Data Partitioning with a Functional Performance Model of Heterogeneous Processors

Alexey Lastovetsky and Ravi Reddy

International Journal of High Performance Computing Applications 2007; 21; 76
DOI: 10.1177/1094342006074864

DATA PARTITIONING WITH A FUNCTIONAL PERFORMANCE MODEL OF HETEROGENEOUS PROCESSORS

Alexey Lastovetsky
Ravi Reddy

SCHOOL OF COMPUTER SCIENCE AND INFORMATICS, UNIVERSITY COLLEGE DUBLIN, BELFIELD, DUBLIN 4, IRELAND (ALEXEY.LASTOVETSKY@UCD.IE, MANUMACHU.REDDY@UCD.IE)

Abstract

In this paper, we address the problem of optimal distribution of computational tasks on a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds vary with the problem size. We propose a functional performance model of heterogeneous processors that integrates many essential features of a network of heterogeneous computers having a major impact on its performance such as the processor heterogeneity, the heterogeneity of memory structure, and the effects of paging. Under this model, the speed of each processor is represented by a continuous function of the size of the problem whereas traditional models use single numbers to represent the speeds of the processors. We formulate a problem of partitioning of an n-element set over p heterogeneous processors using this model and design an algorithm of the complexity $O(p \times \log_2 n)$ solving the problem.

Key words: heterogeneous systems, scheduling and task partitioning, load balancing and task assignment, high performance computing.

1 Introduction

In this paper, we deal with the problem of optimal distribution of computational tasks across heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds vary with the problem size.

A number of algorithms of parallel solution of scientific and engineering problems on heterogeneous networks of computers (HNOCs) have been designed and implemented (Crandall and Quinn 1993, 1995; Beaumont et al. 2001b; Kalinov and Lastovetsky 2001). They use different performance models of HNOCs but all the models represent the speed of a processor by a single positive number, and computations are distributed over the processors such that their volume is proportional to this speed of the processor. Cierniak et al. (1997) use the notion of normalized processor speed (NPS) in their machine model to solve the problem of scheduling parallel loops at compile time for HNOCs. NPS is a single number and is defined as the ratio of time taken to execute on the processor under consideration, with respect to the time taken on a base processor. In Beaumont et al. (2001b) and Petitet and Dongarra (1999), normalized cycle-times are used, i.e. application dependent elemental computation times, which are computed via small-scale experiments (repeated several times, with an averaging of the results). Several scheduling and mapping heuristics have been proposed to map task graphs onto HNOCs (Tan et al. 1997; Maheswaran and Siegel 1998; Iverson and Ozguner 1998). These heuristics employ a model of a heterogeneous computing environment that uses a single number for the computation time of a subtask on a machine. Yan, Zhang and Song (1996) use a two-level model to study performance predictions for parallel computing on HNOCs. The model uses two parameters to capture the effects of an owner workload. These are the average execution time of the owner task on a machine and the average probability of the owner task arriving on a machine during a given time step.

Thus traditional heterogeneous parallel and distributed algorithms implicitly assume that the relative speed of the processor does not depend on the size of the computational task solved by the processor. This assumption can be quite satisfactory if the code executed by the processors fully fits into the main memory. But as soon as the restriction is relaxed, it will not be true.

First of all, the processors may have significantly different sizes of main memory and the partitioning of the problem may result in some computational tasks not fitting into the main memory of the assigned processor. In this case, solution of the computational task of any fixed size does not guarantee accurate estimation of the relative speed of the processors. The point is that beginning from

The International Journal of High Performance Computing Applications, Volume 21, No. 1, Spring 2007, pp. 76–90
DOI: 10.1177/1094342006074864

some problem size, the task of the same size will still fit into the main memory of some processors and stop fitting into the main memory of others, causing the paging and visible degradation of the speed of these processors. This means that their relative speed will start significantly changing in favor of non-paging processors as soon as the problem size exceeds the critical value.

Secondly, even if the processors of different architectures have almost the same size of main memory, they may employ different paging algorithms resulting in different levels of speed degradation for the task of the same size, which again means the change of their relative speed as the problem size exceeds the threshold causing the paging.

Thus, taking account of memory heterogeneity and the effects of paging significantly complicates the design of algorithms distributing computations in proportion with the relative speed of heterogeneous processors. One approach to this problem is just to avoid the paging as it is normally done in the case of parallel computing on homogeneous multi-processors. However, avoiding paging in local and global heterogeneous networks may not make sense because in such networks it is likely that one processor with paging will be running faster than other processors without paging. It is even more difficult to avoid paging in the case of distributed computing on global networks. There may not be a server available to solve the task of the size you need without paging.

Therefore, to achieve acceptable accuracy of distribution of computations across heterogeneous processors in the possible presence of paging, a more realistic performance model of a set of heterogeneous processors is needed. In this paper, we suggest a model where the speed of each processor is represented by a continuous function of the problem size. This model is application centric in the sense that, generally speaking, different applications will characterize the speed of the processor by different functions.

The rest of the paper is organized as follows. In Section 2, we present the functional performance model. In Section 3, we investigate the simple problem of optimal partitioning of an n-element set over p heterogeneous processors with the functional model and design an algorithm of its solution of the complexity $O(p \times \log_2 n)$. We then apply this set partitioning algorithm to multiplication of large matrices on a cluster of heterogeneous computers. In Section 4, we present results of experiments with this application. Section 5 concludes the paper.

2 Functional Performance Model

Under the functional performance model, the speed of each processor is represented by a continuous function of the problem size.

The speed is defined as the number of computation units performed by the processor per one time unit. The model is application specific. In particular, this means that the computation unit can be defined differently for different applications. The important requirement is that the computation unit does not have to vary during the execution of the application. An arithmetical operation and the matrix update $a = a + b \times c$, where a, b, and c are $r \times r$ matrices of fixed size r, give us examples of computation units.

The problem size is understood as a set of one, two or more parameters characterizing the amount and layout of data stored and processed during the execution of the computational task [as compared with the notion of problem size as the number of basic computations in the best sequential algorithm to solve the problem on a single processor (Kumar et al. 1994)]. The number and semantics of the problem size parameters are problem- or even application-specific. It is assumed that the amount of stored data will increase with the increase of any of the problem size parameters.

For example, the size of the problem of multiplication of two dense square $n \times n$ matrices can be represented by one parameter, n. During solution of the problem, three matrices will be stored and processed. So the total number of elements to store and process will be $3 \times n^2$. In order to compute one element of the resulting matrix, the application uses n multiplications and $(n - 1)$ additions. So, in total $(2 \times n - 1) \times n^2$ arithmetical operations are needed to solve the problem. If n is large enough, the number can be approximated by $2 \times n^3$. Alternatively, a combined computation unit, which is made up of one addition and one multiplication, can be used to express the volume of computation needed to multiply two large square $n \times n$ matrices. In this case, the total number of computation units will be approximately equal to n^3. Therefore, the speed of the processor demonstrated by the application when solving the problem of size n can be calculated as n^3 (or $2 \times n^3$) divided by the execution time of the application. This gives us a function from the set of natural numbers representing problem sizes into the set of nonnegative real numbers representing speeds of the processor, $f: \mathbf{N} \rightarrow \mathbf{R}_+$. The functional performance model of the processor is obtained by continuous extension of function $f: \mathbf{N} \rightarrow \mathbf{R}_+$ to function $g: \mathbf{R}_+ \rightarrow \mathbf{R}_+$ ($f(n) = g(n)$ for any n from $\mathbf{N}$).

Another example is the problem of multiplication of two dense rectangular $n \times k$ and $k \times m$ matrices. The size of this problem is represented by three parameters, n, k, and m. The total number of matrix elements to store and process is $(n \times k + k \times m + n \times m)$. The total number of arithmetical operations needed to solve this problem is $(2 \times k - 1) \times n \times m$. If k is large enough, the number can be approximated by $2 \times k \times n \times m$. Alternatively, a com-

bined computation unit, which is made up of one addition and one multiplication, can be used to express this volume of computation. In this case, the total number of computation units will be approximately equal to $k \times n \times m$. Therefore, the speed of the processor exposed by the application when solving the problem of size (n, k, m) can be calculated as $k \times n \times m$ (or $2 \times k \times n \times m$) divided by the execution time of the application. This gives us a function, f: $\mathbf{N}^3 \rightarrow \mathbf{R}_+$, mapping problem sizes to speeds of the processor. The functional performance model of the processor is obtained by continuous extension of function f: $\mathbf{N}^3 \rightarrow \mathbf{R}_+$ to function g: $\mathbf{R}_+^3 \rightarrow \mathbf{R}_+$ ($f(n) = g(n)$ for any n from $\mathbf{N}^3$).

Thus, under the proposed functional model, the speed of the processor is represented by a continuous function of the problem size. Moreover, we can make some further assumptions about the shape of the function. Namely, we can realistically assume that along each of the problem size variables, either the function is monotonically decreasing, or there exists point x such that:

- On the interval $[0, x]$, the function is
  - monotonically increasing,
  - concave, and
  - any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point.
- On the interval $[x, \infty)$, the function is monotonically decreasing.

We have conducted numerous experiments with diverse scientific kernels and computers, and in all the experiments the speed of the processor could be approximated accurately enough by a function satisfying the above assumptions (within the accuracy of measurements). Some typical observed shapes of the speed function are given in this paper.

An alternative approach is to use a piecewise constant function in order to represent the dependence of the speed of the processor on the problem size (Drozdowski and Wolniewicz 2003). There are at least two reasons behind the proposal to represent the speed of the processor by a continuous function of the problem size.

First of all, we want the model to adequately reflect the behavior of common, not very carefully designed applications. Consider the experiments with a range of applications using memory hierarchy in different ways that are presented by Lastovetsky and Twamley (2005) and shown in Figure 1.

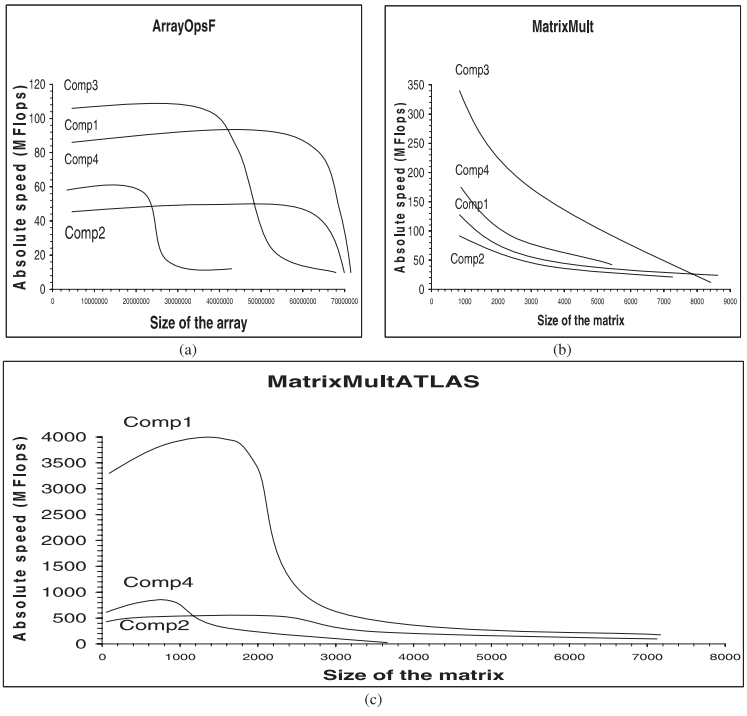

Fig. 1 The effect of caching and paging in reducing the execution speed of each of the four applications run on network of heterogeneous computers shown in Table 1. (a) ArrayOpsF, (b) MatrixMult, and (c) MatrixMultATLAS.

Carefully designed applications **ArrayOpsF** and **MatrixMultAtlas**, which efficiently use memory hierarchy, demonstrate quite a sharp and distinctive performance curve of dependence of the absolute speed on the problem size. For these applications, the speed of the processor can be approximated by a stepwise constant function of the problem size. At the same time, application **MatrixMult**, which implements a straightforward algorithm of multiplication of two dense square matrices and uses inefficient memory reference patterns, displays quite a smooth dependence of speed on the problem size. For such applications, the speed of the processor cannot be accurately approximated by a stepwise constant function. It should be approximated by a continuous function of the problem size if we want the performance model to be accurate enough.

The other main motivation is that we target common heterogeneous networks rather than dedicated high performance computer systems. A computer in such a network is persistently performing some minor routine computations and communications just as an integrated node of the network. Examples of such routine applications include e-mail clients, browsers, text editors, audio applications, etc. As a result, the computer will experience constant and stochastic fluctuations in the workload. This changing transient load will cause a fluctuation in the speed of the computer in the sense that the execution time of the same task of the same size will vary for different runs at different times. The natural way to represent the inherent fluctuations in the speed is to use a speed band rather than a speed function. The width of the band characterizes the level of fluctuation in the performance due to changes in load over time. The shape of the band makes the dependence of the speed of the computer on the problem size less distinctive and sharp even in the case of carefully designed applications efficiently using the memory hierarchy. Therefore, even for such applications the speed of the processor can be realistically approximated by a continuous function of the problem size. Figure 2 shows experiments conducted with application **MatrixMultATLAS** on a set of computers whoses speci-

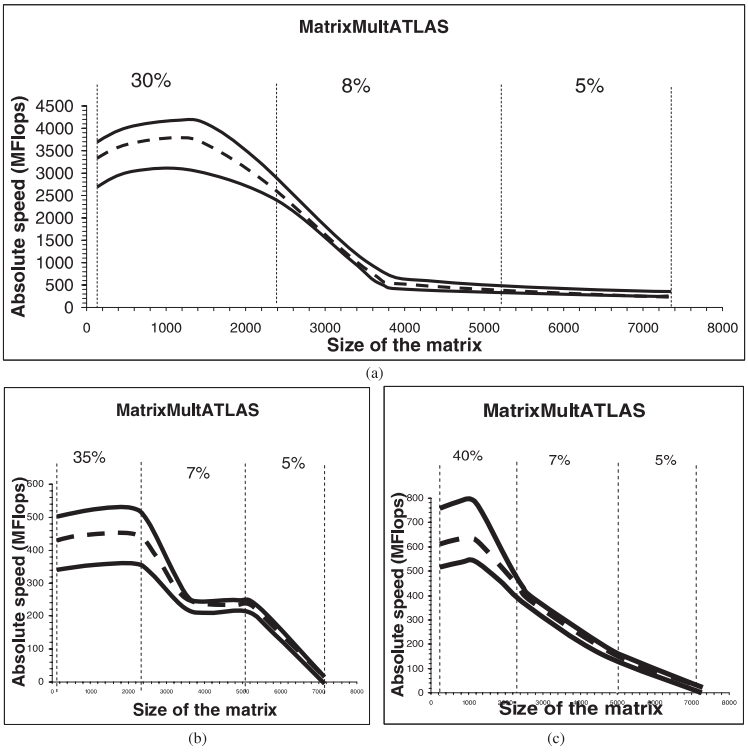

Fig. 2 Effect of workload fluctuations on the execution of application MatrixMultATLAS on computers shown in Table 1. The region between the bold curves composes the performance band. The width of the performance band is given as a percentage of the maximum speed of execution of the application. The dotted curve corresponds to the curve shown in Figure 1. (a) Performance band for Comp1; (b) performance band for Comp2; and (c) performance band for Comp4.

Table 1
Specifications of the four heterogeneous processors

Processor	Architecture	cpu MHz	Main memory (kBytes)	Cache (kBytes)
Comp 1	Linux 2.4.20-8 Intel(R) Pentium(R) 4	2793	513304	512
Comp 2	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	2048
Comp 3	Windows XP	3000	1030388	512
Comp 4	Linux 2.4.7-10 i686	730	254524	256

fications are shown in Table 1. The performance bands are obtained using the procedure given by Lastovetsky, Reddy and Higgins (2006). The application employs the level-3 BLAS routine **dgemm** (Dongarra et al. 1990) supplied by Automatically Tuned Linear Algebra Software (ATLAS; Whaley, Petitet and Dongarra 2000). ATLAS is a package that generates efficient code for basic linear algebra operations. The computers have varying specifications and varying levels of network integration and are representative of the range of computers typically used in networks of heterogeneous computers.

The problem of optimally scheduling divisible loads has been studied extensively and the theory is commonly referred to as divisible load theory (DLT). The main features of earlier works in DLT (Bharadwaj et al. 1996; Drozdowski and Wolniewicz 2003b) are that they assume distributed systems with a flat memory model and use a mathematical model where the speed of the processor is represented by a constant. Drozdowski and Wolniewicz (2003a) propose a new mathematical model that relaxes the above two assumptions. They study distributed systems, which have both the hierarchical memory model and a piecewise constant dependence of the speed of the processor on the problem size. However, the model they formulate is targeted mainly towards optimal distribution of arbitrary tasks for carefully designed applications on dedicated distributed multiprocessor computer systems, whereas our model is aimed towards optimal distribution of arbitrary tasks for any arbitrary application on common heterogeneous networks.

3 Distributing Independent Chunks

In this section, we study the problem of distributing independent chunks of computations over a unidimensional arrangement of heterogeneous processors. The form of presentation is very much inspired by that used in Beaumont et al. (2001a) to present the same problem but for heterogeneous processors whose performance is characterized by constants.

The problem is 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 of respective speeds $s_1(x)$, $s_2(x)$, ..., $s_p(x)$ so that the workload is best balanced? Here, the speed of the processor is understood as the number of computation chunks performed by the processor per one time unit. The speed depends on the number of chunks assigned to the processor and is represented by a continuous function s: $\mathbf{R}_+ \rightarrow \mathbf{R}_+$. How, then, do we distribute chunks to processors? The intuition says that the load x_i of P_i should be proportional to $s_i(x_i)$. Since the load (i.e. numbers of chunks) on each processor must be integers, we use the following two-step algorithm to solve the problem. Let n_i denote the number of chunks allocated to processor P_i. Then, the overall execution time obtained with allocation $(n_1, n_2, \ldots, n_p)$ is given by $\max_i \frac{n_i}{s_i(n_i)}$. The optimal solution minimizes the overall execution time.

Algorithm 1. Optimal distribution for n independent chunks over p processors of speeds $s_1(x)$, $s_2(x)$, ..., $s_p(x)$:

- **Step 1. Initialization**: We approximate the n_i so that $\frac{n_i}{s_i(n_i)} \approx \text{const}$ and $n - 2 \times p \le n_1 + n_2 + \ldots + n_p \le n$. Namely, we find n_i such that either $n_i = \lfloor x_i \rfloor$ or $n_i = \lfloor x_i \rfloor - 1$ for $1 \le i \le p$ where $\frac{x_1}{s_1(x_1)} = \frac{x_2}{s_2(x_2)} = \ldots = \frac{x_p}{s_p(x_p)}$.

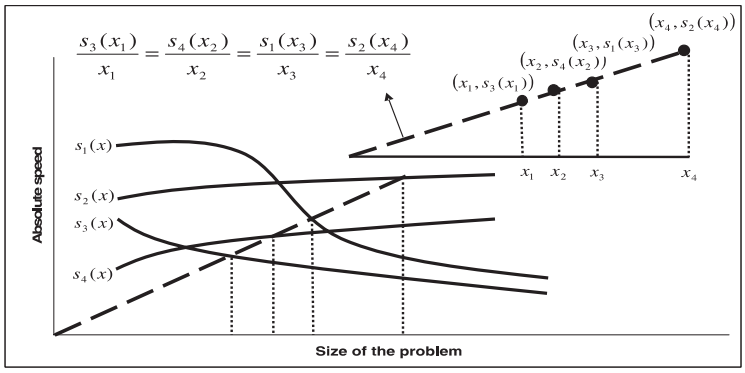

Fig. 3 "Ideal" optimal solution showing the geometric proportionality of the number of chunks to the speed of the processor.

- **Step 2. Refining**: We iteratively increment some n_i until $n_1 + n_2 + \ldots + n_p = n$.

Approximation of the n_i (Step 1) is not as easy as in the case of constant speeds s_i of the processors, when n_i can be approximated as $\left\lfloor \frac{s_i}{\sum_1^p s_i} \times n \right\rfloor$ (see Beaumont 2001a). The algorithm which we propose is based on the following observation: If $\frac{x_1}{s_1(x_1)} = \frac{x_2}{s_2(x_2)} = \ldots = \frac{x_p}{s_p(x_p)}$, then all the 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, being intersecting points of this line with the graphs of the speed functions of the processors. This is shown in Figure 3. Our algorithm is seeking for two straight lines passing through the origin of the coordinate system so that:

- The "ideal" optimal line (that is, the line, which intersects the speed graphs in points $(x_1, s_1(x_1))$, $(x_2, s_2(x_2))$, …, $(x_p, s_p(x_p))$ such that $\frac{x_1}{s_1(x_1)} = \frac{x_2}{s_2(x_2)} = \ldots = \frac{x_p}{s_p(x_p)}$ and $x_1 + x_2 + \ldots + x_p = n$) lies between the two lines.
- There is no more than one point with integer x coordinate on either of these graphs between the two lines.

Algorithm 1.1. Approximation of the n_i so that either $n_i = \lfloor x_i \rfloor$ or $n_i = \lfloor x_i \rfloor - 1$ for $1 \le i \le p$ where $\frac{x_1}{s_1(x_1)} = \frac{x_2}{s_2(x_2)} = \ldots = \frac{x_p}{s_p(x_p)}$ and $x_1 + x_2 + \ldots + x_p = n$:

1. The upper line U is drawn through the points $(0, 0)$ and $\left(\frac{n}{p}, \max_i\left\{s_i\left(\frac{n}{p}\right)\right\}\right)$, and the lower line L is drawn through the points $(0, 0)$ and $\left(\frac{n}{p}, \min_i\left\{s_i\left(\frac{n}{p}\right)\right\}\right)$, as shown in Figure 4.
2. Let $x_i^{(U)}$ and $x_i^{(L)}$ be the coordinates of the intersection points of lines U and L with the function $s_i(x)$

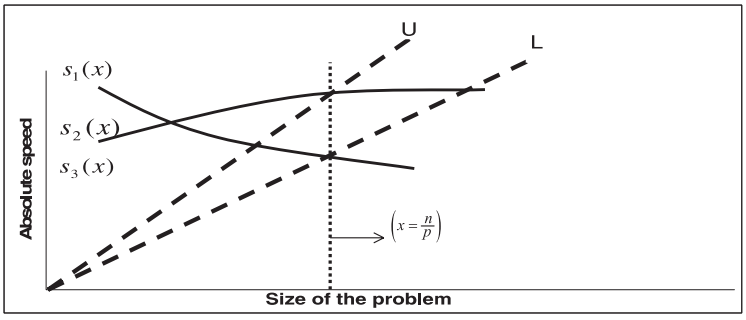

Fig. 4 Selection of the initial two lines L and U; n is the size of the problem and p is the number of processors.

($1 \le i \le p$). **If** there exists $i \in \{1, \ldots, p\}$ such that $x_i^{(L)} - x_i^{(U)} \ge 1$ **then** go to step 3 **else** go to step 5.

3. Bisect the angle between lines U and L by the line M. Calculate coordinates $x_i^{(M)}$ of the intersection points of the line M with the function $s_i(x)$ for $1 \le i \le p$.
4. **If** $\sum_1^p x_i^{(M)} \le n$ **then** $U = M$ **else** $L = M$. Repeat step 2.
5. Approximate the n_i so that $n_i = \lfloor x_i^{(U)} \rfloor$ for $1 \le i \le p$.

Proposition 1. *Let the function $s_i(x)$ $(1 \le i \le p)$ be continuous and there exist point $x' \ge 0$ such that:*

- *On the interval $[0, x']$, the function is*
  - *monotonically increasing,*
  - *concave, and*
  - *any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point.*
- *On the interval $[x', \infty)$, the function is monotonically decreasing.*

Then Algorithm 1.1 finds the n_i such that either $n_i = \lfloor x_i \rfloor$ or $n_i = \lfloor x_i \rfloor - 1$ for $1 \le i \le p$ where $\frac{x_1}{s_1(x_1)} = \frac{x_2}{s_2(x_2)} = \ldots = \frac{x_p}{s_p(x_p)}$ and $x_1 + x_2 + \ldots + x_p = n$.

Proof. First we formulate a few obvious properties of the functions $s_i(x)$.

Lemma 1.1. *The functions $s_i(x)$ are bounded.*

Lemma 1.2. *Any straight line coming through the origin of the coordinate system intersects the graph of the function $s_i(x)$ in no more than one point.*

Lemma 1.3. *Let $x_i^{(M_k)}$ be the coordinate of the intersection point of $s_i(x)$ and a straight line M_k coming through the origin of the coordinate system ($k \in \{1, 2\}$). Then $x_i^{(M_1)} \ge x_i^{(M_2)}$ if and only if $\angle(M_1, X) \le \angle(M_2, X)$, where $\angle(M_k, X)$ denotes the angle between the line M_k and the x-axis.*

Since $s_i(x)$ are continuous and bounded, the initial lines U and L always exist. Since there is no more than one point of intersection of the line L with each of $s_i(x)$, L will make a positive angle with the x-axis. Thus, both U and L will intersect each $s_i(x)$ exactly in one point. Let $x_i^{(U)}$ and $x_i^{(L)}$ be the coordinates of the intersection points of U and L with $s_i(x)$ ($1 \le i \le p$) respectively. Then by design, $\sum_{i=1}^p x_i^{(U)} \le n \le \sum_{i=1}^p x_i^{(L)}$. This invariant will hold after each iteration of the algorithm. Indeed, if line M bisects the angle between lines U and L, then $\angle(L, X) \le \angle(M, X) \le \angle(U, X)$. Hence, $\sum_{i=1}^p x_i^{(U)} \le \sum_{i=1}^p x_i^{(M)} \le \sum_{i=1}^p x_i^{(L)}$. If $\sum_{i=1}^p x_i^{(M)} \le n$, then $\sum_{i=1}^p x_i^{(U)} \le \sum_{i=1}^p x_i^{(M)} \le n \le \sum_{i=1}^p x_i^{(L)}$ and after step 4 of the algorithm $\sum_{i=1}^p x_i^{(U)} \le n \le \sum_{i=1}^p x_i^{(L)}$. If $\sum_{i=1}^p x_i^{(M)} \ge n$, then $\sum_{i=1}^p x_i^{(U)} \le n \le \sum_{i=1}^p x_i^{(M)} \le \sum_{i=1}^p x_i^{(L)}$ and after step 4 of the algorithm $\sum_{i=1}^p x_i^{(U)} \le n \le \sum_{i=1}^p x_i^{(L)}$. Thus, after each iteration of the algorithm, the "ideal" optimal line O such that $\sum_{i=1}^p x_i^{(O)} = n$ will be lying between lines U and L. When the algorithm reaches step 5, we have $x_i^{(L)} - x_i^{(U)} < 1$ for all $1 \le i \le p$, which means that the interval $\left[x_i^{(L)}, x_i^{(U)}\right]$ contains at most one integer value. Therefore, either $n_i = \lfloor x_i^{(U)} \rfloor = \lfloor x_i^{(O)} \rfloor$ or $n_i = \lfloor x_i^{(U)} \rfloor = \lfloor x_i^{(O)} \rfloor - 1$.

Algorithm 1.2. Iterative incrementing of some n_i until $n_1 + n_2 + \ldots + n_p = n$:

1. **If** $n_1 + n_2 + \ldots + n_p < n$ **then** go to step 2 **else** stop the algorithm.
2. Find $k \in \{1, \ldots, p\}$ such that $\frac{n_k + 1}{s_k(n_k + 1)} = \min_{i=1}^p \left\{ \frac{n_i + 1}{s_i(n_i + 1)} \right\}$.
3. $n_k = n_k + 1$. Repeat step 1.

Note. It is worth stressing that Algorithm 1.2 cannot be used to search for the optimal solution beginning from an arbitrary approximation n_i satisfying inequality $n_1 + n_2 + \ldots + n_p < n$, but only from the approximation found by Algorithm 1.1.

Proposition 2. *Let the functions $s_i(x)$ $(1 \le i \le p)$ satisfy the conditions of Proposition 1. Let $(n_1, n_2, \ldots, n_p)$ be the approximation found by Algorithm 1.1. Then Algorithm 1.2 gives the optimal allocation.*

Proof. The execution time obtained with allocation $(n_1, n_2, \ldots, n_p)$ is given by $\max_i \frac{n_i}{s_i(n_i)}$. The geometrical interpretation of this formula is as follows. Let M_i be the straight line connecting the points (0,0) and $(n_i, s_i(n_i))$. Then $\frac{n_i}{s_i(n_i)} = \cot\angle(M_i, X)$. Therefore, minimization of $\max_i \frac{n_i}{s_i(n_i)}$ is equivalent to maximization of $\min_i \angle(M_i, X)$. Let $\{S_1, S_2, \ldots\}$ be the set of all straight lines such that:

- S_k connects (0,0) and $(m, s_i(m))$ for some $i \in \{1, \ldots, p\}$ and some integer m,
- S_k lies below M_i for any $i \in \{1, \ldots, p\}$.

Let $\{S_1, S_2, \ldots\}$ be ordered in the decreasing order of $\angle(S_k, X)$. The execution time of the allocation $(n_1, n_2, \ldots, n_p)$ is represented by line M_k such that $\angle(M_k, X) = \min_i \angle(M_i, X)$. Incrementing of any n_i means moving one more line from the set $\{S_1, S_2, \ldots\}$ into the set of lines representing the allocation. At each step of the incrementing, Algorithm 1.2 moves the line making the largest angle with the x-axis. This means that after each increment the algorithm gives the optimal allocation $(n_1, n_2, \ldots, n_p)$ under the assumption that the total number of chunks, which should be allocated, is equal to $n_1 + n_2 + \ldots + n_p$ (any other increment gives a smaller angle and, hence, longer execution time). Therefore, after the last increment the algorithm gives the optimal allocation $(n_1, n_2, \ldots, n_p)$ under the assumption that $n_1 + n_2 + \ldots + n_p = n$.

3.1 Complexity

In this section, we estimate the complexity of Algorithm 1. We start with the complexity of Algorithm 1.1.

Definition. The heterogeneity of the set of p physical processors $P_1, P_2, \ldots, P_p$ of respective speeds $s_1(x), s_2(x), \ldots, s_p(x)$ is *bounded* if and only if there exists a constant c such that $\max_{x \in R_+} \frac{s_{\max}(x)}{s_{\min}(x)} \le c$ where $s_{\max}(x) = \max_i s_i(x)$ and $s_{\min}(x) = \min_i s_i(x)$.

Proposition 3. *Let the functions $s_i(x)$ $(1 \le i \le p)$ satisfy the conditions of Proposition 1 and the heterogeneity of processors $P_1, P_2, \ldots, P_p$ be bounded. Then, the complexity of Algorithm 1.1 is $O(p \times \log_2 n)$.*

Proof. First, we estimate the complexity of one iteration of Algorithm 1.1. At each iteration we need to find the points of intersection of p graphs $y = s_1(x)$, $y = s_2(x)$, ..., $y = s_p(x)$ and a straight line $y = a \times x$. In other words, at each iteration we need to solve p equations of the form $a \times x = s_i(x)$. As we need the same constant number of operations to solve each equation, the complexity of this part of one iteration will be $O(p)$. The test for stopping (step 2 of the algorithm) also takes a constant number of operations per function $s_i(x)$ making the complexity of this part of one iteration $O(p)$. Therefore, overall the complexity of one iteration of Algorithm 1.1 will be $O(p)$.

Next, we estimate the number of iterations of this algorithm. To do it, we use the following lemma that states one important property of the initial lines U and L obtained at the step 1 of Algorithm 1.1.

Lemma 3.1. *Let the functions $s_i(x)$ $(1 \le i \le p)$ satisfy the conditions of Proposition 1 and the heterogeneity of processors $P_1, P_2, \ldots, P_p$ be bounded. Let O be the point $(0, 0)$, A_i be the point of intersection of the initial line U and $s_i(x)$, and B_i be the point of intersection of the initial line L and $s_i(x)$. Then, there exist constants c_1 and c_2 such that $c_1 \le \frac{OB_i}{OA_i} \le c_2$ for any $i \in \{1, 2, \ldots, p\}$.*

Proof of Lemma 3.1. The full proof of Lemma 3.1 is technical and very lengthy. Here, we give a relatively compact proof of the lemma under the additional assumption that the functions $s_i(x)$ $(1 \le i \le p)$ are monotonically decreasing. First, we prove that there exist constants c_1 and c_2 such that $c_1 \le \frac{OB}{OA} \le c_2$ where A is the point of intersection of the initial line U and $s_{\max}(x) = \max_i s_i(x)$, and B is the point of intersection of the initial line L and $s_{\max}(x)$ (see Figure 5). Since the heterogeneity of the processors $P_1, P_2, \ldots, P_p$ is bounded, there exists a constant c such that $\max_{x \in R_+} \frac{s_{\max}(x)}{s_{\min}(x)} \le c$. In particular, this means that $\frac{BD}{FD} \le c$ and $\frac{AC}{EC} \le c$. Let us prove that $\frac{OB}{OA} \le c$. We have $OB = \sqrt{OD^2 + BD^2}$. Since $\frac{OD}{OC} = \frac{BD}{EC}$, we have $OD = \frac{BD}{EC} \times OC$. Since $s_{\min}(x)$ monotonically decreases on the interval $\left[\frac{n}{p}, \infty\right]$, $FD \le EC$ and, hence, $\frac{BD}{EC} \le \frac{BD}{FD} \le c$. Thus, $OD \le c \times OC$ and $BD \le c \times EC$. Therefore $\sqrt{OD^2 + BD^2} \le \sqrt{c^2 \times OC^2 + c^2 \times EC^2} = c \times \sqrt{OC^2 + EC^2} = c \times OE$, and hence $\frac{OB}{OE} \le c$. Since $OA \ge OE$, then $\frac{OB}{OA} \le \frac{OB}{OE} \le c$. Next let us prove that

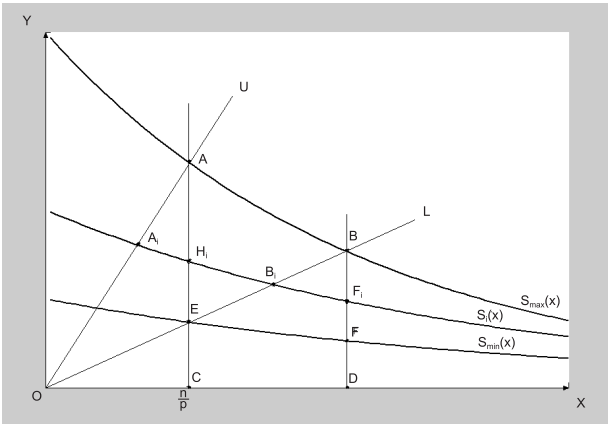

Fig. 5 The picture after the initial step of Algorithm 1.1. Here, $s_{max}(x) = \max_i s_i(x)$ and $s_{min}(x) = \min_i s_i(x)$.

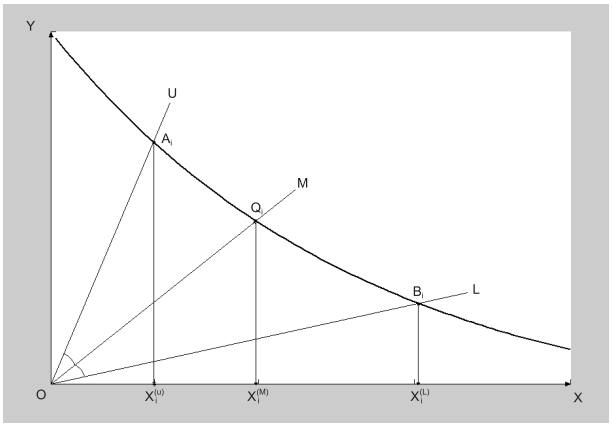

Fig. 6 Bisection of the angle $\angle A_i O B_i$ at the very first iteration into two equal angles. The segment A_iB_i of the graph of the function $s_i(x)$ will be divided in the proportion $\frac{Q_iB_i}{A_iQ_i} \approx \frac{OB_i}{OA_i}$.

$\frac{OB}{OA} \geq \frac{1}{c}$. We have $OB \geq OE$ and $AC \leq c \times EC$. Therefore,

$$\frac{OB}{OA} \geq \frac{OE}{OA} = \frac{\sqrt{OC^2 + EC^2}}{\sqrt{OC^2 + AC^2}} = \frac{OC \times \sqrt{1 + \left(\frac{EC}{OC}\right)^2}}{OC \times \sqrt{1 + c^2 \times \left(\frac{EC}{OC}\right)^2}} =$$

$$\frac{1}{c} \times \sqrt{\frac{1 + \left(\frac{EC}{OC}\right)^2}{\frac{1}{c^2} + \left(\frac{EC}{OC}\right)^2}}.$$ Since $c \geq 1$, then $\sqrt{\frac{1 + \left(\frac{EC}{OC}\right)^2}{\frac{1}{c^2} + \left(\frac{EC}{OC}\right)^2}} \geq 1$ and hence $\frac{OB}{OA} \geq \frac{1}{c}$.

Now we are ready to prove Lemma 3.1. We have $\frac{OB_i}{OA_i} \leq \frac{OB}{OA_i} = \frac{1}{OA_i} \times OB$. Since $s_i(x)$ is monotonically decreasing, $\frac{OA}{OA_i} \leq \frac{AC}{CH_i}$. Since the heterogeneity of the processors is bounded by the constant c, $\frac{AC}{CH_i} \leq c$. Hence, $\frac{1}{OA_i} \leq \frac{c}{OA}$. Therefore, $\frac{OB_i}{OA_i} \leq \frac{c}{OA} \times OB = c \times \frac{OB}{OA} \leq c^2$. Next, we have $\frac{OB_i}{OA_i} \geq \frac{OB_i}{OA}$. Since $s_i(x)$ is monotonically decreasing, $\frac{BD}{F_iD} \geq \frac{OB}{OB_i}$. Since the heterogeneity of the processors is bounded by the constant c, $\frac{BD}{F_iD} \leq c$. Therefore, $OB_i \geq \frac{OB}{c}$. Thus, $\frac{OB_i}{OA_i} \geq \frac{OB_i}{OA} \geq \frac{OB}{c \times OA} \geq \frac{1}{c^2}$.

This proves Lemma 3.1.

Bisection of the angle $\angle A_i O B_i$ at the very first iteration will divide the segment A_iB_i of the graph of the function $s_i(x)$ in the proportion $\frac{Q_iB_i}{A_iQ_i} \approx \frac{OB_i}{OA_i}$ (see Figure 6). Correspondingly, $\frac{x_i^{(L)} - x_i^{(M)}}{x_i^{(M)} - x_i^{(U)}} \approx \frac{OB_i}{OA_i}$. Since $(b - a)$ approximates the number of integers in the interval $[a, b]$, $\Delta_i = \min\left\{\frac{x_i^{(L)} - x_i^{(M)}}{x_i^{(L)} - x_i^{(U)}}, \frac{x_i^{(M)} - x_i^{(U)}}{x_i^{(L)} - x_i^{(U)}}\right\}$ will approximate the lower bound on the fraction of the set $\left\{\left\lceil x_i^{(U)} \right\rceil, \left\lceil x_i^{(U)} \right\rceil + 1, \ldots, \left\lfloor x_i^{(L)} \right\rfloor\right\}$ of possible numbers of chunks to be allocated to the processor P_i, which is excluded from consideration after this bisection. Since $c_1 \leq \frac{OB_i}{OA_i} \leq c_2$, then $\Delta_i \geq \frac{c_1}{c_2 + 1} = \Delta$. Indeed, let $q_i = x_i^{(L)} - x_i^{(M)}$ and $r_i = x_i^{(M)} - x_i^{(U)}$. We have $c_1 \leq \frac{q_i}{r_i} \leq c_2$. Therefore, $c_1 \times r_i \leq q_i \leq c_2 \times r_i$ and $(c_1 + 1) \times r_i \leq q_i + r_i \leq (c_2 + 1) \times r_i$. Hence, $\frac{q_i}{q_i + r_i} \geq \frac{c_1 \times r_i}{(c_1 + 1) \times r_i} = \frac{c_1}{c_2 + 1}$.

$\Delta_i \geq \frac{c_1}{c_2 + 1} = \Delta$ means that after this bisection, at least $\Delta \times 100\%$ of the possible solutions will be excluded from consideration for each processor P_i. The difference in length between OB_i and OA_i will be getting smaller and smaller with each next iteration. Therefore, no less than $\Delta \times 100\%$ of the possible solutions will be excluded from consideration after each iteration of Algorithm 1.1. The number of possible solutions in the initial set for each processor P_i is obviously less than n. The constant Δ does not depend on p or n (actually, this parameter just characterizes the heterogeneity of the set of processors). Therefore, the number of iterations k needed to arrive at the final solution can be found from the equation $(1 - \Delta)^k \times n = 1$, and we have $k = \frac{1}{\log_2\left(\frac{1}{1-\Delta}\right)} \times \log_2 n$. Thus, the overall complexity of Algorithm 1.1 will be $O(p \times \log_2 n)$. Proposition 3 is proved.

Note. The low complexity of Algorithm 1.1 is mainly due to the bounded heterogeneity of the processors. This very property guarantees that each bisection will reduce the space of possible solutions by a fraction lower bounded by some finite positive number independent on n. The assumption of bounded heterogeneity will be inaccurate if the speed of some processors becomes too slow for large n, effectively approaching 0. One approach to this problem is to use a relaxed functional model where the speed of the processor is represented by a continuous function until some given size of the problem and by zero for all sizes greater than this one. Data partitioning algorithms with that model are presented by Lastovetsky and Reddy (2005). The other approach is to use algorithms not sensitive to the shape of performance functions such as the algorithm of complexity $O(p^2 \times \log_2 n)$ presented in Lastovetsky and Reddy (2004).

Proposition 4. *Let the functions $s_i(x)$ $(1 \leq i \leq p)$ satisfy the conditions of Proposition 1 and the heterogeneity of processors $P_1, P_2, \ldots, P_p$ be bounded. Then, the complexity of Algorithm 1 is $O(p \times \log_2 n)$.*

Proof. If $(n_1, n_2, \ldots, n_p)$ is the approximation found by Algorithm 1.1, then $n - 2 \times p \leq n_1 + n_2 + \ldots + n_p \leq n$ and Algorithm 1.2 gives the optimal allocation in at most $2 \times p$ steps of increment, so that the complexity of Algorithm 1.2 is $O(p^2)$. This complexity is given by a naïve implementation of Algorithm 1.2. The complexity of this algorithm can be reduced down to $O(p \times \log_2 p)$ by using ad hoc data structures (Beaumont et al. 2001a). Thus, overall the complexity of Algorithm 1 will be $O(p \times \log_2 p + p \times \log_2 n) = O(p \times \log_2 (p \times n))$. Since $p < n$, then $\log_2(p \times n) < \log_2(n \times n) = \log_2(n^2) = 2 \times \log_2 n$. Thus, the overall complexity of Algorithm 1 will be bounded by $O(2 \times p \times \log_2 n) = O(p \times \log_2 n)$.

3.2 Application of the Partitioning Algorithm

In this section, we apply the set partitioning algorithm to a matrix multiplication application using horizontal striped partitioning of matrices on a network of p heterogeneous computers. Our main aim is not to show how matrices can be efficiently multiplied but to explain in simple terms how the set partitioning algorithm using the functional model can be applied to optimally schedule computational tasks on networks of heterogeneous computers.

The matrix multiplication application shown in Figure 7(a) multiplies matrix A and matrix B, i.e. implementing matrix operation $C = A \times B$, where A, B, and C are dense square $n \times n$ matrices. The matrices A and C are horizontally sliced such that the number of elements in a slice is proportional to the speed of the processor owning the slice. All the processors contain all the elements of matrix B. We assume one process per processor configuration.

For this application, the absolute speed of the processor is obtained based on multiplication of two dense matrices of size $n_1 \times n$ and $n \times n$ respectively to obtain a resultant matrix of size $n_1 \times n$ as shown in Figure 7(b). The size of the problem is represented by two parameters, n_1 and n. The total number of matrix elements to store and process is $(2 \times n_1 \times n + n \times n)$. 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 $n_1 \times n \times n$. Therefore, the speed of the processor exposed by the application when solving the problem of size (n_1, n) can be calculated as $n_1 \times n \times n$ divided by the execution time of the application. This gives us a function, f: $\mathbf{N}^2 \rightarrow \mathbf{R}_+$, mapping problem sizes to speeds of the processor. The functional performance model 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 $\mathbf{N}^2$).

A practical procedure to build the functional performance model of a processor is given by Lastovetsky, Reddy and Higgins (2006). In brief, the algorithm presented by Lastovetsky, Reddy and Higgins (2006) exploits historic records of workload fluctuations of the processor in order to minimize the number of experimental points needed to accurately approximate the performance band by a piecewise linear function fitting within the band.

The speed function is geometrically represented by a surface as shown in Figures 8(a) and 8(b) for two processors X1 and X5 used in experiments and whose speci-

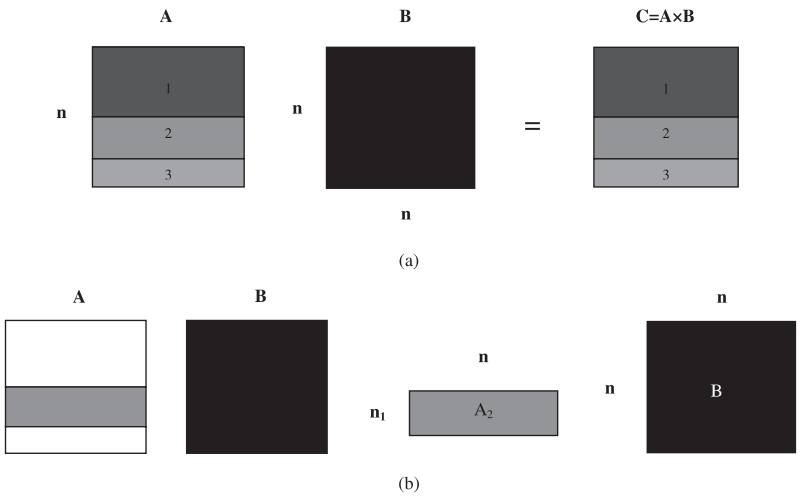

Fig. 7 (a) Matrix operation C = A × B with matrices A, B, and C. Matrices A and C are horizontally sliced such that the number of elements in the slice is proportional to the speed of the processor. (b) Serial matrix multiplication A_2 × B of dense matrix A2 of size n_1 x n and dense matrix B of size n x n to estimate the absolute speed of processor 2.

Table 2
Specifications of the eight heterogeneous processors

Processor	Architecture	CPU MHz	Main memory (kBytes)	Free main memory (kBytes)	Cache (kBytes)
X1	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	1977	1030508	938976	512
X2	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	1977	1030508	972924	512
X3	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	1977	1030508	967176	512
X4	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	1977	1030508	967312	512
X5	SunOS 5.9 sun4u sparc SUNW,Ultra-5_10	440	524288	134400	2048
X6	SunOS 5.9 sun4u sparc SUNW,Ultra-5_10	440	524288	409600	2048
X7	SunOS 5.9 sun4u sparc SUNW,Ultra-5_10	440	524288	418816	2048
X8	SunOS 5.9 sun4u sparc SUNW,Ultra-5_10	440	524288	395264	2048

fications are shown in Table 2. Figure 8(c) shows the geometrical representation of the relative speed of these two processors calculated as the ratio of their absolute speeds. One can see that the relative speed varies significantly depending on the value of variables n_1 and n.

When partitioning a square $n \times n$ matrix, we use the fact that the width of partitions is fixed and equal to n. Firstly, we section the surfaces representing the absolute speeds of the processors by the plane parallel to the axis representing the parameter n_1 and parallel to the axis rep-

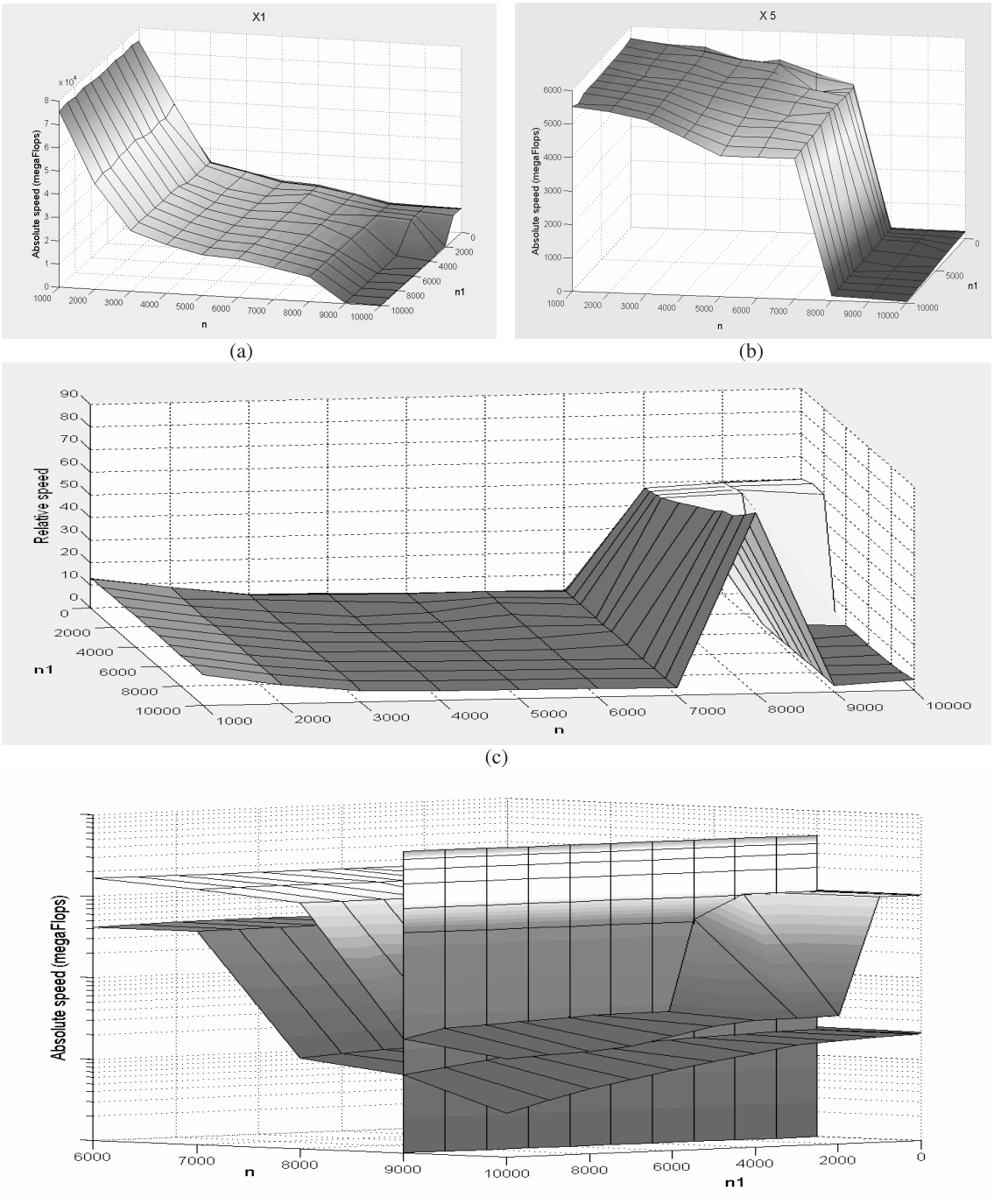

Fig. 8 (a) The absolute speed of the processor X1 as a function of n_1 and n. (b) The absolute speed of the processor X5 as a function of n_1 and n. (c) The relative speed of the two processors calculated as the ratio of their absolute speeds. (d) Two surfaces representing the absolute speeds of the two processors are sectioned by the plane n = 9000 parallel to the axis of parameter n1 and parallel to the axis of absolute speed of the processor. Curves on this plane represent the absolute speeds of the processors against variable n_1 given parameter n is fixed.

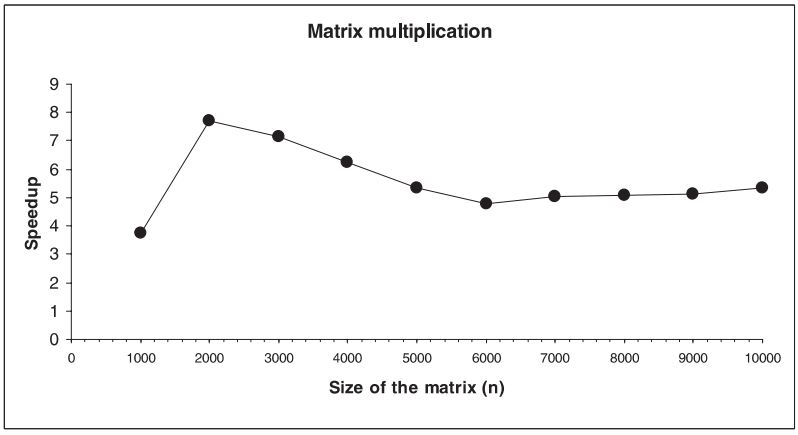

Fig. 9 Speedup of matrix multiplication application using the network of heterogeneous computers shown in Table 2 over the matrix multiplication application using the most powerful computer X1.

resenting the absolute speed of the processor and having an intercept of n on the axis representing the parameter n. This is illustrated in Figure 8(d) for two surfaces representing the absolute speeds of the processors X1 and X5. In this way we obtain a set of p curves on this plane that represent the absolute speeds of the processors against variable n_1 given parameter n is fixed. Then we apply the set partitioning algorithm to this set of p curves to obtain optimal distribution of slices in matrices A and C.

4 Experimental results

A small heterogeneous local network of 8 different Solaris and Linux workstations shown in Table 2 is used in the experiments. The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers. The amount of memory, which is the difference between the main memory and free main memory shown in the tables, is used by the operating system processes and a few other user application processes that perform routine computations and communications such as e-mail clients, browsers, text editors, audio applications etc. These processes use a constant percentage of CPU.

Figure 9 shows the speedup of using the network of heterogeneous computers shown in Table 2 over the most powerful computer X1. The speedup calculated is the ratio of execution time of the serial matrix multiplication application using X1 over the execution time of the parallel matrix multiplication application, described in Section 3.2, using the functional model on the network of heterogeneous computers.

Figure 10 shows the speedup of the matrix multiplication application, described in Section 3.2, executed on this network using the functional model over the matrix multiplication using the single number model. In the figures, for each problem size, the speedup calculated is the ratio of the execution time of the application using the single number model over the execution time of the application using the functional model.

We consider three cases for comparing the functional model with the single number model in the range (1000, 10 000) of matrix sizes. For the first case the single number model uses speed obtained by multiplying matrices of sizes 500×1000 and 1000×1000. This case covers the range of small sized matrices. The single number model for the second case uses speed based on multiplication of matrices of sizes 2500×5000 and 5000×5000. This case covers the range of medium sized matrices. For the third case, speed obtained by multiplying matrices of sizes 4000×8000 and 8000×8000 is used. This case covers the large sized matrices. The ratios of speeds of the most powerful computer X1 and the least powerful computer X5 in these cases are 13.5, 3.75, and 57.0 respectively.

It can be seen from the figure that the single number model in the first case performs poorly in the range of medium sized to large sized matrices. In the second case the single number model does not perform well for small sized and large sized matrices. In the third case the single number model does not perform well in the range of small sized and medium sized matrices and for large sized matrices with problem size greater than (8000,8000). Therefore the functional model performs better than the single number model for a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds vary with the problem size. It can be concluded that our set partitioning algorithm using the functional model performs better for all sizes of matrices.

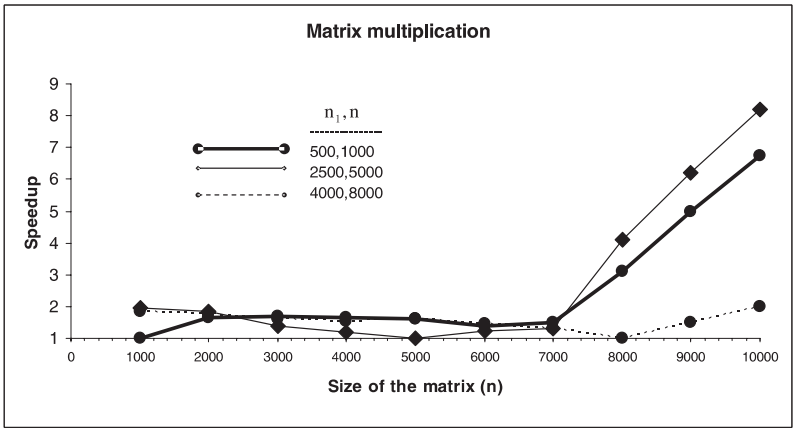

Fig. 10 The speedup of the parallel matrix multiplication application using the functional model over single number model on the network of heterogeneous computers shown in Table 2. The speeds used in the single number model in the three curves for comparison are obtained using serial matrix multiplication of matrices of problem sizes (n_1, n) = (500, 1000), (2500, 5000), and (4000, 8000) respectively.

5 Conclusion

In this paper, we addressed the problem of optimal distribution of computational tasks on a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds vary with the problem size. We have proposed and analyzed the functional performance model of heterogeneous processors. This model integrates many essential features of a network of heterogeneous computers having a major impact on its performance such as the processor heterogeneity, the heterogeneity of memory structure, and the effects of paging. Under this model, the speed of each processor is represented by a continuous function of the size of the problem whereas traditional models use single numbers to represent the speeds of the processors. We have formulated a problem of partitioning of an n-element set over p heterogeneous processors using this model and designed an algorithm of the complexity $O(p \times \log_2 n)$ solving the problem. Some early results on the functional model and data partitioning with this model were presented by Lastovetsky and Reddy (2004).

Acknowledgment

The work was supported in part by the Science Foundation Ireland.

Author Biographies

Alexey Lastovetsky received his Ph.D. degree from the Moscow Aviation Institute in 1986, and a Doctor of Science degree from the Russian Academy of Sciences in 1997. His main research interests include algorithms, models and programming tools for high performance heterogeneous computing. He is the author of C[], a parallel language for vector and superscalar processors, and mpC, the first parallel programming language for heterogeneous networks of computers. He designed HeteroMPI, an extension of MPI for heterogeneous parallel computing, and SmartNetSolve, an extension of NetSolve aimed at higher performance of scientific computing on global networks. He has also made contributions into heterogeneous data distribution algorithms and modeling the performance of processors in heterogeneous environments. He has published over 70 technical papers in refereed journals, edited books and international conferences. He authored the monograph "Parallel computing on heterogeneous networks" published by Wiley in 2003. He is currently a senior lecturer in the School of Computer Science and Informatics at University College Dublin, National University of Ireland. At UCD, he also created and leads the Heterogeneous Computing Laboratory. He is an editor of the research journals "Parallel Computing" (Elsevier) and "Programming and Computer Software" (Springer).

Ravi Reddy received his Ph.D. degree from the Computer Science Department, University College Dublin, National University of Ireland in 2005. His main research interests are design of algorithms and tools for parallel and distributed computing systems.

References

Beaumont, O., Boudet, V., Petitet, A., Rastello, F. and Robert, Y. (2001a). A proposal for a heterogeneous cluster

ScaLAPACK (dense linear solvers), *IEEE Transactions on Computers*, **50**: 1052–1070.

Beaumont, O., Boudet, V., Rastello, F. and Robert, Y. (2001b). Matrix multiplication on heterogeneous platforms, *IEEE Transactions on Parallel and Distributed Systems*, **12**: 1033–1051.

Bharadwaj, V., Ghose, D., Mani, V. and Robertazzi, T. G. (1996). *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE Computer Society Press and John Wiley & Sons, Los Alamitos, CA.

Cierniak, M., Li, W. and Zaki, M. J. (1997). Compile-time scheduling algorithms for heterogeneous network of workstations, *Computer Journal, Special Issue on Automatic Loop Parallelization*, **40**(6): 356–372.

Crandall, P. and Quinn, M. (1993). Block data decomposition for data-parallel programming on a heterogeneous workstation network, in *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC '93)*, 20–23 July, Spokane, WA, USA, pp.42–49, IEEE Computer Society.

Crandall, P. and Quinn, M. (1995). Problem decomposition for non-uniformity and processor heterogeneity, *Journal of the Brazilian Computer Society*, **2**: 13–23.

Dongarra, J., Croz, J. D., Duff, I. S. and Hammarling, S. (1990). A set of level-3 basic linear algebra subprograms, *ACM Transactions on Mathematical Software*, **16**: 1–17.

Drozdowski, M. and Wolniewicz, P. (2003a). Out-of-core divisible load processing, *IEEE Transactions on Parallel and Distributed Systems*, **14**: 1048–1056.

Drozdowski, M. and Wolniewicz, P. (2003b). Divisible load scheduling in systems with limited memory, *Cluster Computing*, **6**: 19–29.

Iverson, M. and Ozguner, F. (1998). Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment, in *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, Orlando, FL, IEEE Computer Society Press, pp.70–78, March.

Kalinov, A. and Lastovetsky, A. (2001). Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers, *Journal of Parallel and Distributed Computing*, **61**: 520–535.

Kumar, V., Grama, A., Gupta, A. and Karypis, G. (1994). *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings, Addison-Wesley, Reading, MA.

Lastovetsky, A. and Reddy, R. (2004). Data partitioning with a realistic performance model of networks of heterogeneous computers, in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 26–30 April 2004, Santa Fe, New Mexico, USA, CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Lastovetsky, A. and Reddy, R. (2005). Data partitioning for multiprocessors with memory heterogeneity and memory constraints, *Scientific Programming*, **13**: 93–112, IOS Press.

Lastovetsky, A., Reddy, R. and Higgins, R. (2006). Building the functional performance model of a processor, in *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC'06)*, 23–27 April 2006, Dijon, France, pp.746–753, ACM Press.

Lastovetsky, A. and Twamley, J. (2005). Towards a realistic performance model for networks of heterogeneous computers, in *High Performance Computational Science and Engineering: Proceedings of IFIP TC5 Workshop*, World Computer Congress, 22–27 August 2004, Toulouse, France, eds M. K. Ng, A. Doncescu, L.T. Yang, T. Leng, pp.39–58, Springer.

Maheswaran, M. and Siegel, H. J. (1998). A dynamic matching and scheduling algorithm for heterogeneous computing systems, in *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW 1998)*, Orlando, FL, pp.57–69, IEEE Computer Society Press, March.

Petitet, A. and Dongarra, J. (1999). Algorithmic redistribution methods for block-cyclic decompositions, *IEEE Transactions on Parallel and Distributed Systems*, **10:** 1201–1216.

Tan, M., Siegel, H. J., Antonio, J. K. and Li, Y. A. (1997). Minimizing the application execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system, *IEEE Transactions on Parallel and Distributed Systems*, **8**: 857–871.

Yan, Y., Zhang, X., and Song, Y. (1996). An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW, *Journal of Parallel and Distributed Computing*, **38**: 63–80.

Whaley, R. C., Petitet, A. and Dongarra, J. (2000). Automated empirical optimizations of software and the atlas project, Technical report, Department of Computer Sciences, University of Tennessee, Knoxville, March.