Parallel Processing Letters

Dynamic Load Balancing of Parallel Computational Iterative Routines on Highly Heterogeneous HPC Platforms

David Clarke, Alexey Lastovetsky, Vladimir Rychkov
School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland
David.Clarke.1@ucdconnect.ie, {Alexey.Lastovetsky, vladimir.rychkov}@ucd.ie

Received (27 December 2010)
Revised (29 March 2011)
Communicated by (Guest Editors)

ABSTRACT

Traditional load balancing algorithms for data-intensive iterative routines can successfully load balance relatively small problems. We demonstrate that they may fail on highly heterogeneous HPC platforms. Traditional algorithms use models of processors' performance which are too simplistic to reflect the many aspects of heterogeneity. This paper presents a new class of dynamic load balancing algorithms based on the advanced functional performance models. The models are functions of problem size and are built adaptively by measuring the execution time of each iteration. Two particular load balancing algorithms of this class are presented in the paper. The low execution cost of distribution of computations between heterogeneous processors in these algorithms make them suitable for employment in self-adaptable applications. Experimental results demonstrate that our algorithms can successfully balance data-intensive iterative routines on parallel platforms with high heterogeneity for the whole range of problem sizes.

Keywords: dynamic load balancing; iterative algorithms; highly heterogeneous HPC platforms; functional performance models of processors; data partitioning.

1. Introduction

In this paper we study load balancing of data-intensive parallel iterative routines on heterogeneous platforms. These routines are characterised by a high data-to-computation ratio within a single iteration. The computation load of a single iteration can be broken into any number of equal independent computational units [1]. Each iteration is dependent on the previous one. The generalised scheme of these routines can be summarised as follows: (i) data is partitioned over the processors, (ii) at each iteration some independent calculations are carried out in parallel by the processors, and (iii) some data synchronisation takes place. Typically computational workload is directly proportional to the size of data. Examples of scientific compu-

tational routines include Jacobi method, mesh-based solvers, signal processing and image processing.

High performance of iterative routines on heterogeneous platforms can be achieved when all processors complete their work within the same time. This is achieved by partitioning the computational workload and, hence, data unevenly across all processors. Workload should be distributed with respect to the processor speed, memory hierarchy and communication network [2]. Load balancing of parallel applications on heterogeneous platforms has been widely studied for different types of applications and in various aspects of heterogeneity. Many load balancing algorithms are not appropriate to either the applications or platforms considered in this paper. Applicable algorithms use models of processors' performance which are too simplistic and may fail.

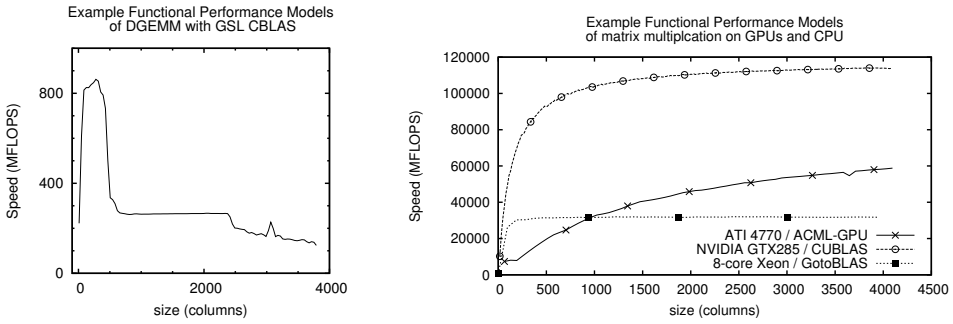
Conventional algorithms for distribution of computations between heterogeneous processors are based on a performance model that represents the speed of a processor by a constant positive number; computations are distributed between the processors in proportion to this speed of the processor. The constant characterising 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.

The fundamental assumption of the conventional algorithms based on the constant performance models (CPMs) is that the absolute speed of the processors does not depend on the size of the computational task. This assumption proved to be accurate enough if:

- The processors, between which we distribute computations, are all general-purpose ones of the traditional architecture,
- The same code is used for local computations on all processors, and
- The partitioning of the problem results in a set of computational tasks that are small enough to fit into the main memory of the assigned processors and large enough not to fit into the cache memory.

These conditions are typically satisfied when medium-sized scientific problems are solved on a heterogeneous network of workstations. Actually, heterogeneous networks of workstations were the target platform for the conventional heterogeneous parallel algorithms. However, the assumption that the absolute speed of the processor is independent of the size of the computational task becomes much less accurate in the following situations:

- The partitioning of the problem results in some tasks either not fitting into the main memory of the assigned processor and hence causing paging or fully fitting into faster levels of its memory hierarchy (Fig. 1).
- Some processing units involved in computations are not traditional general-purpose processors (say, accelerators such as GPUs or specialised cores). In this case, the relative speed of a traditional processor and a non-traditional one may



(a) (b)

Fig. 1. Functional performance models showing speed against problem size for matrix multiplication. (a) Non-optimised GSL CBLAS library on single core Xeon. (b) Optimised BLAS librarys on two GPU's and an 8 core Xeon processor [3].

differ for two different sizes of the same computational task even if both sizes fully fit into the main memory.

- Different processors use different codes to solve the same computational problem locally.

The above situations become more and more common in modern and especially perspective high-performance heterogeneous platforms. As a result, applicability of the traditional CPM-based distribution algorithms becomes more restricted. Indeed, if we consider two really heterogeneous processing units P_i and P_j, then the more different they are, the smaller will be the range R_{ij} of sizes of the computational task where their relative speed can be accurately approximated by a constant. In the case of several different heterogeneous processing units, the range of sizes where CPM-based algorithms can be applied will be given by the intersection of these pair-wise ranges, $\bigcap_{i,j=1}^{p} R_{ij}$. Therefore, if a high-performance computing platform includes a relatively large number of significantly heterogeneous processing units, the range of applicability of CPM-based algorithms may become quite small or even empty. For such platforms, new algorithms are needed that would be able to optimally distribute computations between processing units for the full range of problem sizes.

The functional performance model (FPM) of heterogeneous processors proposed and analysed in [4] has proven to be more realistic than the constant performance models because it integrates many important features of heterogeneous processors such as the architectural and platform heterogeneity, the heterogeneity of memory structure, the effects of paging and so on. The algorithms employing it therefore distribute the computations across the heterogeneous processing units more accurately than the algorithms employing the constant performance models. Under this model, the speed of each processor is represented by a continuous function of the size of the problem. While this model is application centric because, generally speaking, differ-

ent applications will characterise the speed of the processor by different functions, a speed function is supposed to satisfy some general restrictions on its shape [4]. In particular, beginning from some point, it should be monotonically decreasing.

The cost of experimentally building the full functional performance model of a processor, i.e., the model for the full range of problem sizes, is very high. This is due to several reasons. To start with, the accuracy of the model depends on the number of experimental points used to build it. The larger the number, the more accurate the model 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 functional performance model increases remarkably as the number of parameters used to represent the problem size increases, as shown in the experimental results in this paper.

The high model-construction cost limits the applicability of parallel algorithms based on full FPMs to situations where the construction of the full FPMs of heterogeneous processors and their use in the application can be separated. For example, if we develop an application for dedicated stable heterogeneous platforms, with the intention of executing the application on the same platform multiple times, we can build the full FPMs for each processor of the platform once and then use these models multiple times during the repeated execution of the application. In this case, the time of construction of the FPMs can become very small compared to the accumulated performance gains during the multiple executions of the optimised application. However, this approach does not apply to applications for which each run is considered unique. This is the case for applications that are intended to be executed in dynamic environments or any other environments where the available processors and their performance characteristics can change. This is also the case for applications that can be run just once or a small number of times in each environment. Such applications should be able to optimally distribute computations between the processors of the executing platform assuming that the platform is different and *a priori* unknown for each run of the application. In this paper, we call applications that automatically adapt at runtime to any set of heterogeneous processors with a priori unknown performance characteristics *self-adaptable* applications.

This paper presents new dynamic load balancing algorithms for data-intensive iterative routines on highly heterogeneous computational clusters. In contrast to the traditional algorithms, our algorithms are adaptive and take into account different aspects of heterogeneity. They assume that the speed of processors depends on the problem size but do not require the speed functions to be given. Instead, they estimate the speed functions of the processors for different problem sizes during iterations. The algorithms do not construct the complete speed function for each processor but rather build and use a partial estimate, which is sufficient for optimal distribution of computations. Load balancing decisions are based on the functional performance models, which are constantly improved with each iteration

[5]. Use of the functional performance models allows a computational scientist to utilise the maximum available resources on a given cluster. We demonstrate that our algorithms succeed in balancing the load even in situations when the traditional algorithms fail.

This paper is structured as follows. In Section 2, related work is discussed. In Section 3, we describe the target class of iterative routines and the design of traditional load balancing algorithms. Then we analyse the shortcomings of the traditional algorithms and illustrate them by experimental results. In Section 4, we propose a design of dynamic load balancing algorithms based on functional performance models, which overcomes the shortcomings of the traditional algorithms. In Sections 5 and 6, we present two particular algorithms implementing this design. The algorithms use different data partitioning algorithms and different approximations of speed functions. We demonstrate that, unlike the traditional algorithms, they can successfully balance data-intensive iterative routines for the whole range of problem sizes.

2. Related Work

In this section, we classify load balancing algorithms and discuss their applicability to data-intensive iterative routines and dedicated computational clusters.

Load balancing algorithms can be either static or dynamic. **Static** algorithms [6, 7, 8] use *a priori* information about the parallel application and platform. This information can be gathered either at compile-time or run-time. These strategies are restricted to applications with pre-determined workload and cannot be applied to such iterative routines as adaptive mesh refinement [9], for which the amount of computation data grows unpredictably. **Dynamic** algorithms [10, 11, 12, 13, 14] do not require *a priori* information and can be used with a wider class of parallel applications. In addition, dynamic algorithms can be deployed on non-dedicated platforms. The algorithms we present in this paper are dynamic.

Another classification is based on how load balancing decisions are made: in a centralised or non-centralised manner. In **non-centralised** algorithms [13, 14], load is migrated locally between neighbouring processors, while in **centralised** ones [6, 7, 8, 10, 11, 12], load is distributed based on global load information. Non-centralised algorithms are slower to converge. At the same time, centralised algorithms typically have higher overhead. Our algorithms belong to the class of centralised algorithms.

Centralised algorithms can be subdivided into two groups: task queue and predicting the future [2]. **Task queue** algorithms [11, 12] distribute independent tasks and schedule them on shared-memory platforms; hence they are not suitable for iterative routines on a distributed memory platform. **Predicting-the-future** algorithms [6, 7, 8, 10] can distribute both tasks and data by predicting future performance based on past information. They are suitable for data-intensive iterative routines and any parallel computational platform.

The traditional approach taken for load balancing of data-intensive iterative routines belongs to static/dynamic centralised predicting-the-future algorithms. In these traditional algorithms, computation load is evaluated either in the first few iterations [8] or at each iteration [10] and globally redistributed among the processors. Current speed measurements are used to predict future performance. As it will be demonstrated in Section 3, when applied to large scientific problems and highly heterogeneous parallel platforms, this strategy may never balance the load, because it uses simplistic models of processors' performance.

It has been shown in [4] that it is more accurate to represent performance as a function of problem size. In this paper, we propose new dynamic load balancing algorithms based on partial functional performance models of processors [5]. Unlike traditional algorithms, our algorithms impose no restriction on problem sizes.

We would also like to mention some advanced load balancing strategies that are not directly applicable to data-intensive iterative routines on heterogeneous clusters. It has been shown that the task queue model implemented in [12] can outperform the model [11] because decisions are based on adaptive speed measurements rather than single speed measurements. The algorithm presented in this paper also applies an adaptive performance model, but in such a way that it is applicable to scientific computational iterative routines.

In this paper, we focus on dynamic load balancing with respect to computational performance, and to this end, we do not take into account communication heterogeneity. Future work could be the development of a hybrid approach, similar to [7], in which our algorithms will be combined with one of the many existing communication models.

3. Traditional Algorithms of Dynamic Load Balancing of Iterative Routines

Iterative routines have the following structure: $x^{k+1} = f(x^k)$, $k = 0, 1, ...$ with x^0 given, where each x^k is an n-dimensional vector, and f is some function from $\mathbb{R}^n$ into itself [14]. The iterative routine can be parallelized on a cluster of p processors by letting x^k and f be partitioned into p block-components. During an iteration, each processor calculates its assigned elements of x^{k+1}. Therefore, each iteration is dependent on the previous one.

The objective of load balancing algorithms for iterative routines is to distribute computations across a cluster of heterogeneous processors in such a way that all processors will finish their computation within the same time and thereby minimising the overall computation time: $t_i = t_j$, $1 \leq i, j \leq p$. The computation, consisting of n computational units, is spread across a cluster of p processors $P_1, ..., P_p$ such that $p \ll n$. Processor P_i contains d_i elements of x^k and f, such that $n = \sum_{i=1}^{p} d_i$.

Traditional load balancing algorithms work by measuring the computation time of one iteration, calculating the new distribution and redistributing the workload, if necessary, for the next iteration. A typical algorithm works as follows:

Initially. The computation workload is distributed evenly between all processors, $d_i^0 = n/p$. All processors execute n/p computational units in parallel.
At each iteration.

(1) The computation execution times $t_1(d_1^k), ..., t_p(d_p^k)$ for this iteration are measured on each processor and gathered to the root processor.
(2) If $\max_{1 \le i,j \le p} \left| \frac{t_i(d_i^k) - t_j(d_j^k)}{t_i(d_i^k)} \right| \le \varepsilon$ then the current distribution is considered balanced and redistribution is not needed.
(3) Otherwise, the root processor calculates the new distribution of computations $d_1^{k+1}, ..., d_p^{k+1}$ as $d_i^{k+1} = n \times s_i^k / \sum_{j=1}^{p} s_j^k$, where s_i^k is the speed of the $i'th$ processor given by $s_i^k = d_i^k / t_i(d_i^k)$.
(4) The new distribution $d_1^{k+1}, ..., d_p^{k+1}$is broadcast to all processors and where necessary data is redistributed accordingly.

3.1. *Analysis of traditional load balancing*

The traditional load balancing algorithm is based on the assumption that the absolute speed of a processor depends on problem size but the speed is represented by a constant at each iteration. This strategy can work well in regions where speed is approximately invariant with problem size as depicted in Fig. 2. The problem is initially divided evenly between two processors for the first iteration and then redistributed to the optimal distribution at the second iteration.

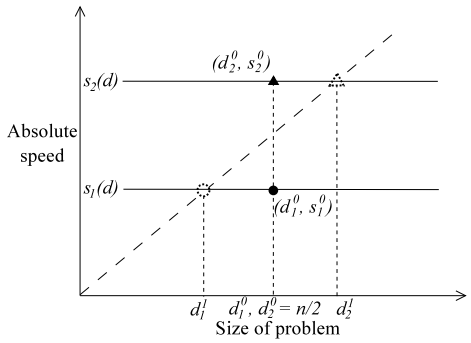


Fig. 2. Traditional load balancing algorithm successfully applied to two processors in a region where speed is invariant with problem size. Initially the problem is partitioned evenly and the execution time is measured. Based on this measurement the algorithm computes a new distribution (outlined points). This new distribution will be successful as the points lie on the speed functions $s_1(d)$ and $s_2(d)$.

Consider the situation in which the problem can still fit within the total main memory of the cluster but the problem size is such that the memory requirement of n/p is close to the available memory of one of the processors. In this case paging can occur. If paging does occur, the traditional load balancing algorithm is no longer

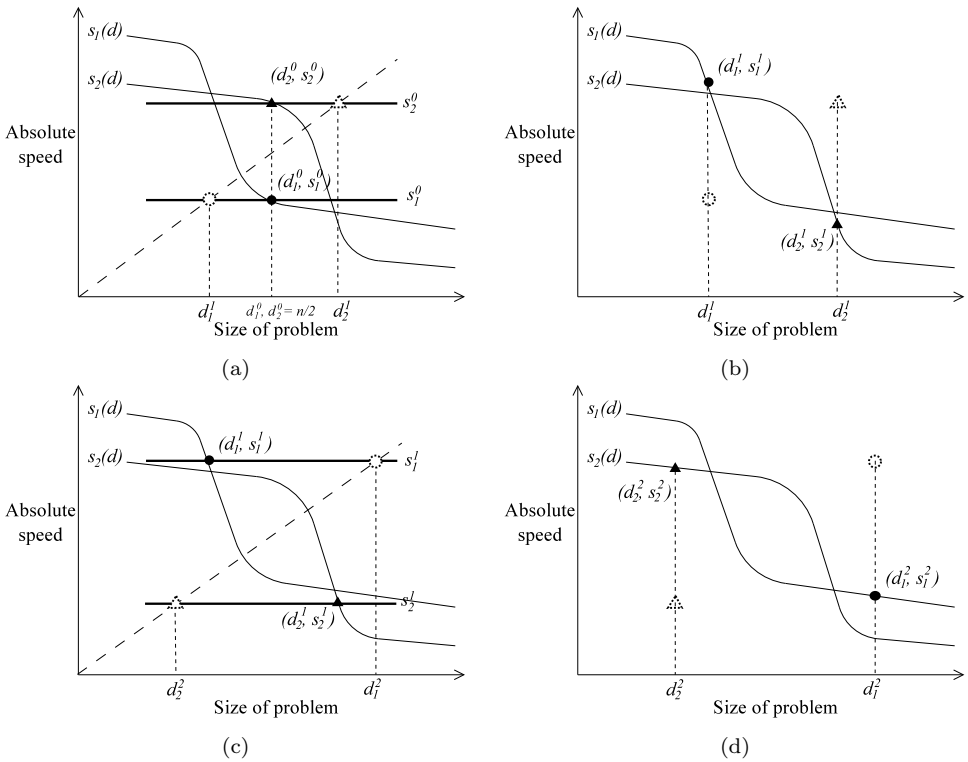


Fig. 3. Traditional load balancing algorithm applied to two processors in a region where the speed varies with problem size. Hence, the algorithm is unable to achieve balance. (a) Initially speed is measured for an equal data distribution and the algorithm computes a new distribution with a predicted speed (outlined points). (b) The difference between the predicted and actual speed of the processors measured at the second iteration. (c) Based on the speed measurements from the second iteration, the constant models are recalculated and a new distribution is computed. (d) At the third iteration, there is a large difference between the predicted speed and the actual speed.

adequate. This is illustrated for two processors in Fig. 3. Let the real performance of processors P_1 and P_2 be represented by the speed functions $s_1(x)$ and $s_2(x)$ respectively. Processor P_1 is a faster processor but with less main memory than P_2. The speed function drops rapidly at the point where main memory is full and paging is required. First, n independent units of computations are evenly distributed, $d_1^0 = d_2^0 = n/2$, between the two processors and the speeds of the processors, s_1^0, s_2^0, are measured Fig. 3(a). Then at the second iteration the computational units are divided according to $\frac{d_1^1}{d_2^1} = \frac{s_1^0}{s_2^0}$, where $d_1^1 + d_2^1 = n$. Therefore at the second iteration, P_1 will execute less computational units than P_2. However, P_1 will perform much faster and P_2 will perform much slower than the model predicts, Fig. 3(b). Moreover the speed of P_2 at the second iteration is slower than P_1 at the first iteration.

Based on the speeds of the processors demonstrated at the second iteration, their constant performance models are changed accordingly, Fig. 3(c), and the com-

putational units are redistributed again for the third iteration as: $\frac{d_1^2}{d_2^2} = \frac{s_1^1}{s_2^1}$, where $d_1^2 + d_2^2 = n$. Now the situation is reversed, P_2 performs much faster than P_1, Fig. 3(d). This situation will continue in subsequent iterations with the algorithm never converging. The majority of the computational units will oscillate between the processors.

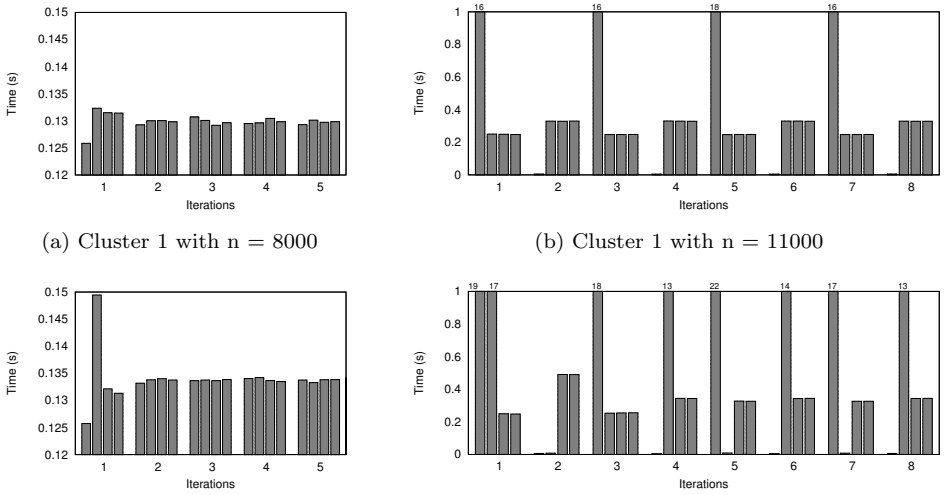
3.2. *Experimental results of the traditional load balancing algorithm*

The traditional load balancing algorithm was applied to the Jacobi method, which is representative of the class of iterative routines we study. The program was tested successfully on a cluster of 16 processors. For clarity the results presented here are from two configurations of 4 processors, Table 1. The clusters differ by the number of processors with 256MB RAM.

Table 1. Specifications of Cluster 1 (P_1, P_3, P_4, P_5) and Cluster 2 (P_1, P_2, P_3, P_4.)

	P_1	P_2	P_3	P_4	P_5
Processor	3.6 Xeon	3.0 Xeon	3.4 P4	3.4 Xeon	3.4 Xeon
RAM (MB)	256	256	512	1024	1024

The memory requirement of the partitioned routine is a $n \times d_i$ block of a matrix, three n dimensional vectors and some additional arrays of size p. For 4 processors with an even distribution, problem sizes of n=8000 and n=11000 will have a



(a) Cluster 1 with n = 8000

(b) Cluster 1 with n = 11000

(c) Cluster 2 with n = 8000

(d) Cluster 2 with n = 11000

Fig. 4. Time taken for each of the 4 processors to complete their assigned computational units during iterations. In (a) and (c) the problem fits in main memory and the load converges to a balanced solution. In (b) and (d) paging occurs on some machines and the load remains unbalanced.

memory requirement which lies either side of the available memory on the 256MB RAM machines, and hence will be good values for benchmarking.

The traditional load balancing algorithm worked efficiently for small problem sizes, Fig. 4(a,c). For problem sizes sufficiently large to potentially cause paging on some machines the load balancing algorithm caused divergence as the theory, in Section 3.1, predicted, Fig. 4(b,d).

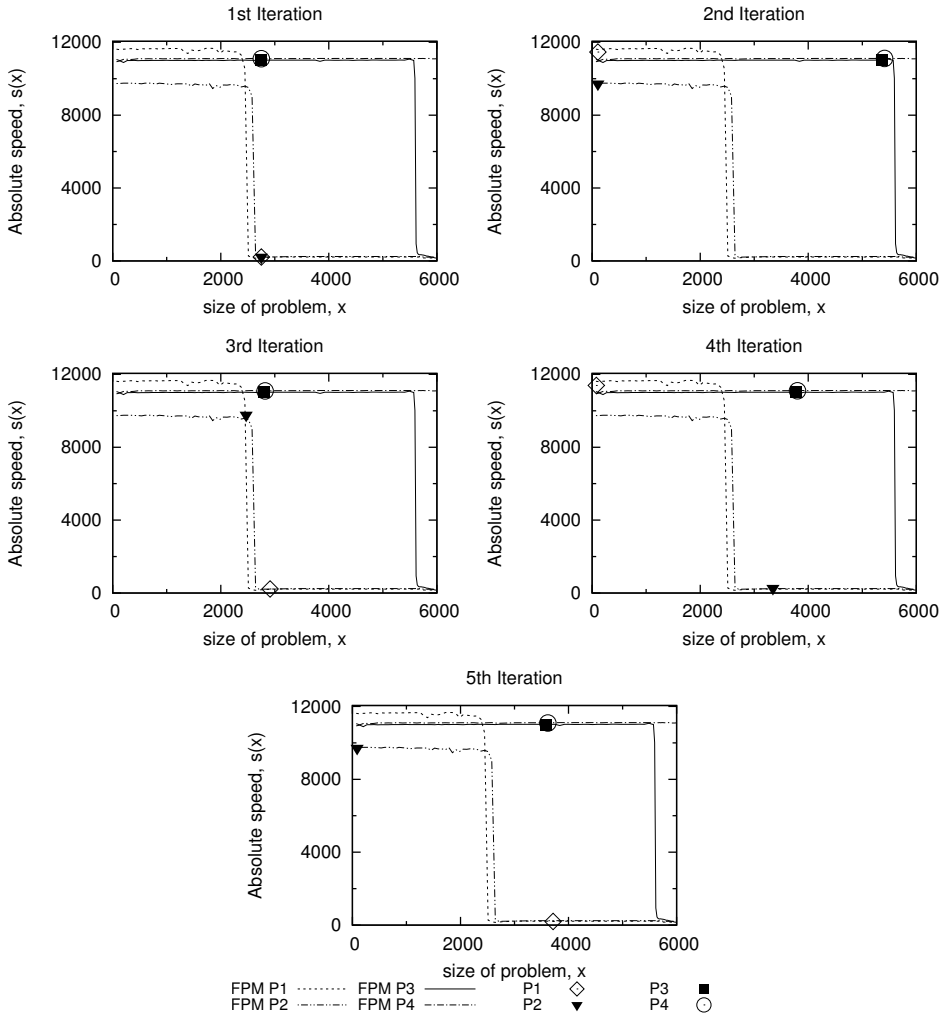


Fig. 5. Traditional load balancing algorithm for four processors on cluster 2 with n=11000. Showing initial distribution at n/4 and four subsequent iterations. The x axis represents the number of elements of x' computed by each processor as well as representing the memory requirements of the problem, namely, the number of rows of the matrix stored in memory. The full functional performance models are dotted in to aid visualisation.

A plot of problem size vs. absolute speed can help to illustrate why the traditional load balancing algorithm is failing for large problems. Fig. 5 shows the absolute speed of each of the processors for the first five iterations.

The experimentally built full functional models for the processors are dotted in to aid visualisation, but this information was not available to the load balancing algorithm. Initially each processor has $n/4$ rows of the matrix. At the second iteration, $P_1$and P_2 are given very few rows as they both performed slowly at the first iteration, however they now compute these few rows quickly. At the third iteration, P_1 is given sufficient rows to cause paging and hence a cycle of oscillating row allocation ensues.

Since data partitioning is employed in our iterative routine, it is necessary to redistribute data after each change of distribution. When the balancing algorithm converges quickly to an optimum distribution, the network load from data redistribution is acceptable. However, if the distribution oscillates, not only is the computation time affected but there will also be a heavy load on the network. On cluster 2 with $n = 11000$ approximately 300MB is been passed back and forth between P_1 and P_2 with each iteration.

4. Dynamic Load Balancing Based on Functional Performance Models

Functions much more accurately represent the speed of processors than constants [15]. Being application-centric and hardware-specific, functional performance models reflect different aspects of heterogeneity. To overcome the shortcomings of the traditional algorithms of dynamic load balancing, we propose a new design, which uses functional models of processors instead of single speed values. In this section, we define the key features of the dynamic load balancing algorithms based on functional performance models.

The speeds of p processors are represented by positive continuous functions of problem size $s_1(x), ..., s_p(x)$: $s_i(x) = \frac{x}{t_i(x)}$, where $t_i(x)$ is the execution time for processing of x elements on the processor i. Speed functions are defined at $[0, n]$, where n is a problem size to partition. As in traditional algorithms, load balancing is achieved when all processors execute their work within the same time: $t_1(x_1) = ... = t_p(x_p)$. This can be expressed as:

$$\frac{x_1}{s_1(x_1)} = ... = \frac{x_p}{s_p(x_p)} \text{ , where } x_1 + x_2 + ... + x_p = n \quad (1)$$

The solution to these equations, $d_1, ..., d_p$, can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system (Fig. 6).

In practice, the speed function can be built as a piecewise linear function fitting within a band of historic records of workload fluctuations of the processor [15]. This building procedure is very time consuming, especially for full functional performance models, which are characterised by numerous points. Generating the speed functions

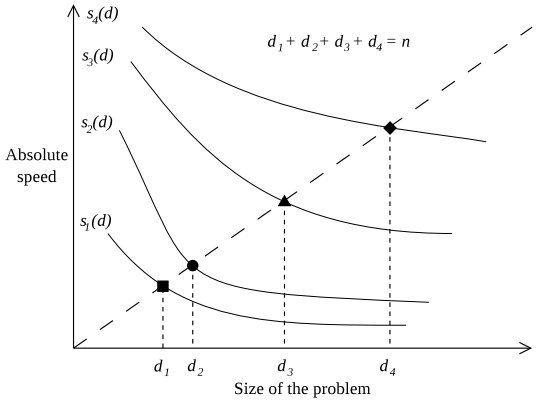


Fig. 6. Optimal distribution of computational units showing the geometric proportionality of the number of chunks to the speeds of the processors.

is especially expensive in the presence of paging. For example, the measurement of just 20 points for the function in Fig. 5 took approximately 1473 seconds, 4 times longer than the actual calculation with a homogeneous distribution for 20 iterations. This forbids building full functional performance models at run time.

To reduce the cost of building the speed functions, the partial functional performance models were proposed [5]. They are based on a few points and estimate the real functions in detail only in the relevant regions: $\bar{s}_i(x) \approx s_i(x)$, $1 \leq i \leq p$, $\forall x \in [a, b]$. Both the partial models and the regions are determined at runtime, while the data partitioning algorithm is iteratively applied to the partially built speed functions. The result of the data partitioning, the estimate of the optimal data distribution $d_1^k, ..., d_p^k$, determines the next experimental points $(d_1^k, s_1(d_i^k)), ..., (d_p^k, s_p(d_p^k))$ to be added to the partial models $\bar{s}_1(x), ..., \bar{s}_p(x)$. The more points are added, the closer the partial functions approximate the real speed functions in the relevant regions. The low cost of partially building the models makes it ideal for employment in self-adaptive parallel applications, particularly in dynamic load balancing. The partial models can be built during the execution of the computational iterative routine.

This work studies dynamic load balancing of computational iterative routines. The performance of such routines can be represented by the speed of a single iteration as all iterations perform the same amount of computation. We propose a new design of dynamic load balancing algorithm that is based on partial speed functions instead of single speed values. It can be summarised as follows. At the iteration k of the routine:

(1) The data is distributed in accordance with the partition obtained at the previous iteration $d^k = (d_1^k, ..., d_p^k)$. For $k = 0$, the data is distributed evenly: $d^0 = (n/p, ..., n/p)$.

(2) The computation is executed and its performance is evaluated on all processors $s_1^k, ..., s_p^k$.
(3) The new observation points $\left(d_1^k, s_1^k\right), ..., \left(d_p^k, s_p^k\right)$ are added to the partial performance models of processors and ***approximations of the speed functions*** $\bar{s}_1(x), ..., \bar{s}_p(x)$ are recalculated.
(4) ***Data partitioning algorithm*** is applied to the current approximations of the speed functions and returns the refined partition d^{k+1} for the next iteration.

Since $\bar{s}_i(x) \to s_i(x)$ as $k \to \infty$, $1 \leq i \leq p$, this procedure adaptively converges to the optimal data distribution $d^k \to d^*$.

In Sections 5 and 6, we present two algorithms implementing this design. They both execute steps 1-4 but use different data partitioning algorithms and different approximations of the speed functions. The first algorithm is based on the geometrical data partitioning algorithm proposed in [4]. It imposes some restrictions on the shape of speed functions but guarantees the existence and uniqueness of the optimal data partitioning.

The second algorithm is based on a new data partitioning algorithm proposed in this paper. This new algorithm formulates the original data partitioning problem as a problem of finding a solution to a multi-dimensional system of nonlinear equations. It employs a numerical multi-dimensional non-linear solver to find the optimal partitioning. It is not that restrictive to the shape of speed functions as the geometrical algorithm and therefore can use more accurate approximations of the real-life speed functions. However, the second algorithm does not guarantee a unique solution.

5. Dynamic Load Balancing Algorithm Based on Geometrical Data Partitioning

In this section, we present the algorithm balancing the computational load with help of the data partitioning algorithm proposed in [4]. It is based on the geometrical solution of the problem (1), assuming that any straight line passing through the origin of the coordinate system intersects the speed functions only once.

To ensure the existence of a unique optimal data distribution some restrictions were placed on the shape of the speed functions. Experiments performed with many scientific kernels on various heterogeneous networks of workstations have demonstrated that, in general, processor speed could be approximated, within some acceptable degree of accuracy, by a function satisfying the following assumptions [4]:

(1) On the interval $[0, X]$, the function is monotonically increasing and concave.
(2) On the interval $[X, \infty]$, the function is monotonically decreasing.

This guarantees that any straight line passing through the origin of the coordinate system intersects the graph of the function in no more than one point.

5.1. *Data partitioning algorithm*

Any line passing through the origin and intersecting the speed functions represents an optimum distribution for a particular problem size. Therefore, the space of solutions of the problem (1) consists of all such lines. The two outer bounds of the solution space are selected as the starting point of algorithm. The upper line represents the optimal data distribution $d_1^u, ..., d_p^u$ for some problem size $n_u < n$, $n_u = d_1^u + ... + d_p^u$, while the lower line gives the solution $d_1^l, ..., d_p^l$ for $n_l > n$, $n_l = d_1^l + ... + d_p^l$. The region between two lines is iteratively bisected.

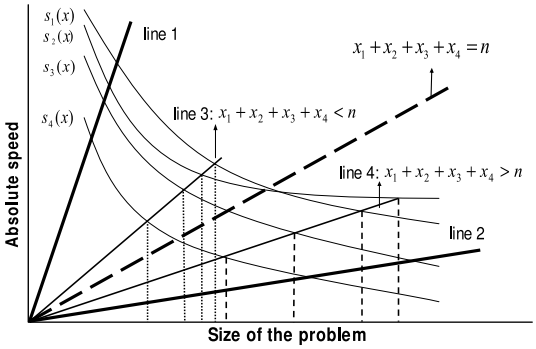


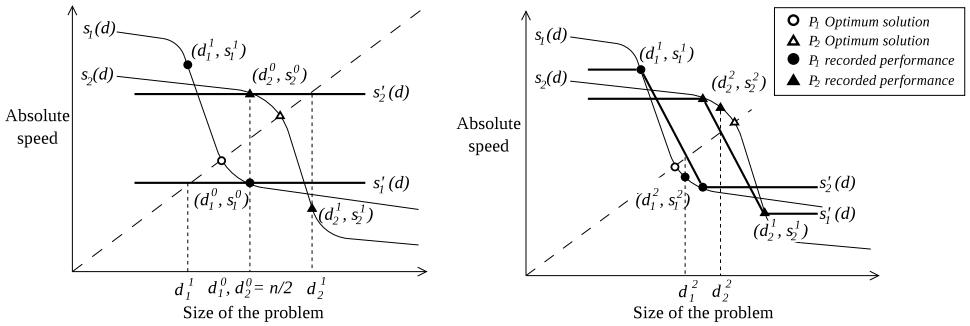
Fig. 7. Geometrical data partitioning algorithm. Line 1 (the upper line) and line 2 (the lower line) represent the two initial outer bounds of the solution space. Line 3 represents the first bisection. Line 4 represents the second one. The dashed line represents the optimal solution.

At the iteration k, the problem size corresponding to the new line intersecting the speed functions at the points $d_1^k, ..., d_p^k$ is calculated as $n_k = d_1^k + ... + d_p^k$. Depending on whether n_k is less than or greater than n, this line becomes a new upper or lower bound. Making n_k close to n, this algorithm finds the optimal partition of the given problem $d_1, ..., d_p$: $d_1 + ... + d_p = n$. The assumptions about the shape of the speed functions provide the existence and uniqueness of the solution. Fig. 7 illustrates the work of the bisection algorithm.

5.2. *Approximation of partial speed functions*

Let us outline how the partial functions $\bar{s}_i(x)$, $1 \le i \le p$, are constructed in the dynamic load balancing algorithm based on the geometrical data partitioning. The first approximations of the partial speed functions, $\bar{s}_i(x)$, are created as constants $\bar{s}_i(x) = s_i^0 = s_i(n/p)$, Fig. 8(a). At the iteration k, the piecewise linear approximations $\bar{s}_i(x)$ are improved by adding the points (d_i^k, s_i^k), Fig. 8(b). Namely, let $\{(d_i^{(j)}, s_i^{(j)})\}_{j=1}^m$, $d_i^{(1)} < \ldots < d_i^{(m)}$, be the experimentally obtained points of $\bar{s}_i(x)$ used to build its current piecewise linear approximation, then

(1) If $d_i^k < d_i^{(1)}$, then the line segment $(0, s_i^{(1)}) \rightarrow (d_i^{(1)}, s_i^{(1)})$ of the $\bar{s}_i(x)$ approximation will be replaced by two connected line segments $(0, s_i^k)) \rightarrow (d_i^k, s_i^k)$ and $(d_i^k, s_i^k) \rightarrow (d_i^{(1)}, s_i^{(1)})$;
(2) If $d_i^k > d_i^{(m)}$, then the line $(d_i^{(m)}, s_i^{(m)}) \rightarrow (\infty, s_i^{(m)})$ of this approximation will be replaced by the line segment $(d_i^{(m)}, s_i^{(m)}) \rightarrow (d_i^k, s_i^k)$ and the line $(d_i^k, s_i^k) \rightarrow (\infty, s_i^k)$;
(3) If $d_i^{(j)} < d_i^k < d_i^{(j+1)}$, the line segment $(d_i^{(j)}, s_i^{(j)}) \rightarrow (d_i^{(j+1)}, s_i^{(j+1)})$ of $\bar{s}_i(d)$ will be replaced by two connected line segments $(d_i^{(j)}, s_i^{(j)}) \rightarrow (d_i^k, s_i^k)$ and $(d_i^k, s_i^k) \rightarrow (d_i^{(j+1)}, s_i^{(j+1)})$.



(a) (b)

Fig. 8. Construction of partial speed functions using linear interpolation.

After adding the new data point (d_i^j, s_i^j) to the partial speed function $\bar{s}_i(x)$, we verify that the shape of the resulting piecewise linear approximation satisfies the above assumptions, and update the value of s_i^j when required. Namely, to keep the partial speed function increasing and convex on the interval $[0, X]$, we ensure that $s_i^{j-1} \leq s_i^j \leq s_i^{j+1}$ and $\frac{s_i^{j-1}-s_i^{j-2}}{d_i^{j-1}-d_i^{j-2}} \geq \frac{s_i^j-s_i^{j-1}}{d_i^j-d_i^{j-1}} \geq \frac{s_i^{j+1}-s_i^j}{d_i^{j+1}-d_i^j}$. The latter expression represents non-increasing tangent of the pieces, which is required for the convex shape of the piecewise linear approximation. On the interval $[X, \infty]$, we ensure that $s_i^{j-1} \geq s_i^j \geq s_i^{j+1}$ for monotonously decreasing speed function.

5.3. *Experimental results*

We tested our geometrical data partitioning algorithm on the same experimental setup as in Section 3.2. For small problem sizes ($n = 8000$, $p = 4$), our algorithm performed in much the same way as the traditional algorithm. For larger problem sizes ($n = 11000$), our algorithm was able to successfully balance the computational load within a few iterations (Fig. 9). As in the traditional algorithm, paging also occurred but our algorithm experimentally fit the problem to the available RAM. Paging at the 8^{th} iteration on P_1 demonstrates how the algorithm experimentally finds the

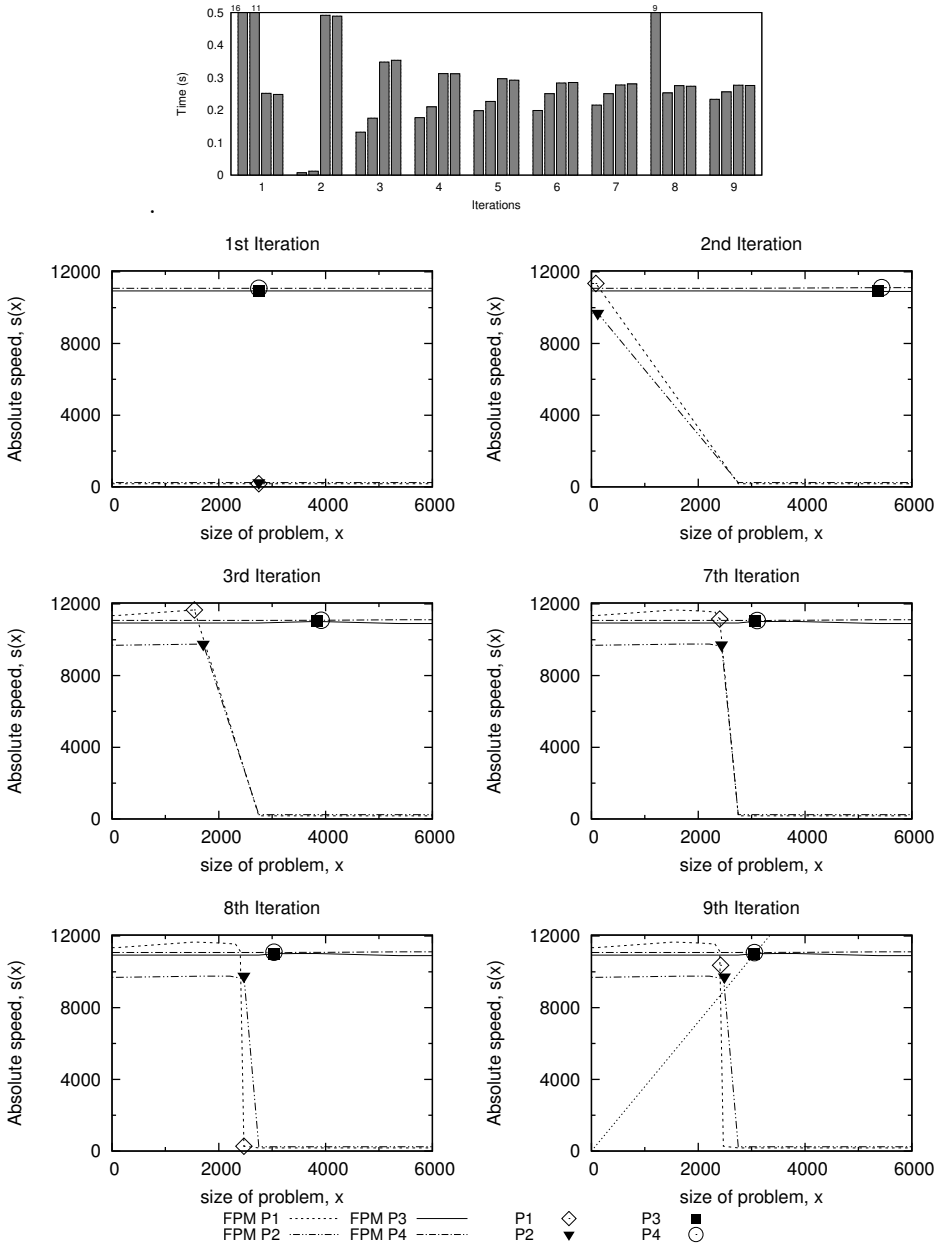
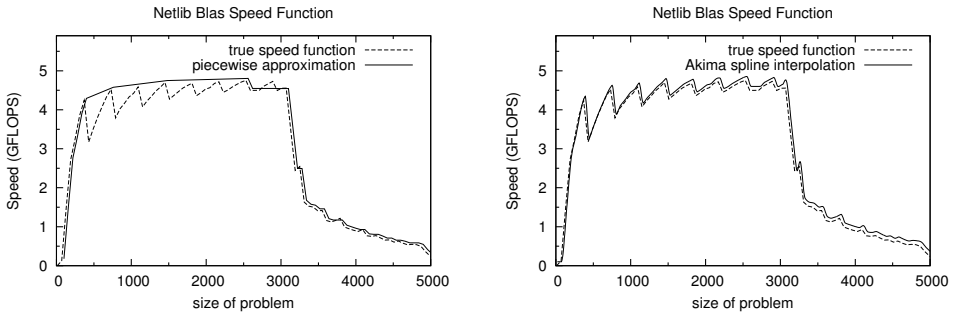


Fig. 9. Dynamic load balancing of Jacobi iterative routine with geometrical data partitioning. Problem size n=11000 on cluster 2. Bar graph shows time taken by each of the 4 processors to compute each iteration. Speed plots show dynamically built functional performance models. The line intersecting the origin represents the optimum solution and points converge towards this line.

memory limit of P_1. The 9^{th} iteration represents a near optimum distribution for the computation on this hardware. A plot of speed vs. problem size, Fig. 9, shows how the computational distribution approaches an optimum distribution within 9 iterations. We can see why P_1 performs slowly at the 8^{th} iteration. At the 9^{th} iteration, we can see that the maximum performance of processors P_1 and P_2 has been achieved.

6. Dynamic Load Balancing Algorithm Based on Multidimensional Root-Finding Data Partitioning

In order to converge to a solution, the algorithm in the previous section requires each speed function to be monotonically increasing and concave up to some point and then monotonically decreasing which guarantees that each speed function will be intersected only once by any line passing from the origin. In general speed functions have this shape, but it is not always the case. For example, a non-optimised algorithm such as Netlib BLAS can have a sawtooth function due to cache misses (Fig. 10). A less accurate function must be fitted to the data to satisfy the shape restrictions (Fig. 10(a)).



(a) (b)

Fig. 10. Speed function for non-optimised Netlib BLAS. (a) Fitting shape restricted piecewise approximation. (b) Fitting Akima spline interpolation. Both speed functions have been offset slightly for clarity.

Here we present a new data partitioning algorithm which removes these restrictions and therefore represents the speed of the processor with more accurate continuous functions. This allows us to perform more accurate partitioning. For example, by using the more accurate fit in Fig. 10(b), we can achieve a speedup of 1.26 for some problem sizes. For different routines this speedup could potentially be much bigger.

6.1. *Data partitioning algorithm based on nonlinear multidimensional root finding*

If the processor speeds are approximated by continuous differentiable functions of arbitrary shape, the problem of optimal data partitioning (1) can be formulated as ***multidimensional root finding*** for the system of nonlinear equations $F(x) = 0$, where

$$F(x) = \begin{cases} n - \sum_{i=1}^{p} x_i \\ \frac{x_i}{s_i(x_i)} - \frac{x_1}{s_1(x_1)}, \ 2 \le i \le p \end{cases} \tag{2}$$

$x = (x_1, ..., x_p)$ is a vector of real numbers corresponding a data partition $d = (d_1, ..., d_p)$. The first equation specifies to the distribution of n computational units between p processors. The rest specify the balance of computational load. The problem (2) can be solved by a number of iterative algorithms based on the Newton–Raphson method:

$$x_{k+1} = x_k - J(x_k)F(x_k) \tag{3}$$

The equal data distribution

$$x^0 = (n/p, ..., n/p) \tag{4}$$

can be reasonably taken as the initial guess for the location of the root. $J(x)$ is a Jacobian matrix, which can be calculated as follows:

$$J(x) = \begin{pmatrix} -1 & -1 & ... & -1 \\ -\frac{s_1(x_1) - x_1 s_1'(x_1)}{s_1^2(x_1)} & \frac{s_2(x_2) - x_2 s_2'(x_2)}{s_2^2(x_2)} & 0 & 0 \\ ... & 0 & ... & 0 \\ -\frac{s_1(x_1) - x_1 s_1'(x_1)}{s_1^2(x_1)} & 0 & 0 & \frac{s_p(x_p) - x_p s_p'(x_p)}{s_p^2(x_p)} \end{pmatrix} \tag{5}$$

We use the HYBRJ algorithm, a modified version of Powell's Hybrid method, implemented in the MINPACK library [16]. It retains the fast convergence of the Newton method and reduces the residual when the Newton method is unreliable. Our experiments demonstrated that for the given vector-function (2) and initial guess (4), the HYBRJ algorithm is able to find the root $x^* = (x_1^*, ..., x_p^*)$, which will be the optimal data partition after rounding and distribution of remainders: $d = round(x^*)$.

At each iteration k of the dynamic load balancing algorithm, the problem (2)-(5) is solved for the current approximations of the speed functions $\bar{s}_i(x)$, $1 \le i \le p$. Since the functions are smooth, the root $x^* = (x_1^*, ..., x_p^*)$ will be found in a few steps of the multidimensional root finding algorithm. The optimal data partition for the next iteration will be obtained by rounding and distributing the remainders: $d^{k+1} = round(x^*)$.

6.2. *Approximation of partial speed functions*

Let us consider a set of k data points (x_i, s_i), $0 < x_i < n$, $1 \le i \le k$. Here and after in this section, the data points (x_i, s_i) correspond to a single processor, for which the speed function $s(x)$ is approximated. To approximate the speed function in the interval $[0, n]$, we also introduce two extra points: $(0, s_1)$ and (n, s_k). The linear interpolation does not satisfy the condition of differentiability at the breakpoints (x_i, s_i). The spline interpolations of higher orders have derivatives but may yield significant oscillations in the interpolated function. However, there is a special nonlinear spline, the ***Akima spline*** [17], that is stable to outliers (Fig. 11). It requires no less than 5 points. In the inner area $[x_3, x_{k-2}]$, the interpolation error has the order $O(h^2)$. This interpolation method does not require solving large systems of equations and therefore it is computationally efficient.

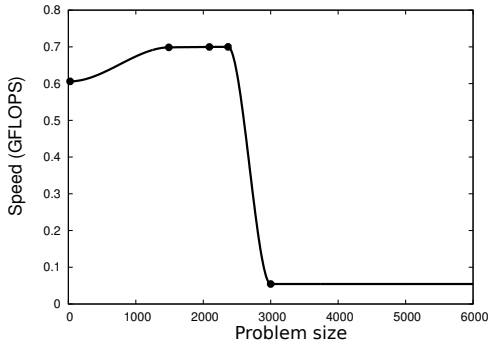


Fig. 11. Akima spline interpolation of a dynamically built functional performance model.

At the first few iterations, when the model consists of less than 5 data points, the Akima splines can be built for an extended model that duplicates the values of the left- and rightmost points, s_1, s_k, as follows:

(1) $k = 1$: $x_1 = n/p$, $s_1 = s(n/p)$, the extended model specifies the constant speed as $(0, s_1), \left(\frac{x_1}{2}, s_1\right), (x_1, s_1), \left(\frac{n-x_1}{2}, s_1\right), (n, s_1)$.
(2) $k = 2$: the extended model is $(0, s_1), (x_1, s_1), (x_2, s_2), \left(\frac{n-x_2}{2}, s_2\right), (n, s_2)$.
(3) $k = 3$: the extended model is $(0, s_1), (x_1, s_1), (x_2, s_2), (x_3, s_3), (n, s_3)$.

The interpolation is recalculated at each iteration of the routine.

Proposition 1. The speed functions s_i are defined within the range $0 < x \le n$ and are bounded, continuous, positive, non-zero and have bounded, continuous first derivatives.

Proof. The data points (x_i, s_i) are calculated with $s_i(x) = \frac{x}{t_i(x)}$. As it is a practical requirement that each iteration finishes in a finite time and the Akima splines closely fit the data points with continuous smooth functions, we can conclude

that s_i is continuous, bounded, positive, non-zero within the range $0 < x \leq n$. A feature of Akima splines is that they have continuous first derivatives [17].

6.3. *Convergence and complexity analysis*

Proposition 2. Within the range $0 < x \leq n$, the system of nonlinear equations $F(x) = 0$ contains no stationary points and the functions $f_i(x)$ have bounded, continuous first derivatives, where $f_i(x)$ is the i'th equation of $F(x)$.

Proof. $F'(x)$ is non-zero for all $0 < x \leq n$, hence $F(x)$ contains no stationary points. $f_0(x)$ has a constant first derivative. For $f_i(x)$, $1 \leq i < n$, if s_i and $s_i^{'}$ are continuous, bounded and if s_i is non zero then $f_i'(x)$ is bounded, continuous. This requirement is satisfied by proposition 1.

Proposition 3. The new data partitioning algorithm presented in this section converges in a finite number of iterations.

Proof. It is proven in [16] that if the range of x is finite, and $F(x)$ contains no stationary points and if $f_i'(x)$ is bounded continuous then the HYBRJ solver will converge to $|F(x)| < \varepsilon$, where ε is a small positive number, in a finite number of iterations. These requirements are satisfied by proposition 2.

Proposition 4. The complexity of one iteration of the solver is $O(p^2)$.

Proof. It is show in [18] that the HYBRJ solver has complexity $O(p^2)$. All other steps of the algorithm are of order $O(p)$.

The number of solver iterations depends on the shape of the functions. In practice we found that often 2 iterations are sufficient when the speed functions are very smooth and up to 30 iterations when partitioning in regions of rapidly changing speed functions.

6.4. *Experimental results*

Fig. 12 illustrates the work of this algorithm for the Jacobi method for 4 processors with $n = 12000$. The algorithm converges to the optimal data distribution with each iteration. By the 7^{th} iteration optimum partitioning has been achieved.

Fig. 13 shows the speedup of the CPM and FPM algorithms over a homogeneous distribution. The FPM algorithm used in the experiments is the one based on nonlinear multidimensional root finding. For small problem sizes the speedup is not realised because of the cost involved with data redistribution, however as the size increases both load balancing algorithms improve up to the point were the traditional algorithm based on a constant performance model fails, from which point it performs worse than the homogeneous distribution. The speed up achieved by FPM based load balancing increases as the difference between the relative speeds of the processors increases.

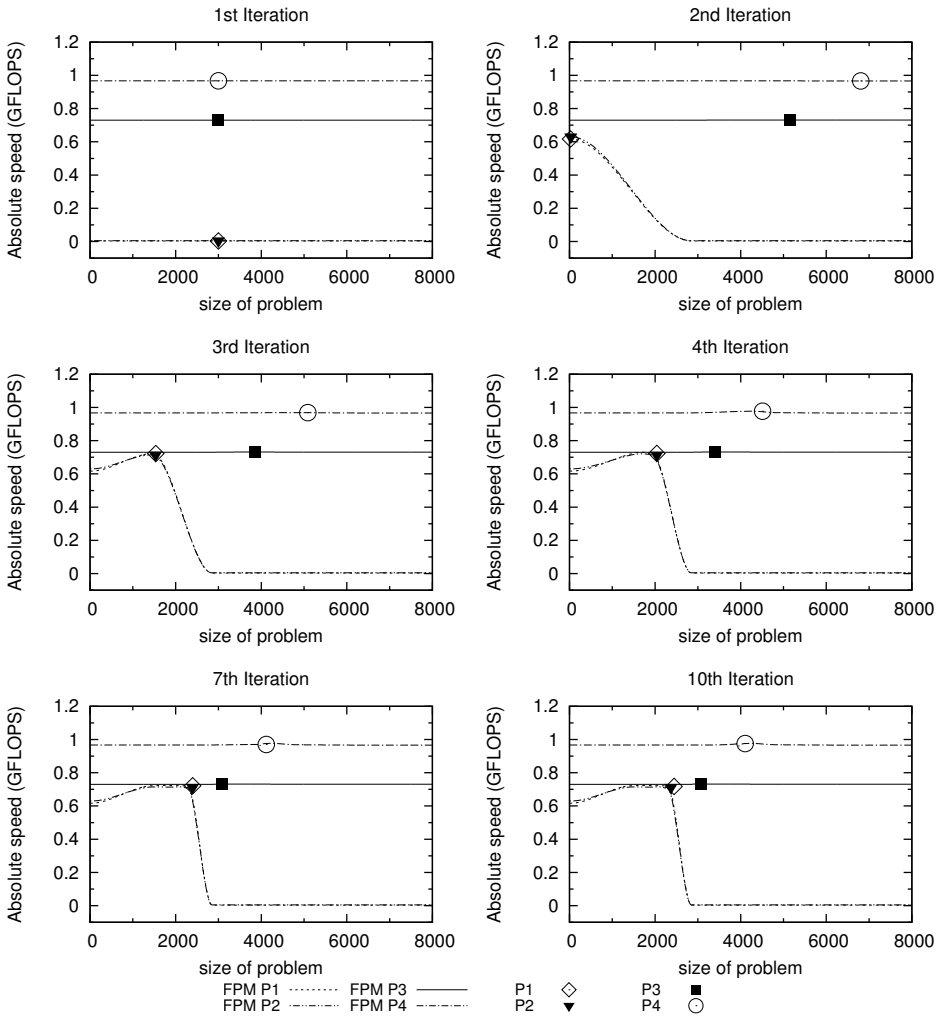


Fig. 12. Dynamic load balancing using multidimensional root-finding partitioning algorithm and the Akima spline interpolation for n=12000 on cluster 2.

7. Conclusion

In this paper, we have shown that traditional dynamic load balancing algorithms can fail on highly heterogeneous parallel platforms. They do not take into account all aspects of heterogeneity and use simplified models of processors' performance. To address this issue, we proposed the new design of dynamic load balancing based on functional performance models of processors. Then we implemented this design in two algorithms which use different methods of data partitioning and different approximations of partially built speed functions. The first, geometrical, algorithm is based on the data partitioning proposed in our previous work. It imposes some

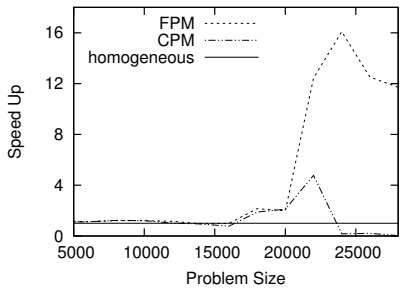


Fig. 13. Speed up of Jacobi iterative routine using dynamic load balancing algorithms over a homogeneous distribution of n/p on a cluster of 16 heterogeneous machines.

restrictions on the shape of speed functions but guarantees the existence and uniqueness of the optimal data partition. In this algorithm, partial functional models are approximated by piecewise linear functions. The second algorithm employs the new numerical solution of the data partitioning problem. It relaxes the restrictions on the shape of speed functions and numerically solves the system of non-linear equations which corresponds to the optimal data partitioning. We have shown that the dynamic load balancing algorithms based on functional models can be used successfully with any problem size and on a wide class of heterogeneous platforms.

Acknowledgements

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

References

[1] Bharadwaj, V., Ghose, D., Robertazzi, T.G., Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. *Cluster Comput.* 6, 7–17 (2003)
[2] Cierniak, M., Zaki, M.J., Li, W., Compile-Time Scheduling Algorithms for Heterogeneous Network of Workstations. *Computer J.* 40, 356–372 (1997)
[3] Higgins, R., Modelling the Performance of Processors in Heterogeneous Computing Environments. PhD Thesis, School of Computer Science and Informatics, University College Dublin, 2011.
[4] Lastovetsky, A., Reddy, R., Data Partitioning with a Functional Performance Model of Heterogeneous Processors. *Int. J. High Perform. Comput. Appl.* 21, 76–90 (2007)
[5] Lastovetsky, A., Reddy, R., Distributed Data Partitioning for Heterogeneous Processors Based on Partial Estimation of their Functional Performance Models. In: *HeteroPar'2009.* LNCS, vol. 6043, pp. 91–101. Springer (2010)
[6] Ichikawa, S., Yamashita, S., Static Load Balancing of Parallel PDE Solver for Distributed Computing Environment. In: *PDCS-2000*, pp. 399–405. ISCA (2000)
[7] Legrand, A., Renard, H., Robert, Y., Vivien, F., Mapping and load-balancing iterative computations. *IEEE T. Parall. Distr.* 15, 546–558 (2004)
[8] Martínez, J.A., Garzón, E.M., Plaza, A., García, I., Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE. *J. Supercomput.* (published online 5 November 2009)

[9] Li, X.-Y., Teng, S.-H., Dynamic Load Balancing for Parallel Adaptive Mesh Refinement. In: *IRREGULAR'98* pp. 144–155. Springer (1998)
[10] Galindo, I., Almeida, F., Badía-Contelles, J. M., Dynamic Load Balancing on Dedicated Heterogeneous Systems. In: *EuroPVM/MPI 2008*, pp. 64–74. Springer (2008)
[11] Hummel, S.F., Schmidt, J., Uma, R. N., Wein, J., Load-sharing in heterogeneous systems via weighted factoring. In: *SPAA'96*, pp. 318–328. ACM (1996)
[12] Cariño, R.L., Banicescu, I., Dynamic load balancing with adaptive factoring methods in scientific applications. *J. Supercomput.* 44, 41–63 (2008)
[13] Cybenko, G., Dynamic load balancing for distributed memory multi-processors. *J. Parallel Distr. Com.* 7, 279–301 (1989)
[14] Bahi, J.M., Contassot-Vivier, S., Couturier, R., Dynamic Load Balancing and Efficient Load Estimators for Asynchronous Iterative Algorithms. *IEEE T. Parall. Distr.* 16, 289–299 (2005)
[15] Lastovetsky, A., Reddy, R., Higgins, R., Building the Functional Performance Model of a Processor. In: *SAC 2006*, pp. 746–753. ACM (2006)
[16] M.J.D. Powell, A Hybrid Method for Nonlinear Equations. In: *Gordon and Breach, Eds, Numerical Methods for Nonlinear Algebraic Equations*, pp., 87–114 (1970)
[17] Akima, H., A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedures. *J. ACM* 17, 589–602 (1970)
[18] M.J.D. Powell, A Fortran Subroutine for Solving Systems on Nonlinear Algebraic Equations. In: *Gordon and Breach, Eds, Numerical Methods for Nonlinear Algebraic Equations*, pp. 115–161 (1970)