FuPerMod: a Framework for Optimal Data Partitioning for Parallel Scientific Applications on Dedicated Heterogeneous HPC Platforms

David Clarke, Ziming Zhong, Vladimir Rychkov, and Alexey Lastovetsky

School of Computer Science and Informatics, University College Dublin,
Belfield, Dublin 4, Ireland,
`{david.clarke, ziming.zhong}@ucdconnect.ie`
`{vladimir.rychkov, alexey.lastovetsky}@ucd.ie`
`http://hcl.ucd.ie`

Abstract. Optimisation of data-parallel scientific applications for modern HPC platforms is challenging in terms of efficient use of heterogeneous hardware and software. It requires partitioning the computations in proportion to the speeds of computing devices. Implementation of data partitioning algorithms based on computation performance models is not trivial. It requires accurate and efficient benchmarking of devices, which may share the same resources but execute different codes, appropriate interpolation methods to predict performance, and mathematical methods to solve the data partitioning problem. In this paper, we present a software framework that addresses these issues and automates the main steps of data partitioning. We demonstrate how it can be used to optimise data-parallel applications for modern heterogeneous HPC platforms.

Keywords: heterogeneous computing, data partitioning, computation performance models, hybrid platforms

1 Introduction

Many scientific applications implement data-parallel algorithms, originally designed for homogeneous HPC platforms. The applications range from linear algebra routines to computer simulations, such as computational fluid dynamics. They are characterised by divisible computational workload, which is directly proportional to the size of data and dependent on data locality. In order to execute data-parallel scientific applications on a highly heterogeneous HPC platforms efficiently, computational workload has to be distributed between computing devices in proportion to their speeds. Our target architecture is a dedicated highly heterogeneous HPC platform, which has a stable performance in time, a complex hierarchy of heterogeneous computing devices, and a heterogeneous software stack. We consider this platform as a hierarchical heterogeneous distributed-memory system, and therefore, apply data partitioning, a method of load balancing widely used for distributed-memory supercomputers. Data partitioning algorithms use performance models of computing devices and computation kernels to distribute workload. In this work, we address the problem of implementation of data partitioning algorithms in data-parallel applications for dedicated heterogeneous platforms.

Data partitioning algorithms based on computation performance models require accurate and efficient performance measurement, implementation of interpolation methods for realistic performance prediction, and formalisation and solution of the data partitioning problem. For static data partitioning, the optimality of the load distribution is critical, while for dynamic data partitioning, the cost-efficiency is equally important. In the first case, the use of exhaustive benchmarks to build very detailed computation performance models in advance is justified. In the second case, only short measurements can be performed, whose inaccuracy has to be compensated by advanced interpolation methods. On modern multicore and hardware-accelerated platforms, special performance measurement techniques and computation performance models are required to take into account resource contention. Despite the active research in the area of data partitioning, there is no software available that would address these challenges. In this paper, we present such a software framework.

Our software framework is designed to help in construction of computation performance models for any data-parallel application with given accuracy and cost-effectiveness. There are two types of models supported: constant and functional. The models can be built either in advance to be used in static data partitioning, or at runtime during dynamic load balancing. The framework provides a range of general-purpose data partitioning algorithms based on computation performance models. The choice of the algorithms is determined by users' applications. In this paper, we demonstrate how they can be used for optimal execution of the parallel matrix multiplication and the Jacobi method. The framework supports a wide range of dedicated heterogeneous platforms consisting of uniprocessors, multicores, hardware accelerators. The framework is extensible. New measurement techniques for new types of hardware can be added. Other computation performance models and data partitioning algorithms can be included.

The rest of this paper is organised as follows. In Section 2, we overview existing data partitioning software. In Section 3, we discuss the main challenges in optimisation of data-parallel applications for heterogeneous platforms, and formulate the features of a framework for data partitioning based on computation performance models. In Section 4, we present the new software framework and describe the use cases, namely, optimisation of heterogeneous parallel matrix multiplication and dynamic load balancing of the Jacobi method.

2 Existing Data Partitioning Software

Matrices and meshes are the most common objects of parallel scientific applications. Since they can be represented as graphs, most data partitioning software implement graph partitioning algorithms. Algorithms implemented in

ParMetis [9], SCOTCH [4], JOSTLE [16] reduce the number of edges between the target subdomains, and hence, minimise the total communication cost of the application. They take into account heterogeneity of the platform by specifying

– weights of the target subdomains, which represent the relative speeds of processors [9], or
– a weighted graph of the platform, which contains information about the speeds of processors and the bandwidths of links [4], [16].

Algorithms implemented in Zoltan [3], PaGrid [1] minimise the execution time of the application using some cost function. The cost function depends on both the graph and the parameters of the heterogeneous platform defined by:

– a description of the hierarchy of processors [3], or
– a weighted graph of the platform, with the speeds of processors and the latencies/bandwidths of links [1].

To distribute computations between the processors, all these graph partitioning libraries use simplistic computation performance models, where the speeds of processors are given by constants (weights). Despite the fact that the result of data partitioning is very sensitive to the weights, the libraries do not provide any methods to find the values that balance the load for given data-parallel application on heterogeneous platform. Application programmers are responsible for building the computation performance models and distributing the load.

Traditionally, the constants characterising the performance of the processors are found as their relative speeds demonstrated during the execution of a serial benchmark code solving locally the core computational task of some given size. This approach is not always accurate and may result in non-optimal partitioning on modern highly heterogeneous platforms as it was demonstrated in [6]. Existing data partitioning software, which is based on this approach, do not take into account memory hierarchy, hierarchy of computing devices, software heterogeneity, optimisations and out-of-core techniques used in software.

For modern heterogeneous platforms, more realistic computation performance models have been proposed along with more elaborate general-purpose model-based data partitioning algorithms to find the optimal load distribution ratios, which can be used as weights in graph partitioning. However, integration of these algorithms into data-parallel applications is not trivial. In the following section, we discuss the main challenges of software implementation of heterogeneous data-parallel applications and define the features of a framework for data partitioning based on computation performance models.

3 Optimisation of Data-Parallel Applications for Heterogeneous Platforms

In this section, we analyse the main challenges application programmers face while optimising data-parallel applications for modern heterogeneous HPC architectures. Given a data-parallel scientific application, originally designed for distributed-memory platforms and implemented with help of MPI, how to execute it efficiently on a heterogeneous platform? The total volume of communications is minimised at the application level (for example, by multilevel graph partitioning in mesh applications [9], or by arrangement of matrix blocks in matrix applications [2]). In the computationally intensive part, the application calls a library of routines, for which the hardware-optimised implementations are available (for example, multi-threaded and GPU solvers). Therefore, in order to execute this application efficiently on the heterogeneous platform, we do not design new hybrid kernels that employ multiple computing devices simultaneously. Instead, we use existing high performance kernels for these devices and distribute the application data unevenly between them, based on the a priori information about their performance.

A general-purpose data partitioning algorithms based on computation performance models proceeds as follows. As input, it requires performance models, which can be constructed either in advance or at run-time. The models approximate the speed of the application on each of the computing devices, and in their turn require empirical information about the real performance. This information can be obtained from the benchmarks that assess the performance of the application on the devices. Therefore, the main challenges the application programmer faces are accurate and efficient performance measurement, construction of computation performance models and implementation of model-based data partitioning algorithms.

Accurate and cost-effective methods of performance measurement are paramount for data partitioning to work in real-life heterogeneous environments. The use of wrong estimates can fully destroy the resulting performance of the application. Performance can be found by benchmarking *a computation kernel*, a serial code performing much less computations but still representative for the entire application [10]. For example, computationally intensive applications often perform the same core computation multiple times in a loop. The benchmark made of one such core computation can be representative of the performance of the whole application and can be used as a kernel. **Timing the computation kernel on heterogeneous devices** may be non-trivial, and therefore, its automation would facilitate development of data-parallel applications, especially on the platforms where special techniques are required for accurate performance measurements, such as multicore and hardware-accelerated platforms.

For example, on multicore platforms, parallel processes interfere with each other through shared memory so that the speed of individual cores cannot be measured independently. In this case, the performance of cores in a group can be measured, when all cores are executing the benchmarks in parallel [18]. Interactions between CPUs and GPUs include data transfers between the host and GPU memory over PCI Express, launching of GPU kernels, and some other operations. Performance measurement techniques for heterogeneous GPU-accelerated systems were studied in [13]. It was concluded that the synchronous approach, when the host CPU core observes the beginning and the end of an operation, is valid for measurement of routines implemented in synchronous libraries, such as

CUBLAS. This technique covers all interactions between devices and does not require any special measurement mechanisms. The performance of out-of-core routines can also be measured from the host CPU core. Incorporation of these **performance measurement techniques** into the data-parallel application add extra complexity; they have to be implemented as routine procedures.

The results of performance measurements are used in computation performance models, which implement different interpolation methods to predict the execution time and speed. On this prediction, heterogeneous data partitioning algorithms will be based. A number of models and algorithms have been proposed. Their choice depends on the data-parallel application and the heterogeneous platform. The software framework for data partitioning has to provide a collection of such models and algorithms. We briefly summarise the applicability of recent work related to data partitioning on heterogeneous multicore and multi-GPU platforms.

When the problems fit the main memory of the processors/devices and the processors/devices execute the same codes for these problems, the absolute speeds do not vary. In this case, data partitioning algorithms based on **constant performance model** (CPM) can be used. In [8], constants representing the sustained performance of the application on CPU/GPU were used to partition data. The constants were found a priori. In [17], a similar constant performance model was proposed, but it was built adaptively, using the history of performance measurements. CPM-based algorithms are cost efficient and do not introduce much complexity into heterogeneous applications. The fundamental assumption of these algorithms is that the absolute speed of processors/devices does not depend on the size of a computational task. However, it becomes less accurate when the partitioning of the problem results in some tasks fitting into different levels of memory hierarchy (i), or when processors/devices switch between different codes to solve the same computational problem (ii).

For the cases (i)-(ii), more elaborate computation performance models and data-partitioning algorithms have been proposed. In [12], the execution time of CPU/GPU was approximated by linear functions of problem size, and an empirical approach to estimate the application-specific **linear performance models** was proposed. A more elaborate analytical predictive model was proposed in [14]. It is also application-specific, however, not only the values of parameters, but also their number and the predictive formulas are defined individually for each application, based on thorough performance analysis of the main steps of the application. In [14], it was admitted that linear models might not fit the actual performance in the case of resource contention (iii), and therefore, they were replaced by **analytical piecewise model**. This model can achieve high accuracy but there is no generic way to build it for an arbitrary application and hardware. Nevertheless, analytical models and model-based data partitioning algorithms can be implemented once per class of applications and can be included into a framework for data partitioning.

For an arbitrary data-parallel application to be executed for a wide range of problem sizes on a platform with highly heterogeneous hardware/software and resource contention, we proposed the **functional performance model** (FPM), where the speed is represented by a function of problem size that is built empirically and integrates performance characteristics of both the architecture and the application [10]. 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* processed per second. Such a performance model is application and hardware 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. This model can be estimated in the same way for any data-parallel application and applicable in situations (i)-(ii). It approximates the execution time and speed using piecewise linear or Akima spline interpolation [15]. Originally, the functional performance model was designed for uniprocessor machines: it provided optimal data partitioning [7] and efficient dynamic load balancing [6] on heterogeneous networks of uniprocessors. Later, this approach was extended to multicore [18] and hybrid CPU/GPU [19] platforms. Taking into account resource contention, situation (iii), we introduced *a speed of multiple cores*, when multiple cores simultaneously execute the same computation kernel, and *a combined speed of a GPU and a dedicated host CPU*, when the GPU executes a computation kernel and the CPU provides memory management. Integration of all these features into a data-parallel application is challenging and requires appropriate software implementation.

Elaborate computation performance models provide more accurate prediction but complicate data partitioning algorithms. In contrast to the traditional data partitioning algorithms, which distribute computations in proportion to constant speeds, the algorithms based on functional models require solving a system of equations whose solution yields the balance. If the speeds are defined by predictive formulas, the solution of the load balancing problem can be found analytically [12], [14]. Otherwise, if the speeds are interpolated from empirical data, like in [10], the solution can be found geometrically [10] or numerically [15]. Implementation of model-based data partitioning from scratch within a heterogeneous data-parallel application is challenging due to **complexity of data partitioning algorithms**. Therefore, routines implementing these algorithms forms a key functionality of a framework facilitating development of applications for heterogeneous platforms.

In this paper, we present the software framework that addresses the above challenges. On several examples, we illustrate how to adapt data-parallel MPI applications to hybrid heterogeneous platforms, using this framework.

4 New Framework for Model-Based Data Partitioning

In this section, we give a high-level outline of the new framework for model-based data partitioning FuPerMod, available through the open-source license from `http://hcl.ucd.ie/project/fupermod`. The framework provides the programming interface for:

– accurate and cost-effective performance measurement,
– construction of computation performance models implementing different methods of interpolation of time and speed,
– invocation of model-based data partitioning algorithms for static and dynamic load balancing.

This functionality can be incorporated into a data-parallel applications as follows. First, the application programmer has to provide the serial code for the computation kernel of their application and define its computation unit by using the API provided. This code will be used for computation performance measurements, which can be carried out either within the application or separately, in order to obtain the a priori performance information. Then, the programmer chooses the appropriate computation performance model and data partitioning algorithm, and incorporates them into the application. Upon execution of the data-parallel application on the heterogeneous platform, the models of processors/devices will be constructed and the data partitioning algorithm will yield the optimal distribution of workload for a given problem size. Finally, the programmer is responsible to distribute the application data accordingly to the optimal distribution, which will be given in computation units.

4.1 Computation Performance Measurement

The programming interface for computation performance measurement consists of a data structure encapsulating the computation kernel, *fupermod_kernel*, the benchmark function, *fupermod_benchmark*, and a data structure storing the result of the measurement, *fupermod_point*.

The serial code of the computation kernel has to be provided together with the functions to allocate and deallocate the data for a problem size given in computation units. In these functions, the application programmer defines the computation unit and reproduces the memory requirements of the application. To enable conversion of speed from units/sec to FLOPS, the programmer has to specify the complexity of the computation unit. As a whole, *fupermod_kernel* has the following interface:

```
struct fupermod_kernel {
  double (*complexity)(int d, void* params);
  int (*initialize)(int d, void* params);
  int (*execute)(pthread_mutex_t* mutex, void* params);
  int (*finalize)(void* params);
};
```

– *complexity* is a pointer to the function that returns the complexity of computing d units;
– *initialize*/*finalize* allocate and deallocate memory for the problem of d computation units (create and destroy the execution context for the kernel);
– *execute* executes the computation kernel in a separate thread;
– *params* stores the execution context of the kernel;

– *mutex* protects some resources, when kernel is terminated during a long run.

Let us consider how to define the computation kernel for a typical data-parallel application, such as matrix multiplication.

In this application, square matrices A, B and C are partitioned over a 2D arrangement of heterogeneous processors so that the area of each rectangle is proportional to the speed of the processor that handles the rectangle. This speed is given by the speed function of the processor for the assigned problem size. Figure 1(a) shows one iteration of matrix multiplication, with the blocking factor b parameter, adjusting the granularity of communications and computations [5]. At each iteration of the main loop, pivot column of matrix A and pivot row of matrix B are broadcasted horizontally and vertically, and then matrix C is updated in parallel by the GEMM routine of the Basic Linear Algebra Subprograms (BLAS). In this application, we use the matrix partitioning algorithm [2] that arranges the submatrices to be as square as possible, minimising the total volume of communications and balancing the computations on the heterogeneous processors.

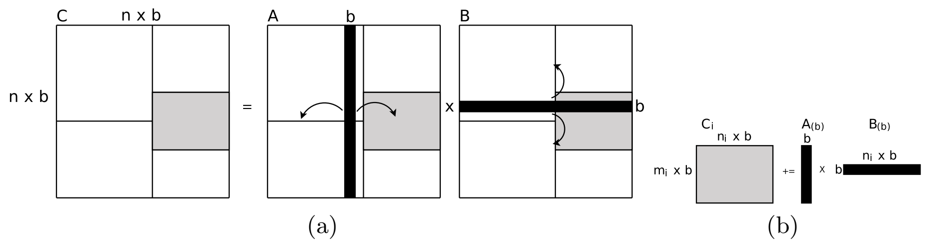


Fig. 1. Heterogeneous parallel column-based matrix multiplication (a) and its computational kernel (b)

We assume that the total execution time of the application can be approximated by multiplying the execution time of a single run of the computational kernel by the number of iterations of the application. Therefore, the speed of the application can be estimated more efficiently by measuring just one run of the kernel. For this application, the computation kernel on the processor i will be an update of a $b \times b$ block of the submatrix C_i with the parts of pivot column $A_{(b)}$ and pivot row $B_{(b)}$: $C_i+ = A_{(b)} \times B_{(b)}$ (Fig. 1(b)). The block update represents one computation unit of the application. The processor i is to process $m_i \times n_i$ such computation units, which is equal to the area of the submatrix if measured in blocks. For nearly-square submatrices, which is the case in this application, one parameter, area, can be used as a problem size. Therefore, in the *initialize* function, for the problem size d_i, we can allocate $\sqrt{d_i} \times \sqrt{d_i}$ blocks for the submatrix C_i and $\sqrt{d_i}$ blocks for the parts of pivot column and row. The *execute* function for this kernel will call the GEMM routine once with these matrices as input. Having the same memory access pattern as the whole application, the kernel will be executed at nearly the same speed as the whole application. The *complexity* function returns the number of arithmetic operations performed by the kernel: $2 \times d_i \times b \times b$.

Performance measurement of this kernel on heterogeneous devices that share resources and use different programming models is challenging. In our previous work, we proposed the measurement techniques for a multicore node [18] or GPU-accelerated node [19], which are now implemented in the FuPerMod framework. They provide reproducible results within some accuracy and can be summarised as follows. Automatic rearranging of the processes provided by operating system may result in performance degradation, therefore, we bind processes to cores to ensure a stable performance. Then, we synchronise the processes that share resources (on a node or a socket), in order to minimise the idle computational cycles, aiming at the highest floating point rate for the application. Synchronisation also ensures that the resources will be shared between the maximum number of processes, generating the highest memory traffic. To ensure the reliability of the measurement, experiments are repeated multiple times until the results are statistically correct.

GPU depends on a host process, which handles data transfer between the host and device and launches kernels on the device. A CPU core is usually dedicated to deal with the GPU, and can undertake partial computations simultaneously with the GPU. Therefore, we measure the combined performance of the dedicated core and GPU, including the overhead incurred by data transfer between them. Due to limited GPU memory, the execution time of GPU kernels can be measured only within some range of problem sizes, unless out-of-core implementations, which address this limitation, are available.

To measure the performance of a computation kernel on heterogeneous processors/devices, FuPerMod provides a function *fupermod_benchmark*, which has the following interface:

```
int fupermod_benchmark(
  fupermod_kernel* kernel, int d,
  fupermod_precision precision,
  MPI_Comm comm_sync,
  fupermod_point* point
);
```

```
struct fupermod_point {
  int d;
  double t;
  int reps;
  double ci;
};
```

This function initialises the *kernel* for the problem size *d* and executes it multiple times accordingly to the *precision* argument, which defines the number of repetitions and statistical parameters. The kernel can be executed in multiple processes. MPI communicator *comm_sync* is used to synchronise the processes running on a multi-CPU/GPU node. The function returns a *point*, which contains the results of the measurement: the problem size in computation units, *d*; the measured execution time, *t*; the number of repetitions the measurement has actually taken, *reps*; and the confidence interval of the measurement, *ci*. Arrays of these experimental points are then used to model the performance of CPU core(s), or the bundled performance of a GPU and its dedicated CPU core, or the total performance of a multi-CPU/GPU node.

4.2 Computation Performance Models

The key abstraction of the programming interface for computation performance modeling is *fupermod_model*, which has the following interface:

```
struct fupermod_model {
  int count;
  fupermod_point* points;
  double (*t)(fupermod_model* model, double x);
  int (*update)(fupermod_model* model, fupermod_point point);
};
```

It encapsulates experimental points obtained from measurements, which are given by the *count* and *points* data fields, and the approximation of the time function, *t*. *update* specifies how the approximation changes after adding a new experimental point. The speed in FLOPS is evaluated using the approximated time and the complexity of the computation kernel: $s(x) = complexity(x)/time(x)$, where x is a problem size given in computation units. These approximations are used in the model-based data partitioning algorithms to predict the computation performance and distribute the workload proportionally.

Currently, FuPerMod implements the following performance models:

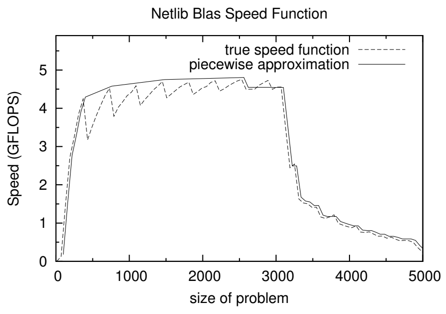
- CPM (requires only one experimental point);
- FPM based on the piecewise linear interpolation of the time function;
- FPM based on the Akima spline interpolation of the time function.

The first FPM is based on some assumptions on the shape of the speed function [10]. In addition to the piecewise linear interpolation, it coarsens the real performance data in order to satisfy those assumptions, as shown in Fig. 2(a). The FPM based on the Akima spline interpolation removes these restrictions [15], and therefore, represents the speed of the processor with more accurate continuous functions (Fig. 2(b)). The *fupermod_model* data structure can be used to implement other computation performance models, for example, application-specific analytical models, such as [14].

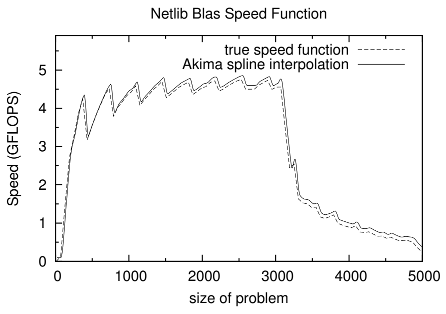
4.3 Static Data Partitioning

Computation performance models of processors/devices are used as input for model-based data partitioning algorithms. The FuPerMod framework currently provides the following algorithms:

- basic algorithm based on CPMs;
- geometrical algorithm based on the piecewise-linear FPMs;
- numerical algorithm based on the Akima-spline FPMs.



(a)



(b)

Fig. 2. Speed functions of the matrix multiplication kernel based on the Netlib BLAS GEMM: (a) piecewise linear interpolation, (b) Akima spline interpolation

The CPM-based algorithm divides the data in proportion to the constant speeds. This is the fastest but least accurate data partitioning algorithm. It is appropriate for the cases when it has been observed that the speeds do not vary much. The geometrical algorithm implements iterative bisection of the speed functions with lines passing through the origin of the coordinate system [10]. Convergence of this algorithm is ensured by putting restrictions on the shape of the speed functions, which is implemented in the piecewise-linear FPMs. The numerical algorithm applies multidimensional solvers to numerical solution of the system of non-linear equations that formalise the problem of optimal data partitioning [15]. It can be applied to smooth speed functions of any shape. As input, the algorithm takes the Akima-spline FPMs, since this approximation provides continuous derivative.

Data partitioning algorithms have the following interface:

```
typedef int (*fupermod_partition)(
  int size, fupermod_model** models, fupermod_dist* dist);
```

where *size* is the number of the processors/devices, *models* is an array of the models corresponding to the processors/devices, and *dist* is the distribution of data. The distribution is an input/output argument and has the following structure:

```
struct fupermod_dist {
  int D;
  int size;
  fupermod_part* parts;
};
```

```
struct fupermod_part {
  int d;
  double t;
};
```

where D is the total problem size to partition (in computation units); *size* is the number of processors/devices; *parts* is the array specifying the workload d that will be assigned to the processors/devices, and the approximated processing time t of the workload. After execution of the data partitioning algorithm, the application programmer distributes the workload in accordance with the *dist* argument. A sample code demonstrating how to use the programming interface for data partitioning will be provided below, within a more practical example of dynamic load balancing.

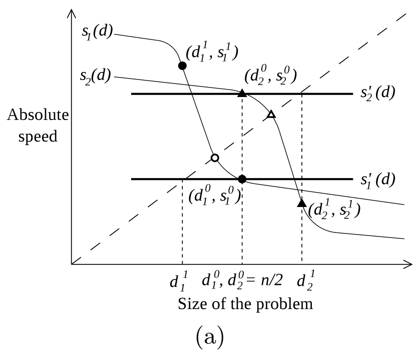
The cost of experimentally building a full computation performance model, i.e. a functional model for the full range of problem sizes, may be very high, which limits the applicability of the above partitioning algorithms to situations where the construction of the models and their use in the application can be separated. For example, if we develop an application that will be executed on the same platform multiple times, we can build the full models once and then use these models multiple times during the repeated execution of the application. In this case, the time of construction of the models can become very small compared to the accumulated performance gains during the multiple executions of the optimized application. Building full functional performance models is not suitable for an application that is run a small number of times on a platform. In this case, computations should be optimally distributed between processors without *a priori* information about execution characteristics of the application running on the platform. In the following section, we describe the programming interface for dynamic data partitioning and load balancing, which can be used to design applications that automatically adapt at runtime to any set of heterogeneous processors.

4.4 Dynamic Data Partitioning and Load Balancing

FuPerMod provides the efficient data partitioning algorithms that do not require performance models as input. Instead, they approximate the speeds around the relevant problem sizes, for which performance measurements are made during the execution of the algorithms. These algorithms do not construct complete performance models, but rather partially estimate them, sufficiently for optimal distribution of computations. They balance the load not perfectly, with a given accuracy. The low execution cost of these algorithms makes them suitable for employment in self-adaptable applications. Currently, FuPerMod provides two such algorithms, designed for dynamic data partitioning [11] and dynamic load balancing [6].

The dynamic algorithms perform data partitioning iteratively, using the partial estimates instead of the full computation performance models. At each iteration, the solution of the data partitioning problem gives new relevant problem sizes. The performance is measured for these problem sizes, and the partial estimates are refined. In the case of dynamic data partitioning, the measurements are made by benchmarking the representative computation kernel of the application. In the case of dynamic load balancing, the real execution of one iteration of the application is timed. Figure 3 shows a few steps of dynamic data partitioning for piecewise linear FPMs and geometrical data partitioning algorithm.

The programming interface for the dynamic algorithms consists of a data structure *fupermod_dynamic*, specifying the context of their execution, and two



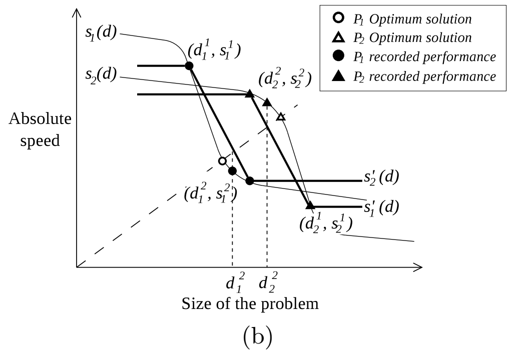


Fig. 3. Construction of the partial FPMs based on piecewise linear interpolation, using the geometrical data partitioning algorithm

functions *fupermod_partition_iterate* and *fupermod_balance_iterate*, implementing one step of dynamic partitioning and load balancing respectively:

```
struct fupermod_dynamic {
  fupermod_partition partition;
  int size;
  fupermod_model** models;
  fupermod_dist* dist;
}
int fupermod_partition_iterate(fupermod_dynamic*, MPI_Comm comm,
  fupermod_precision precision, fupermod_benchmark* benchmark,
  double eps);
int fupermod_balance_iterate(fupermod_dynamic*, MPI_Comm comm,
  struct timespec start);
```

The context includes the pointer to a data partitioning algorithm, *partition*, current partial estimates, *models*, and near-optimal data partition, *dist*. Both function invoke the data partitioning algorithm once, using the current estimates, and store the result in *dist*. The dynamic data partitioning function performs the *benchmark*, with the statistical parameters *precision*, while the dynamic load balancing function uses the *start* time of the current iteration of the application to time. Then both function update the partial estimates. The dynamic data partitioning also requires the accuracy, *error*, as a termination criterion.

In conclusion, we demonstrate how to use this API for optimisation of another data-parallel application, which implements the Jacobi method. This application distributes the matrix and vectors by rows between the processors and iteratively solves the system of equations. In the source code below, the partial FPMs based on piecewise linear interpolation are constructed at runtime during the iterations of the Jacobi method. At each iteration, the load balancing function invokes the geometrical data partitioning algorithm. The system of equations is redistributed accordingly to the newly obtained data distribution. Figure 4 demonstrates that after several iterations of the application, the load is balanced.

```
MPI_Comm_size(comm, &size);
// FPMs based on piecewise linear interpolation
fupermod_model** models = malloc(sizeof(fupermod_model*) * size);
for (i = 0; i < size; i++)
  models[i] = fupermod_model_piecewise_alloc();
// context for dynamic load balancing
fupermod_dynamic balancer = { fupermod_partition_geometric,
  size, models, fupermod_dist_alloc(D, size) };
// current distribution, initially even
fupermod_dist* dist = fupermod_dist_alloc(D, size);
// Jacobi data: dist->parts[i].d rows of matrix and vectors
double *A, *b, *x; // allocation, initialisation
// main loop
double error = DBL_MAX;
while (error > eps) {
  // redistribution of Jacobi data accordingly to balancer.dist
  jacobi_redistribute(comm, dist, A, b, x, balancer.dist);
  // store the current distribution
  fupermod_dist_copy(dist, balancer.dist);
  struct timeval start;
  gettimeofday(&start, NULL);
  // Jacobi iteration
  jacobi_iterate(comm, dist, A, b, x, &error);
  // load balancing with the (dist->parts[i].d, now-start) point
  fupermod_balance_iterate(&balancer, comm, start);
}
```

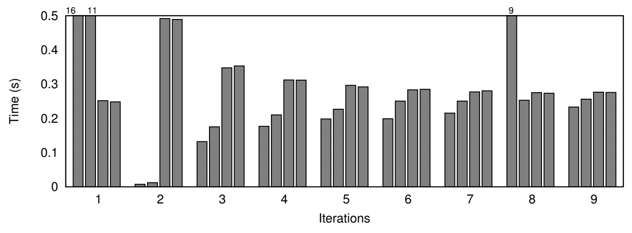


Fig. 4. Dynamic load balancing of Jacobi method with geometrical data partitioning

In this paper, we presented a framework for general-purpose data partitioning based on computation performance models. This framework provides a range of algorithms and models for optimisation of different data-parallel scientific applications on modern heterogeneous platforms.

Acknowledgment. This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/IN.1/I2054. Experiments were carried out on Grid'5000 developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

References

1. Aubanel, E., Wu, X.: Incorporating latency in heterogeneous graph partitioning. In: IPDPS 2007. pp. 1–8 (2007)
2. Beaumont, O., Boudet, V., Rastello, F., Robert, Y.: Matrix multiplication on heterogeneous platforms. IEEE Trans. Parallel Distrib. Syst. 12(10), 1033–1051 (2001)
3. Catalyurek, U., Boman, E., Devine, K., et al.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: IPDPS 2007. pp. 1 –11 (2007)
4. Chevalier, C., Pellegrini, F.: PT-Scotch: A tool for efficient parallel graph ordering. Parallel Computing 34(68), 318 – 331 (2008)
5. Choi, J.: A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In: HPC Asia '97. pp. 224 –229 (1997)
6. Clarke, D., Lastovetsky, A., Rychkov, V.: Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms. Parallel Processing Letters 21, 195–217 (2011)
7. Clarke, D., Lastovetsky, A., Rychkov, V.: Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models. In: HeteroPar'2011. pp. 450–459 (2012)
8. Fatica, M.: Accelerating Linpack with CUDA on heterogenous clusters. In: GPGPU-2. pp. 46–51. ACM (2009)
9. Karypis, G., Schloegel, K.: ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version 4.0 (2013), `http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf`
10. Lastovetsky, A., Reddy, R.: Data partitioning with a functional performance model of heterogeneous processors. Int J High Perform C 21, 76–90 (2007)
11. Lastovetsky, A., Reddy, R.: Distributed data partitioning for heterogeneous processors based on partial estimation of their functional performance models. In: Euro-Par 2009, LNCS, vol. 6043, pp. 91–101. Springer (2010)
12. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: MICRO-42. pp. 45–55 (2009)
13. Malony, A.D., Biersdorff, S., Shende, S., et al.: Parallel performance measurement of heterogeneous parallel systems with GPUs. In: ICPP '11. pp. 176–185 (2011)
14. Ogata, Y., Endo, T., Maruyama, N., Matsuoka, S.: An efficient, model-based CPU-GPU heterogeneous FFT library. In: IPDPS 2008. pp. 1 –10 (2008)
15. Rychkov, V., Clarke, D., Lastovetsky, A.: Using multidimensional solvers for optimal data partitioning on dedicated heterogeneous HPC platforms. In: PaCT-2011, LNCS, vol. 6873, pp. 332–346. Springer (2011)
16. Walshaw, C., Cross, M.: Multilevel mesh partitioning for heterogeneous communication networks. Future Generation Comput. Syst. 17(5), 601–623 (2001)
17. Yang, C., Wang, F., Du, Y., et al.: Adaptive optimization for petascale heterogeneous CPU/GPU computing. In: Cluster'10. pp. 19–28 (2010)
18. Zhong, Z., Rychkov, V., Lastovetsky, A.: Data partitioning on heterogeneous multicore platforms. In: Cluster 2011. pp. 580–584 (2011)
19. Zhong, Z., Rychkov, V., Lastovetsky, A.: Data partitioning on heterogeneous multicore and multi-GPU systems using functional performance models of data-parallel applications. In: Cluster 2012. pp. 191–199 (2012)