execution cost of this algorithm makes it ideal for employment in self-adaptable applications. Experiments with a parallel matrix multiplication application employing this algorithm are performed on a local heterogeneous computational cluster. The results show that the algorithm converges very fast and that its execution time is several orders of magnitude less than the total execution time of the application.

Keywords: distributed algorithms, data partitioning algorithms, functional performance models, heterogeneous platforms

1 Introduction

Conventional data partitioning algorithms for parallel computing on heterogeneous processors [1-2] are based on a 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. The constant characterizing the performance of the processor is typically its relative speed demonstrated during the execution of a serial benchmark code solving locally the core computational task of some given size.

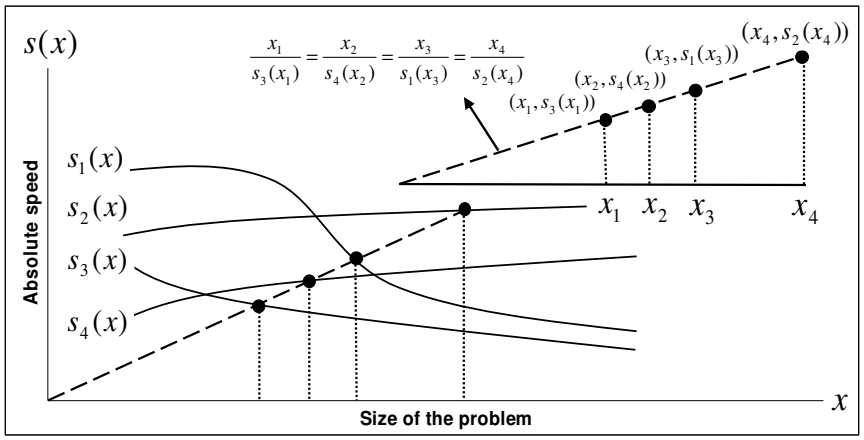


Fig. 1. Optimal data distribution showing the geometric proportionality of the number of chunks to the speed of the processor.

The traditional constant performance models (CPMs) proved to be accurate enough for heterogeneous distributed memory systems if partitioning of the problem results in a set of computational tasks that fit into the main memory of the assigned processors. But these models become less accurate in the presence of paging. The functional performance model (FPM) of heterogeneous processors proposed and analyzed in [3] has proven to be more realistic than the CPMs because it integrates many important features of heterogeneous processors such as the processor heterogeneity, the heterogeneity of memory structure, and the effects of paging. The algorithms employing it therefore distribute the computations across the heterogeneous processors more accurately than the algorithms employing the CPMs. Under this model, the speed of each processor is represented by a continuous function of the size of the problem. This model is application centric because, generally speaking, different applications will characterize the speed of the processor by different functions.

The problem of distributing independent chunks of computations over a unidimensional arrangement of heterogeneous processors using this FPM has been studied in [3]. It can be formulated as follows: Given n independent chunks of computations, each of equal size (i.e., each requiring the same amount of work), how can we assign these chunks to p ($p<n$) physical processors P_1, P_2, ..., P_p with their respective full FPMs represented by speed functions $s_1(x)$, $s_2(x)$, ..., $s_p(x)$ so that the workload is best balanced? An algorithm solving this problem with a complexity of $O(p \times \log_2 n)$ is also proposed in [3]. This and other similar algorithms, which relax the restriction of bounded heterogeneity of the processors [4] and which are not sensitive to the shape of speed functions [5], are based on the observation that the optimal data distribution points $(x_1, s_1(x_1))$, $(x_2, s_2(x_2))$, …, $(x_p, s_p(x_p))$ lie on a straight line passing through the origin of the coordinate system and are the intersecting points of this line with the graphs of the speed functions of the processors. This is shown in Figure 1.

These algorithms are used as building blocks in algorithms solving more complicated linear algebra kernels such as the dense factorizations [6].

The cost of experimentally building the full FPM of a processor, i.e., the FPM for the full range of problem sizes, is very high. This is due to several reasons. To start with, the accuracy of the FPM depends on the number of experimental points used to build it. The larger the number, the more accurate the FPM is. However, there is a cost associated with obtaining an experimental data point, which requires execution of a computational kernel for a specified problem size. This cost is especially high for problem sizes in the region of paging. Also, the number of experimental points required to build the full FPM increases remarkably as the number of parameters used to represent the problem size increases, as shown in the experimental results in this paper.

The problem of minimization of the cost of experimentally building the full FPM of the processor has been studied recently proposing a relatively efficient sub-optimal solution [7]. However, even if an ideal optimal procedure becomes available to build approximations of the FPM of heterogeneous processors, the fact remains that the cost of building the full FPM is too high to forbid the use of data partitioning algorithms, employing the full FPM, in self-adaptable applications.

The paper presents a new algorithm of data partitioning for parallel computing on heterogeneous processors. Like traditional functional partitioning algorithms, the algorithm assumes that the speed of the processors is characterized by speed functions rather than speed constants. Unlike the traditional algorithms, it does not assume the speed functions to be given. Instead, it uses a computational kernel to estimate the speed functions of the processors for different problem sizes during its execution. This makes the algorithm distributed as its execution involves all the heterogeneous processors. The algorithm does not construct the complete speed function for each processor but rather builds and uses their partial estimates sufficient for optimal data distribution. The proposed algorithm does not return a partitioning perfectly balancing the load of the processors but a partitioning balancing their load with a given accuracy.

Using experimental results for parallel matrix multiplication on a local heterogeneous computational cluster, we demonstrate that the execution time of the proposed distributed partitioning algorithm is several orders of magnitude less than the total execution time of the parallel application, thereby making it very suitable for employment in self-adaptable applications.

The rest of the paper is organized as follows. In Section 2, we present the contribution of this paper, which is the distributed iterative partitioning algorithm. This is followed by experimental results on a local heterogeneous computing cluster in Section 3. For the experiments, we use a parallel matrix multiplication application employing the data partitioning algorithm. Finally, we present numerical results demonstrating the efficiency of the distributed iterative partitioning algorithm.

2 Distributed Functional Partitioning Algorithm (DFPA)

The data partitioning problem that we are trying to solve can be formulated as follows:

- Given
  - A set of n independent units of computation each of equal size (i.e., each requiring the same amount of work);
  - A set of p ($p<n$) processors P_1, P_2, ..., P_p, whose speeds of processing x units, $s_i=s_i(x)$, can be obtained by measuring the execution time, $t_i(x)$, of a computational kernel, $s_i(x)=x/t_i(x)$,
  - ε, a required relative accuracy of the solution;
- Partition the set of computation units into p subsets so that
  - There is one-to-one mapping between the partitions and the processors, and
  - $\max_{1\le i,j\le p}\left(\left|\frac{t_i(n_i)-t_j(n_j)}{t_i(n_i)}\right|\right)\le\varepsilon$, where n_i is the number of computation units allocated to processor P_i ($1\le i\le p$).

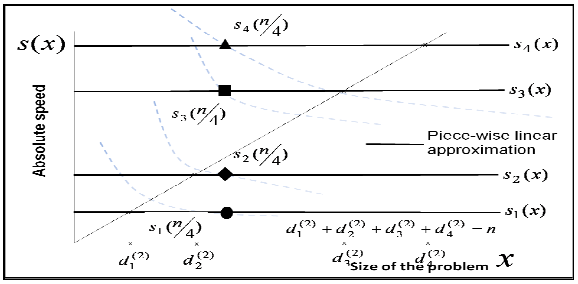
Thus, the problem we study is to balance the load of heterogeneous processors with a given accuracy. The fundamental assumption, which makes efficient solution of this problem particularly difficult, is that the speeds of the processors are not known a priori. Therefore, if a partitioning algorithm needs the speed of processing of a given number of computation units by one or the other processor, it has to execute the corresponding number of units on this processor. Our solution to this problem is the following distributed data partitioning algorithm.

Distributed Functional Partitioning Algorithm (DFPA): The inputs to the algorithm are

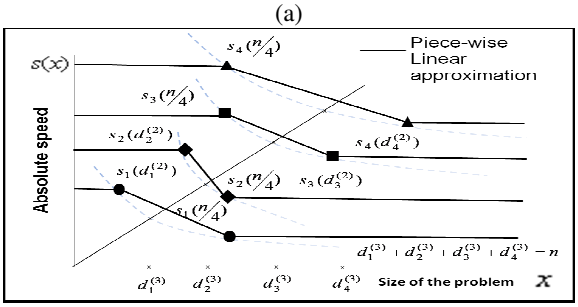
- n, the number of computation units;
- p ($p<n$) processors P_1, P_2, ..., P_p;
- ε, the termination criterion.

The output d is an integer array of size p, the i-th element of which is the number of computation units allocated to processor i. The algorithm can be summarized as follows:

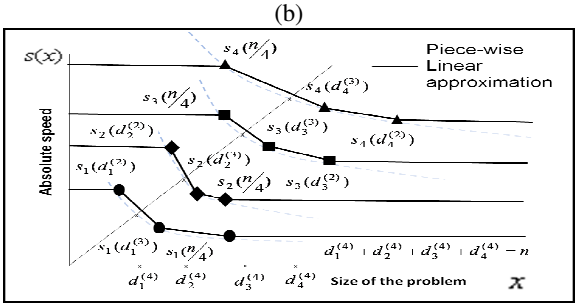
- **Initialization**:
  - All the p processors execute n/p computation units in parallel;
  - The execution times are gathered on processor P_1, $(t_1,\ldots,t_p)\leftarrow(t_1(n/p),\ldots,t_p(n/p))$;
  - If $\max_{1\le i,j\le p}\left(\left|\frac{t_i(n/p)-t_j(n/p)}{t_i(n/p)}\right|\right)\le\varepsilon$ then the even distribution of computations solves the problem and the algorithm stops;
  - Otherwise, processor P_1 calculates the absolute speeds of the processors, $s_i(n/p)=(n/p)/t_i$ for $1\le i\le p$ and builds the first approximation of their FPMs in the form of constant models, $s_i(x)=s_i(n/p)$, as illustrated in Figure 2.
- **Iterating:** At each step,



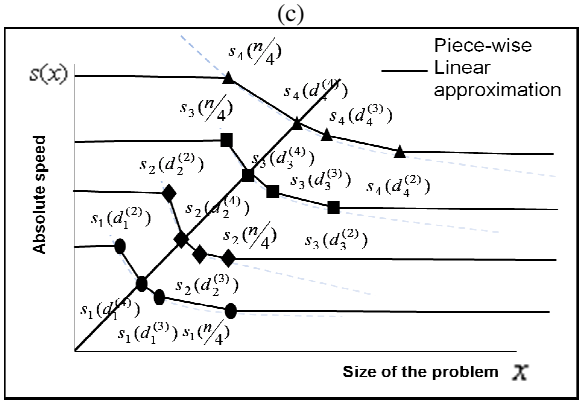
(a)



(b)



(c)



(d)

Fig. 2. Steps of the distributed functional partitioning algorithm (DFPA) illustrated using four heterogeneous processors. The dotted curves are real-life speed functions.

- Using the data partitioning algorithm [3], processor P_1 calculates a new distribution of computation units, $(d_1,\ldots,d_p)$, which will be optimal for the current approximations of the FPMs, and then sends a message to each processor P_i informing the latter of its new allocation of computation units, d_i ($1 \le i \le p$);
- Each processor P_i then executes d_i computation units in parallel with the other processors, $1 \le i \le p$;
- The execution times are gathered on processor P_1, $(t_1,\ldots,t_p) \leftarrow (t_1(d_1),\ldots,t_p(d_p))$;
- If $\max_{1\le i,j\le p}\left(\left|\frac{t_i - t_j}{t_i}\right|\right) \le \varepsilon$, then the current distribution of computation units, $(d_1,\ldots,d_p)$, solves the problem and the algorithm stops;
- Otherwise, processor P_1 calculates the absolute speeds, which the processors demonstrated for this distribution of computation units, $s_i(d_i) = \frac{d_i}{t_i}$ ($1 \le i \le p$), and uses these newly obtained points of the FPMs of processors P_i, $(d_i, s_i(d_i))$, to build their more accurate piecewise linear approximations (as illustrated in Figure 2). Namely, let $\{(d_i^{(j)}, s_i(d_i^{(j)}))\}_{j=1}^{m}$ ($d_i^{(1)} < \ldots < d_i^{(m)}$) be the experimentally obtained points of $s_i(x)$ used to build its current piecewise linear approximation, then
  - If $d_i < d_i^{(1)}$, then the line segment $(0, s_i(d_i^{(1)})) \rightarrow (d_i^{(1)}, s_i(d_i^{(1)}))$ of this approximation will be replaced by two connected line segments $(0, s_i(d_{(i)})) \rightarrow (d_i, s_i(d_i))$ and $(d_i, s_i(d_i)) \rightarrow (d_i^{(1)}, s_i(d_i^{(1)}))$;
  - If $d_i > d_i^{(m)}$, then the line $(d_i^{(m)}, s_i(d_i^{(m)})) \rightarrow (\infty, s_i(d_i^{(m)}))$ of this approximation will be replaced by the line segment $(d_i^{(m)}, s_i(d_i^{(m)})) \rightarrow (d_i, s_i(d_i))$ and the line $(d_i, s_i(d_i)) \rightarrow (\infty, s_i(d_i))$;
  - If $d_i^{(k)} < d_i < d_i^{(k+1)}$, the line segment $(d_i^{(k)}, s_i(d_i^{(k)})) \rightarrow (d_i^{(k+1)}, s_i(d_i^{(k+1)}))$ will be replaced by two connected line segments

$(d_i^{(k)}, s_i(d_i^{(k)})) \rightarrow (d_i, s_i(d_i))$ and

$(d_i, s_i(d_i)) \rightarrow (d_i^{(k+1)}, s_i(d_i^{(k+1)}))$.

– Then, the algorithm proceeds to the next step.

Proposition. *Given the full FPMs of the processors P_1, P_2, ..., P_p satisfy the assumptions about their shape stated in [3], the DFPA algorithm always converges.*

Space limitations do not allow us to give the full formal proof of this proposition. In brief, its main points are as follows. First of all, by construction, the piecewise linear approximations of the full FPMs used in the algorithm will satisfy the same assumptions about their shape as the full FPMs themselves. Therefore, at each iteration step, application of algorithm [3] to the set of approximate FPMs will be successful and return the optimal solution for these approximate FPMs. Second, each next iteration step of the algorithm results in more accurate approximation of the segments of the full FPMs that contain the points of the optimal solution. Therefore, after a number of iterations, the approximations of the full FPMs will become accurate enough in order algorithm [3] to return a solution sufficiently close to the optimal one.

Figure 2 illustrates the operation of the DFPA algorithm using an example with four heterogeneous processors (P_1,P_2,P_3,P_4).

3 Experimental Results

We use a small heterogeneous local network of 16 different Linux processors (hcl01-hcl16) for the experiments. The specifications of the network are available at the URL http://hcl.ucd.ie/Hardware/Cluster+Specifications. The network is based on 2 Gbit Ethernet with a switch enabling parallel communications between the computers. The software used is MPICH-1.2.5 and ATLAS [8], which provides an optimized BLAS library.

Figure 3(a) shows the parallel matrix multiplication application. It implements matrix operation $C=A\times B$, multiplying matrix A and matrix B, where A, B, and C are dense square matrices of size $n\times n$ matrix elements on a network of p heterogeneous processors. We use a 1D processor arrangement of size 3 for illustration purposes. Each element is a square matrix block of size $b\times b$ (the value of b used is 16). The matrices A and C must be horizontally sliced such that the height of the slice is proportional to the speed of the processor owning the slice. All the processors contain all the elements of matrix B. We assume that only one process is configured to execute on a processor. We purposely choose an application with no communications because the goal of the experiments is not to show how to multiply matrices in parallel but to demonstrate the practical speed of convergence of the distributed

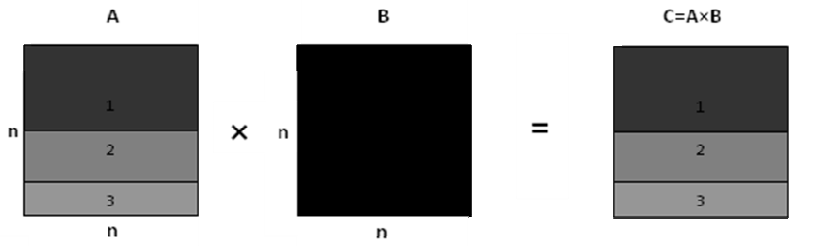
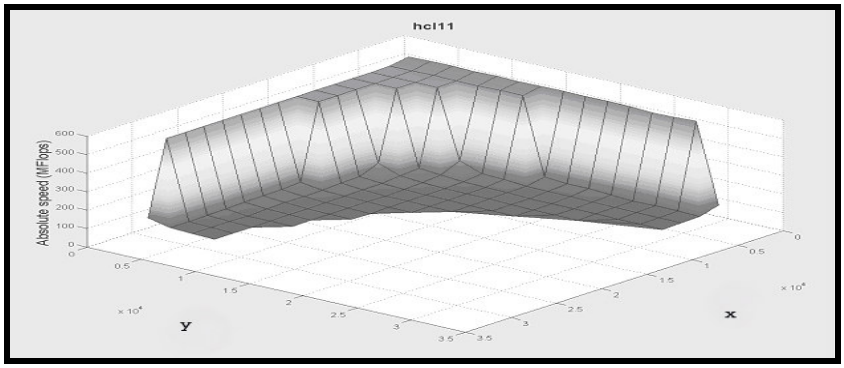


Fig. 3. (a) Matrix operation $C=A\times B$ on a network of three heterogeneous processors. Matrices A and C are horizontally sliced such that the height of the slice (n_b) is proportional to the speed of the processor. (b) The computational kernel (shown here for processor 2 for example) performs a matrix update of A_b of size $n_b\times 1$ and B_b of size $1\times n$ to give a dense matrix C_b of size $n_b\times n$. The matrix elements represent $b\times b$ matrix blocks.

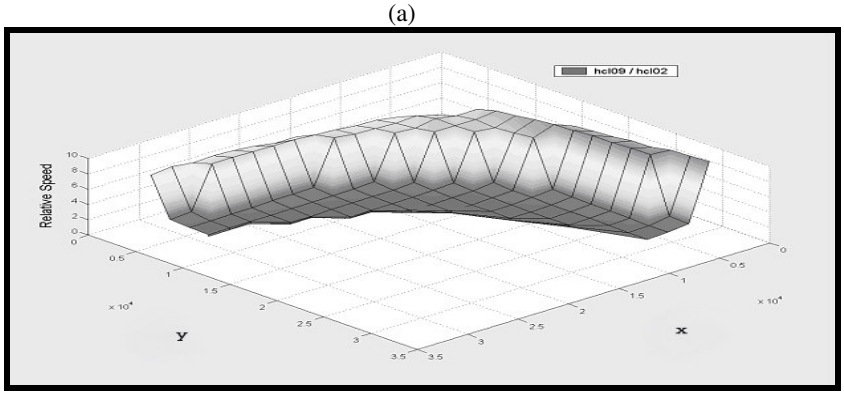
partitioning algorithm. The results will not differ significantly for more complicated algorithms involving communications.

For this application, the core computational kernel performs a matrix update of a matrix C_b of size $n_b\times n$ using A_b of size $n_b\times 1$ and B_b of size $1\times n$ as shown in Figure 3(b). Each element is a square matrix block of size $b\times b$. The size of the problem is represented by two parameters, n_b and n. The total number of matrix elements stored on each processor will be $(2\times n_b\times n+n\times n)$. We use a combined computation unit, which is made up of one addition and one multiplication, to express the volume of computation. If n is large enough, the total number of computation units needed to solve this problem will be approximately equal to $n_b\times n$ (namely, multiplications of two $b\times b$ matrices). Therefore, the absolute speed of the processor exposed by the application when solving the problem of size (n_b, n) can be calculated as $n_b\times n$ divided by the execution time of the matrix update. This gives us a function, f: $\mathbf{N}^2 \rightarrow \mathbf{R}_+$, mapping problem sizes to speeds of the processor. The FPM of the processor is obtained by continuous extension of function f: $\mathbf{N}^2 \rightarrow \mathbf{R}_+$ to function g: $\mathbf{R}_+^2 \rightarrow \mathbf{R}_+$ (f(n,m)=g(n,m) for any (n,m) from $\mathbf{N}^2$). Figure 4(a) depicts this function for one of the processors, *hcl11*, used in experiments. Figure 4(b) shows the relative speed of two processors, *hcl09* and *hcl02*, calculated as the ratio of their absolute speeds. One can see that the relative speed varies significantly depending on the value of variables x and y (the variables represent n_b and n).

The heterogeneity of the network due to the heterogeneity of the processors is calculated as the ratio of the absolute speed of the fastest processor to the absolute speed of the slowest processor. For example, consider the benchmark code of a local DGEMM update of two matrices 2560×16 and 16×2560, the absolute speeds of the



(a)



(b)

Fig. 4. (a) The absolute speed of a processor 'hcl11' as a function of the size of the computational task of updating a dense $x \times y$ matrix. (b) The relative speed of two processors ('hcl09', 'hcl02') calculated as the ratio of their absolute speeds.

processors hcl01-hcl16 in million flop/s performing this update are {7696, 5196, 7852, 14418, 8000, 8173, 7288, 7396, 9037, 8987, 13661, 14194, 11182, 14410, 12008, 15257}. As one can see, hcl16 is the fastest processor and hcl02 is the slowest processor. The heterogeneity is therefore 3.

We compare the efficiency of the DFPA-based matrix multiplication application with the application based on the Full-Functional-Model Partitioning Algorithm (FFMPA). The difference between these applications is that the FFMPA-based one uses pre-built full FPMs of the processors for partitioning the matrices. More specifically, it uses the piecewise linear approximation of the full FPMs obtained with the GBBP procedure [7], which employs the same computational kernel as the DFPA-based application. Unlike the FFMPA-based application, the DFPA-based application does not need the FPMs of the processors as input. In all our experiments, the FFMPA returned the same data distribution as the DFPA.

Figure 5 shows the execution times of the sequential application and the parallel applications employing the FFPMA and DFPA and solving the same matrix

multiplication problem. The sequential application uses optimized BLAS library (ATLAS) and is executed on the fastest processor (*hcl09*). The execution of the

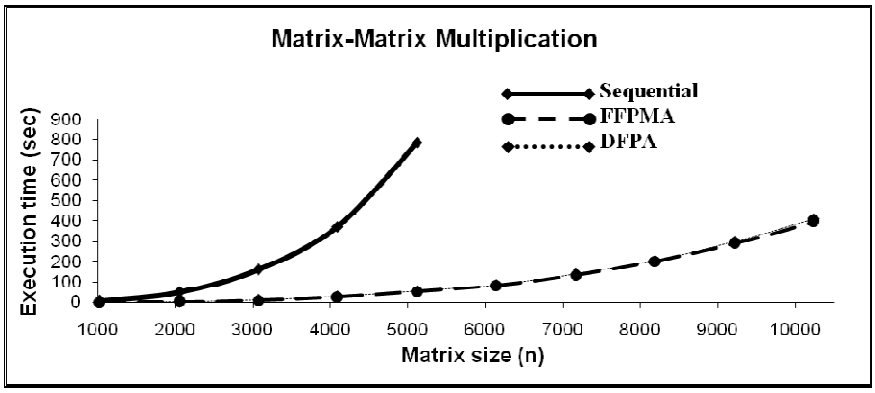


Fig. 5. Execution times of sequential and parallel applications with FFPMA and DFPA solving the same matrix multiplication problem.

Table 1. Execution times of the parallel matrix multiplication application employing FFPMA and DFPA.

Size of the matrix (n)	Number of iterations of DFPA	DFPA execution time (sec)	Execution time using DFPA (sec)	Execution time using FFPMA (sec)
1024	2	0.06	0.2	0.2
2048	2	0.09	2.2	1.9
3072	2	0.3	9.9	8.5
4096	5	2	28	25.3
5120	5	3	53.3	50.5
6144	5	3	84.4	80.7
7168	5	4	137.7	132.4
8192	5	5	204.3	199.7
9216	5	7	295.3	287.6
10240	5	11	405.9	393.3

parallel matrix multiplication application consists of two parts. Firstly, all the processors execute the DFPA/FFPMA data partitioning algorithm to partition the matrices and then they perform the parallel matrix multiplication itself. For problem sizes (n>5120), the sequential application fails due to the problem size exceeding the memory limit of the processor. One can conclude that the parallel applications outperform the sequential application.

Table 1 shows the execution times of the parallel matrix multiplication applications employing the FFPMA and the DFPA. The second column shows the number of iterations of DFPA. The third column shows the execution time of the DFPA. The

fourth column shows the total execution time of the DFPA-based application. This includes the execution time of the DFPA. The fifth column shows the total execution time of the parallel application employing the FFPMA algorithm, which obviously does not include the time of construction of the FPMs of the processors.

One can see that the execution times of the parallel applications employing the FFPMA and DFPA differ only marginally. The difference is the execution time of the DFPA algorithm shown in the third column of Table 1. Most of it is spent in the partial estimation of the FPMs of the processors. It should be noted that the execution time of the parallel application employing the FFPMA does not take into consideration the time taken to build the full FPMs of the processors.

The execution time taken to build the full FPMs of the processors, which are used in the FFPMA-based application, is 425 seconds. The range of problem sizes, (n_b, n), used for building them satisfy the inequalities, $n_b \leq 10240$, $n \leq 10240$, and $n_b \leq n$. One can see that the execution time is significant compared to the DFPA execution times shown in the third column. The maximum number of experimental points used to build the full FPMs for this range is 60. This is compared to a maximum of 6 using DFPA (number of iterations plus one shown in column 2 of Table 1).

Thus, we can conclude that the DFPA converges very fast and its execution time is several orders of magnitude less than the execution time of the application. It is also efficient in terms of the number of experimental points.

This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/IN.1/I2054.

References

[1] Kalinov, A., Lastovetsky, A.: Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. Journal of Parallel and Distributed Computing, 61(4), 520--535 (2001)

[2] Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Matrix Multiplication on Heterogeneous Platforms. IEEE Transactions on Parallel and Distributed Systems. 12(10), 1033--1051 (2001)

[3] Lastovetsky, A., Reddy, R.: Data Partitioning with a Functional Performance Model of Heterogeneous Processors. International Journal of High Performance Computing Applications. 21(1), 76--90 (2007)

[4] Lastovetsky, A., Reddy, R.: Data Partitioning for Multiprocessors with Memory Heterogeneity and Memory Constraints. Scientific Programming. 13(2), 93--112 (2005)

[5] Lastovetsky, A., Reddy, R.: Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers. In: 18th International Parallel and Distributed Processing Symposium, IEEE Computer Society (2004)

[6] Lastovetsky, A., Reddy, R.: Data distribution for dense factorization on computers with memory heterogeneity. Parallel Computing, 33(12), 757--779 (2007)

[7] Lastovetsky, A., Reddy, R., Higgins, R.: Building the Functional Performance Model of a Processor. In: 21st Annual ACM Symposium on Applied Computing, pp.746--753, ACM Press, 2006

[8] Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/