Advances in Engineering Software 42 (2011) 247–258

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Reliable performance prediction for multigrid software on distributed memory systems

Giuseppe Romanazzi [a,*], Peter K. Jimack [b], Christopher E. Goodyer [b]

[a] CMUC, Departamento de Matemática, Universidade de Coimbra, Coimbra 3001-454, Portugal
[b] School of Computing, University of Leeds, Leeds LS2 9JT, UK

ARTICLE INFO

Article history:
Available online 20 November 2010

Keywords:
Performance prediction
Parallel engineering software
Multigrid algorithms
Partial differential equations
Distributed memory architectures
Parallel distributed algorithms

ABSTRACT

We propose a model for describing and predicting the parallel performance of a broad class of parallel numerical software on distributed memory architectures. The purpose of this model is to allow reliable predictions to be made for the performance of the software on large numbers of processors of a given parallel system, by only benchmarking the code on small numbers of processors. Having described the methods used, and emphasized the simplicity of their implementation, the approach is tested on a range of engineering software applications that are built upon the use of multigrid algorithms. Despite their simplicity, the models are demonstrated to provide both accurate and robust predictions across a range of different parallel architectures, partitioning strategies and multigrid codes. In particular, the effectiveness of the predictive methodology is shown for a practical engineering software implementation of an elastohydrodynamic lubrication solver.

1. Introduction

Parallel computational engineering software is now becoming widely used for the simulation of a broad range of problems across a significant number of application domains. For the computational engineer there are a number of advantages associated with the use of parallel computer systems, including faster job turnaround and the ability to run simulations that would not otherwise be possible (due to their memory requirements for example). In practice however, this computational work is not undertaken in isolation and so there is a need to map computational tasks onto available resources, schedule tasks across resources, prioritize tasks within a prescribed budget, etc. – often in competition with other users who wish to access some or all of these resources. The work described in this paper is motivated by these constraints and the resulting need for both the computational scientists and the automated resource schedulers to be able to make accurate predictions as to the execution time of a given job on a given computational resource. If such information can be obtained cheaply and reliably then informed decisions concerning the trade-off between extra computational resolution versus machine availability, turn-around time, charges, etc., can be made with confidence. Similarly, scarce resources can be scheduled and assigned with greater efficiency.

In order to keep the task tractable, we focus our attention here on parallel engineering software that is based upon use of multigrid methods [4,5,33]. Multigrid is a computational technique that is growing in importance across a very wide selection of engineering applications, from computational fluid dynamics [9,20], through to phase change problems [16,28], for example. The idea behind the technique is to accelerate the iterative solution of a discrete set of algebraic equations on a very fine finite difference/element/volume mesh by taking some of the iterations, on a suitably modified problem, on a coarser mesh. These iterations can, in turn, be accelerated by iterations on a mesh that is coarser still, repeating the process until only a very coarse mesh is used at the most basic level. The success of this approach is based upon choosing the iterative method on each mesh to satisfy the so-called *smoothing property*, which ensures that the error components of the highest frequency that may be represented on each mesh are eliminated the fastest. See [4,5,33] for further details.

The parallel implementation of multigrid software has been the topic of much recent research, e.g. [3,11–13,19]. This is typically based upon using a geometric partition of the coarsest mesh that is present and then mapping this in the natural manner to each of the finer meshes. Such an approach is also taken throughout this work. Of course there are many ways in which this geometric partition may be undertaken [20], however for the majority of this paper we restrict our attention to the use of a so-called strip partition, which is described more fully in Section 3 below. This is primarily because the practical engineering software [12,13], that we use as the most challenging test case in this paper is written with this

* Corresponding author.
E-mail address: roman@mat.uc.pt (G. Romanazzi).

0965-9978/$ - see front matter © 2010 Civil-Comp Ltd and Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2010.10.005

partitioning strategy. In the final sections of the paper we also discuss how our results extend to other partitioning strategies.

All of the computational domains considered in this work are in two dimensions, although extensions to problems in three dimensions are briefly discussed towards the end of the paper. Furthermore, the domains that we consider are typically rectangular in shape. It is important to emphasize however that use of a rectangular domain does not necessarily mean that the problems being solved have such a simple geometry. For example, the elastohydrodynamic lubrication software that is described in Section 6, uses a lubrication approximation to allow contact problems over a range of complex geometries, including flows for which cavitation occurs, to be represented [12,13]. Similar long-wave approximations are widely used in thin film flow simulations to allow all sorts of geometric complexity to be modelled on simple rectangular domains in two dimensions, e.g. [9,10]. Other techniques which allow geometry to be represented implicitly on simple domains include phase-field, level-set and volume-of-fluid methods (see, for example [28,1,14] respectively).

As already indicated above, the main contribution of this work is to propose, and assess, a model for describing and predicting the performance of parallel multigrid software running on a distributed memory architecture. The goal of the model is to allow reliable predictions to be made as to the execution time of a given code on a large number of processors, of a given parallel system, by only benchmarking the code on small numbers of processors. We show that the prediction model is accurate and robust with respect to both the processor and the communication architectures considered. In particular we consider heterogeneous and multicore processor architectures, combined with different communication architectures, such as Myrinet and Fast Ethernet switching. Furthermore, unlike our previously published work [23–25,27], we demonstrate the effectiveness of our approach on practical engineering software [12,13], in addition to codes that are based upon slightly less complex mathematical models [11].

The layout of the paper is as follows. In the next section we describe some related work into performance modelling. In Section 3, we provide a very brief introduction to parallel multigrid algorithms for the solution of elliptic or parabolic partial differential equations (PDEs) in two space dimensions. This is followed by an analysis of the typical multigrid performance on an abstract distributed memory architecture. This analysis is then used to build a predictive model, for this class of code, that is designed to allow estimates of run times to be obtained for large numbers of processors, based upon observed performance on very small numbers of processors. In Section 4 we describe in detail our initial implementation of the predictive methodology [24], and show some prediction results for benchmark codes across different parallel architectures. These results show the great potential of our strategy but also illustrate some limitations of this initial implementation, which are discussed. In Section 5 we therefore generalize the implementation of our predictive model and illustrate that this delivers both accurate and robust predictions across all of our test problems. In Section 6, we then describe how the general predictive methodology that has been presented may be applied to a practical parallel multilevel code that is used to model elastohydrodynamic lubrication (EHL) [12,13]. The paper concludes with two further sections, which discuss additional generalizations and applicability of our work.

2. Related work

Research into performance modelling and prediction for parallel and distributed software and systems is by no means new. Indeed, an excellent recent survey of the state-of-the-art in this area is provided in [21]. It makes no sense to attempt to repeat such a survey here, however we do present a very brief overview of some of the main techniques that have been used, in order to allow the novelty of the approach that we propose to be highlighted. In particular, two broad classes of methods may be identified: one of which is based upon the use of analytical models which provide a detailed description of an individual application; whilst the other is based upon obtaining a detailed description of the target high performance computing (HPC) systems that may be used to execute a code (based upon carefully selected benchmark tests), and then combining this with the output of an application trace for the software in question.

Examples of the former approach include a the LogP approach of [7] or, more recently, the work of [17]. In [17], for example, the focus is on a particular large-scale parallel application and so it is deemed worthwhile to invest the effort required to build a detailed analytic model of this application for a selection of possible HPC resources. The advantage of this approach is the very accurate predictions that are possible, and are delivered consistently for the applications and HPC systems that are considered. The main drawback however is that substantial code-specific knowledge is required, and considerable person effort is needed each time a new HPC resource becomes available for consideration.

The alternative class of method tends to be rather more empirical, but is also more flexible and portable than the analytic approaches. For example, [6,22] present techniques for performance prediction in which user knowledge of the code is replaced by the need to execute a detailed trace of the software. This trace identifies the main communication and computation activities, and is then combined with the output from thorough benchmarks of the HPC systems being considered. The main advantages of this approach are the lack of a need to have a highly detailed knowledge of the software being considered, and the resulting relative ease with which new software may be assessed. However, this trace-based approach typically requires significant computational effort and could lead to the requirement for new benchmarks to be obtained for each HPC system. This could be a significant disadvantage.

In the approach that we present in the following sections, we aim to develop a compromise between these two different classes of method. This paper represents the development, and maturing, of our initial efforts to find such a middle ground, which have appeared in [23–27]. The goal is to build predictions based upon the empirical approach, in order to allow a wide variety of codes to be incorporated with relative ease, but to avoid the need for detailed and expensive benchmarking of every target HPC architecture, or for expensive traces to be executed for each parallel code being considered. This is achieved by running the target software on each target architecture using different, but always small, numbers of processors. Empirical models that allow us to extrapolate the timings obtained, to much larger numbers of processors, are then parameterized and applied.

In addition to the development of performance modelling to allow individual engineering practitioners to make informed decisions concerning mesh resolution, resource selection, etc. (in terms of availability, turn-around time and cost), the use of reliable models is also of great practical value in the scheduling of resources on computational Grids. The research in [29], for example, addresses this particular issue. There, the execution time for each job is broken into two parts, representing computation and communication costs, that are subsequently estimated. This is the same general approach that is used in our work. Finally, we note that techniques have also been developed, involving stochastic models, to predict how an application may behave when there is contention for resources, e.g. [30]. We do not consider this here however.

To our knowledge, no previous research on parallel performance modelling and prediction has focused on multigrid-based software. As described in the introduction, this is an extremely important class of numerical algorithm which, due to its computational efficiency, is becoming more and more widely used in practical engineering software. Illustrative work includes [9,11–13,16,19,20,28], but there are many more examples that could be cited. The original multigrid approach, e.g. [4], was designed for the solution of linear problems however the technique may also be applied directly to nonlinear problems via the so-called full approximation scheme (FAS), see [33] for details. In this paper both linear and nonlinear (FAS) multigrid are considered. In both cases we are able to exploit one of the fundamental properties of multigrid, which is that the computational cost of the standard sequential algorithm grows only linearly with the problem size (i.e. the number of unknowns used in the discretization of the underlying PDE problem), [5,33]. This is illustrated in the following section, see Fig. 2, and allows us to assume that if the number of parallel processors is increased in proportion to the size of the finest mesh used in the discretization, then the computational work per processor will remain approximately constant. The next section begins with a brief discussion of multigrid, and its parallel implementation, before providing further detail of the predictive methodology that we have adopted.

3. Parallel multigrid implementations and predictive methodology

3.1. Parallel multigrid software

The general principal upon which multigrid is based is that, when using a suitable iterative solver (i.e. with the smoothing property) for the system of algebraic equations that results from the discretization of a PDE of interest, the component of the error that is damped most quickly is the highest frequency part [5,33]. This observation leads to the development of an algorithm which takes a very small number of iterations on the finest grid upon which the discretized PDE solution is sought, and then restricts the residual and equations to a coarser grid, to solve for an estimate of the error on this grid. This error is then interpolated back onto the original grid before a small number of further iterations are taken and the process repeated. When the error equation is itself solved in the same manner, using a still coarser grid, and these corrections are repeated recursively down to a very coarse base grid, the resulting process is known as multigrid.

Any parallel software that is based upon the use of multigrid requires a number of components to be implemented in parallel. Each of the following components will require some element of inter-processor communication in order to achieve this implementation:

- application of the iterative solver at each grid level;
- exact solution at the coarsest level;
- a convergence test;
- restriction of the residual to a coarser level;
- interpolation of the error to a finer level.

In addition, in the FAS variant of the algorithm it is also necessary that the solution (as well as the residual) should be restricted to the coarser grid at each level [33].

For the first part of this work we consider the parallel implementation of two multigrid codes: one is linear and the other uses the FAS approach. In both cases they are based upon finite difference (FD) discretizations and they partition the two-dimensional FD grid across a set of parallel processors by assigning blocks of rows to different processors (a so-called strip partition). Note that when the coarsest mesh is partitioned in this manner, then if all finer meshes are uniform refinements of this, they are automatically partitioned too: see Fig. 1 for an illustration.

It is clear from inspection of the meshes in Fig. 1 that each stage of the parallel multigrid process requires communication between neighbouring processes (iteration, restriction, coarse grid solution, interpolation, convergence test). The precise way in which these are implemented will vary from code to code, however the basic structure of the algorithm will remain the same. In this work we consider two different implementations, referred to as $m1$ and $m2$, which are described in the following paragraphs. For a full discussion of these multigrid codes see also our initial work on this problem, [23]. Also, note from Fig. 2 that, as already highlighted in the previous section, the computational time scales linearly with the size of the problem for each of these codes (when run in sequential mode).

3.1.1. The algorithm m1

This algorithm solves the steady-state equation

$$-\nabla^2 u = f \text{ in } \Omega,$$
$$\Omega = [0,1] \times [0,1],$$
$$u_{|\partial\Omega} = 0.$$

The discretization is based upon a central finite difference scheme on each grid. The iterative solver employed is the well-known Red–Black Gauss-Seidel (RBGS) method [18], which is ideally suited to parallel implementation. The partitioning of the grids is based upon Fig. 1 with both processors p and $p+1$ owning the row of unknowns on the top of block p. This means that communications between processors are only required in the restriction

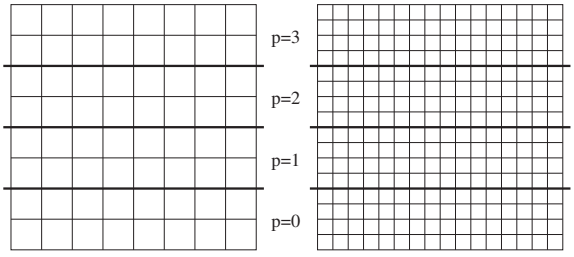

Fig. 1. Strip partitioning of a coarse and a fine mesh across four processors by assigning a block of rows to each processor.

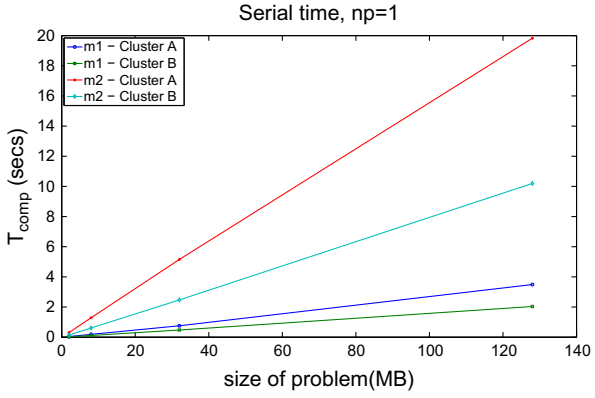

Fig. 2. Computational multigrid time for combinations of two codes ($m1$,$m2$) running on two different Clusters (A,B): in each case the solution time scales linearly with the problem size (as represented by the memory requirement for each run).

phase and after each red and black sweep of RBGS. The code requires then some extra, dummy rows which are updated after each communication. This code uses a series of non-blocking sends and receives in MPI [31] during the communication phase.

3.1.2. The algorithm m2

This algorithm, described in more detail in [18], uses an unconditionally-stable implicit time-stepping scheme to solve the transient problem

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \text{ in } (0,T] \times \Omega,$$
$$\Omega = [0,1] \times [0,1],$$
$$u_{|\partial\Omega} = 0,$$
$$u_{|t=0} = u_0.$$

As for $m1$, the discretization of the Laplacian is based upon the standard five point finite difference stencil. Hence, at each time step it is necessary to solve an algebraic system of equations for which multigrid is used. Again RBGS is selected as the iterative scheme and so communications are required between neighbour processors after each red and black sweep. Different from m1, here only processor p owns the row of unknowns at the top of block p. This means that communications are also required at the interpolation phase, as well as the restriction and convergence test phases. Also different from m1, the inter-processor sends and receives are based upon a mixture of MPI blocking and non-blocking functions. Finally, as mentioned above, m2 is implemented using the FAS algorithm [33] and so the current solution must also be restricted from the fine to the coarser grid at each level.

3.2. Parallel architectures

The model, described in the following sections, is used to predict the performance of multigrid numerical codes, running on two different clusters of the White Rose Grid [8] environment:

- Cluster A (White Rose Grid Node 2) is an heterogeneous cluster of 128 dual processor nodes, each based around 2.2 or 2.4 GHz Intel Xeon processors with 2 GBytes of memory and 512 KB of L2 cache. Myrinet or Fast Ethernet switching are used in the tests to connect the nodes.
- Cluster B (White Rose Grid Node 3) is a multicore cluster of 87 Sun microsystem dual processor AMD nodes, each formed by two dual core 2.0 GHz processors. Each of the $87 \times 4 = 348$ batched processors has L2 cache memory of size 512 KB and access to 8 Gb of physical memory (but only 8 Gb in total, even when all 4 cores are active on a given node). Again, either Myrinet or Fast Ethernet switching may be used.

3.3. Basic predictive methodology

The goal of most parallel numerical implementations is to be able to solve larger problems than would be otherwise possible. For the numerical solution of PDEs this means solving problems on finer grids, so as to be able to achieve higher accuracy. Ideally, when increasing the size of a problem by a factor of np and solving it using np processors (instead of a single processor), the solution time should be unchanged. This would represent a perfect efficiency and is rarely achieved due to the parallel overheads such as inter-processor communications and any computations that are repeated on more than one processor. In this research our primary goal is to be able to predict these overheads. We aim to achieve this by running across a small number of processors, with the size of the problem (the number of discrete unknowns on the finest mesh) scaled in proportion to the number of processors used, as illustrated in Fig. 3.

The predictive model seeks to forecast the execution time of parallel runs of different multigrid codes when used to solve a large "target" problem using np processors on a given HPC architecture. This target problem is defined on a square computational mesh of dimension $N \times N$, with an homogeneous "strip" distribution of the computational domain across the available np processors, see Fig. 1.

In quantitative terms, the basic assumption that we make is that the parallel solution time (on np processors) may be represented as

$$T = T_{comp} + T_{comm}. \quad (1)$$

In (1), T_{comp} represents the computational time for a problem of size $N \times \tilde{N}$ on a single processor (where $\tilde{N} = N/np$), and T_{comm} represents all of the parallel overheads (primarily due to inter-processor communications).

The calculation of T_{comp} is straightforward since this simply requires the execution of a problem of size $N \times \tilde{N}$ on a single processor. Note that it is important that the precise dimensions of the problem solved on each processor in the parallel implementation are maintained for the sequential solve in order to obtain an accurate value for T_{comp}. This is because the memory access and contention patterns observed in the parallel runs (such as cache and multicore effects at the node-level) vary with respect to the geometrical dimensions of the memory allocated to each processor, and they can consequently influence the computational time measured.

The more challenging task is to model T_{comm} in a manner that will allow predictions to be made for large values of np. Recall that our goal is to develop a *simple* model that will capture the main features of this class of numerical algorithm with just a small number of parameters that may be computed based upon runs using only a few processors. The overhead term T_{comm} used in the model (1) depends on a variety of factors, such as the processor, the communication architecture, and, of course, the parallel implementation of the code. We seek to capture this term based upon simulations of the target problem running on a smaller number

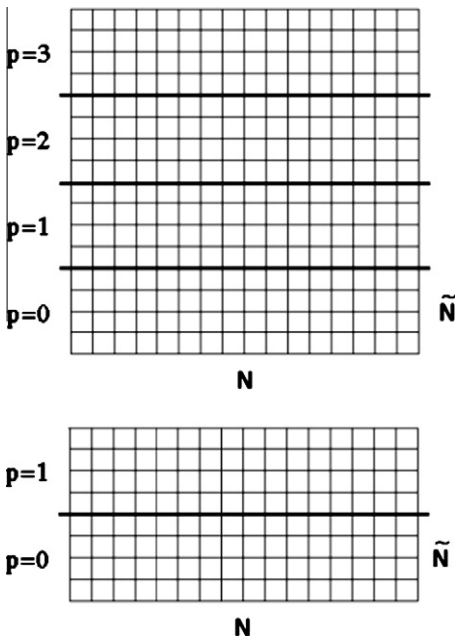

Fig. 3. Partitioning of a square mesh across four processors (top) and the equivalent problem considered on two processors (bottom).

of processors, but using the same length of messages in the send and receive operations as for the target problem. Keeping this in mind we describe the predictive methodology for the parallel overheads in the following sections.

4. Linear predictive methodology for the parallel overheads

A preliminary methodology for predicting the parallel overheads, i.e. T_{comm} in (1), is described here. It is based on the model presented in [25]. For convenience, we define as "work per processor" the memory required by each processor: this is because the computational work per processor in a multigrid code is proportional to the problem size assigned, and therefore to the associated memory required by each processor. Moreover, we use the term "processor" to refer to a single core-processor in the parallel architecture considered.

Figs. 4 and 5 show respectively plots of the actual overhead observed for the multigrid code ($m1$) against the computational work for two different HPC systems: one based upon Fast Ethernet switching, and the other based upon Myrinet. In the associated runs we have kept fixed the first dimension of the problem (i.e. N) and have varied the work per processor. Note that this means that each processor has to solve a problem size of dimension $N \times \widetilde{N}$ where $\widetilde{N}$ is allowed to vary.

Based upon the empirical evidence of the plots shown in Figs. 4 and 5, and similar tests, we can estimate T_{comm} using the following model

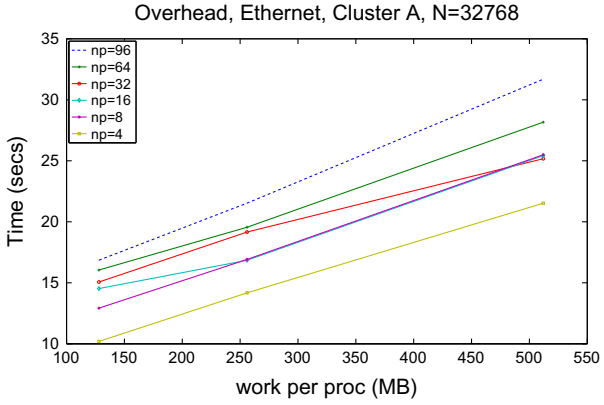

Fig. 4. Overhead (T_{comm}) associated to a fixed size of messages (N) using Fast Ethernet switching and code $m1$.

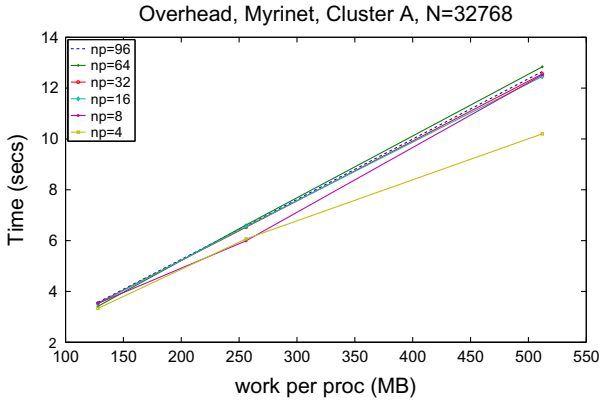

Fig. 5. Overhead (T_{comm}) associated to a fixed size of messages (N) using Myrinet switching and code $m1$.

$$T_{comm} = \alpha(np) + \gamma(np) \cdot work. \tag{2}$$

The length of the messages (which are always of length N on the finest mesh) does not appear in this formula since it is assumed that for a given size of target problem (e.g. a mesh of dimension 32,768 × 32,768) the size of the messages is known *a priori* (in this case, since the partition is by rows, the largest messages will be of length 32,768). Hence there is no need to include N in the model as it is fixed in advance. This is the primary reason that the expression (2) can be so simple.

Furthermore, the following simple relations are assumed to hold:

$$\alpha(np) \approx c + d\log_2(np) \tag{3}$$

$$\gamma(np) \approx \text{constant}. \tag{4}$$

These are based on the observation that, as illustrated in Figs. 4 and 5, in each case we have an almost linear growth in overhead with work, where the slope is approximately constant, and there is an almost constant difference between graphs as np is doubled. Note that the length of the messages is the same in all of these runs (see Fig. 3 for constant work with two different choices of np and Fig. 6 for the same np but half the work per processor).

In order to be able to use the model (2) it is necessary to evaluate the parameters c,d and γ appearing in (3) and (4). In our methodology, these parameters are determined using measurements taken for $np = 4$ and $np = 8$ with $\gamma = \gamma(8)$, and c and d obtained using a simple linear fit through the two data points.

4.1. Algorithm to compute estimated execution time

A summary of the overall predictive methodology is provided by the following steps. We define as $N \times N$ and np the target problem size and number of processors respectively (i.e. we wish to predict a code's performance for these values). Also, let $\widetilde{N} = N/np$ and define $N \times \widetilde{N}$ to be the size of problem on each processor in the target configuration.

(1) Run the code on a single processor with a fine grid of dimension $N \times \widetilde{N}$ and then with dimension $N \times \frac{\widetilde{N}}{2}$ and $N \times \frac{\widetilde{N}}{4}$. In each case collect the computational time T_{comp} and define as *work* the memory allocated by the processor.

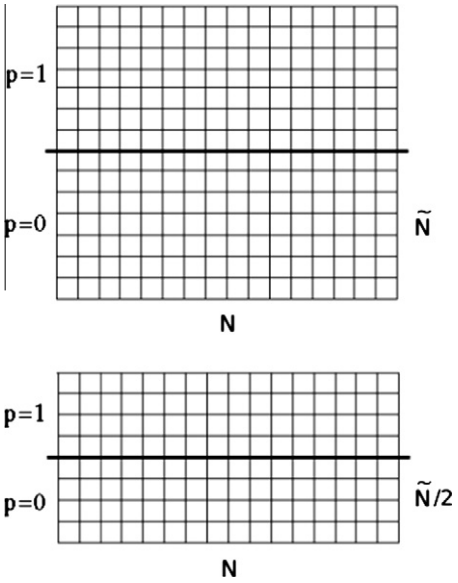

Fig. 6. Scaling the work per processor whilst maintaining the communication volume.

(2) Run the code on $np0 = 4$, 8 processors, with a fine grid of dimension $N \times (np0 \cdot \widetilde{N}), N \times (np0 \cdot \frac{N}{2})$, and $N \times (np0 \cdot \frac{N}{4})$. In each case collect the parallel time T and then compute $T_{comm} = T - T_{comp}$.
(3) Fit a straight line as in Eq. (2) (for both choices of $np = np0$) through the data collected in steps 1 and 2 to estimate $\alpha(np0)$ and $\gamma(np0)$.
(4) Fit a straight line as in (4) through the points $(2, \alpha(4))$ and $(3, \alpha(8))$ to estimate c and d: based upon Eq. (4) now compute $\alpha(np)$ for the required choice of np.
(5) Use the model in Eq. (2) to estimate the value of T_{comm} for the required choice of np (using the values $\gamma(np) = \gamma(8)$ and $\alpha(np)$ determined in steps 3 and 4 respectively).
(6) Combine T_{comm} from step 5 with T_{comp} (determined in step 1, with finest size $N \times \widetilde{N}$) to estimate T as in Eq. (1).

Users of Clusters A and B do not get exclusive access to their resources and hence some variations in the execution time of the same parallel job can be observed across different runs. For this reason it will never be possible to obtain perfect predictions for parallel run times, however it is possible to aim to obtain estimated timings that are within this range of variability. In order to achieve this some basic knowledge of the target HPC systems is required. For example, since Cluster A has a mixture of 2.2 GHz and 2.4 GHz processors it is necessary to ensure that all runs for determining both T_{comp} and T_{comm} use at least one slower processor. Similarly, for the multicore Cluster B, all timings for 4 or 8 cores should be taken using just one or two computational nodes respectively.

4.2. Numerical results

We have tested our model for a range of problems with four different cluster architectures (each permutation of Clusters A and B with Myrinet and Fast Ethernet switching) and on the two multigrid codes ($m1$ and $m2$) described in Sections 3.1.1 and 3.1.2 respectively. Results for a such range of problems are presented in Tables 1 and 2. Each table presents, for a different code, four combinations of parallel architecture (defined by cluster, number of processors and switching used), with the associated problem size for the target problem. The last three rows of each table give the elapsed time in seconds, and the associated predictions and errors from the methodology that has been described.

In many, but not all, cases our approach yields good predictions, with an error of less than 13%. Very disappointing results are obtained for the code $m2$ using architectures based upon Fast Ethernet switching however (see Table 2). In these cases the model severely under predicts the measured run times. Based upon further numerical experiments (see, for example, [24,25]), we conjecture that the simple linear model (Eq. (3)) used to describe the latency term, $\alpha(np)$, in Eq. (2) is too crude to capture the message-passing behaviour with Ethernet switching. This is particularly problematic for code $m2$ which, like most practical codes, contains a mixture of blocking as well as non-blocking communications. In the following section therefore, we generalize the predictive model to include a nonlinear dependence of $\alpha(np)$ on $\log(np)$.

5. A generalized predictive model

As described in the previous section, we now generalize the model (2)–(4) in order to improve the accuracy of the predictions across all of the test cases considered. The new model is based upon Eq. (2), however now the parameters α and γ are assumed to satisfy the following relations:

$$\alpha(np) \approx c + d\log_2(np) + e(\log_2 np)^2, \tag{5}$$
$$\gamma(np) \approx \text{ constant}. \tag{6}$$

The new quadratic term $e(\log_2 np)^2$ has been introduced in (6) (compared to (3)). This allows a degree of nonlinearity to the overhead model, and represents the main difference with respect to the multigrid overhead model described in the previous section.

5.1. The modified algorithm

We now summarize the new algorithm, based upon using (2), (6) and (5) to estimate T_{comm}. Note that the additional term in (6) means that runs are now required on 4, 8 and 16 cores in order to calculate the parameters in the model. The target configuration is again assumed to be for a $N \times N$ mesh with np processors (and, as before, $\widetilde{N} = N/np$).

Table 1
Test cases $np = 64$ on Cluster A and $np = 32$ on Cluster B for the multigrid code $m1$: measurements and prediction are given in seconds.

Cluster	Cluster A	Cluster A	Cluster B	Cluster B
nprocs	$np = 64$	$np = 64$	$np = 32$	$np = 32$
Switching	Ethernet	Myrinet	Ethernet	Myrinet
Size	65,536 × 65,536	65,536 × 65,536	32,768 × 32,768	32,768 × 32,768
Memory per core (GB)	2	2	1	1
Measurement	1703.9	1014.9	443.0	259.5
Prediction	1719.8	1133.9	454.3	226.4
|error| (%)	0.93	11.73	2.55	12.76

Table 2
Test cases $np = 64$ on Cluster A, and $np = 64$ on Cluster B for multigrid code $m2$: measurements and prediction are given in seconds.

Cluster	Cluster A	Cluster A	Cluster B	Cluster B
nprocs	$np = 64$	$np = 64$	$np = 64$	$np = 64$
Switching	Ethernet	Myrinet	Ethernet	Myrinet
Size	65,536 × 32,768	65,536 × 32,768	65,536 × 32,768	65,536 × 32,768
Memory per core (GB)	1.7	1.7	1.7	1.7
Measurement	1008.9	248.0	268.2	162.1
Prediction	546.8	222.8	181.0	153.0
|error| (%)	45.80	10.16	32.51	5.61

(1) Run the code on a single processor with a fine grid of dimension $N \times \widetilde{N}$ and then with dimension $N \times \frac{N}{2}$ and $N \times \frac{N}{4}$. In each case collect the computational time T_{comp} and define as *work* the memory allocated in the processor.
(2) Run the code on $np0 = 4, 8, 16$ processors, with a fine grid of dimension $N \times (np0 * \widetilde{N}), N \times (np0 * \frac{N}{2})$, and $N \times (np0 * \frac{N}{4})$. In each case collect the parallel time T and then compute $T_{comm} = T - T_{comp}$.
(3) Fit the best fitting line as in Eq. (2) (for all three choices of $np = np0$) through the data collected in steps 1 and 2 to estimate $\alpha(np0)$ and $\gamma(np0)$.
(4) Fit a parabola, as in Eq. (6), through the points $(2, \alpha(4))$, $(3, \alpha(8))$ and $(4, \alpha(16))$ to estimate c, d and e: based upon Eq. (6) now compute $\alpha(np)$ for the required choice of np.
(5) Use the model in Eq. (2) to estimate the value of T_{comm} for the required choice of np (using the values $\gamma(np) = \gamma(16)$ and $\alpha(np)$ determined in steps 3 and 4 respectively).
(6) Combine T_{comm} from step 5 with T_{comp} (determined in step 1, with finest size $N \times \widetilde{N}$) to estimate T as in Eq. (1).

5.2. Numerical results

Tables 3 and 4 present results for the generalized model, applied to the multigrid codes $m1$ and $m2$ respectively.

These tables confirm that the generalized model does indeed deliver more accurate performance predictions compared to those obtained with the linear model in Tables 1 and 2. In particular, where the linear model worked well, the generalized model still performs well, as one would expect. Furthermore, the poor results originally obtained for code $m2$, using the HPC systems with Fast Ethernet switching, have been significantly improved. Only one of the cases considered yields results that are outside of the range of variability of the run times across the shared systems (see the first column of Table 4). Even in this case however, the error is almost half that obtained in the previous section.

It should be noted that for fast, low-latency, switching, both of the models presented, in this and the previous section, perform very well. The Ethernet case is particularly challenging since the switching performance is significantly affected by other jobs being run on the system. For this reason we do not propose any further modifications to our models. Indeed, we next demonstrate, in the following section, that the generalized model performs very well when applied to a practical, and highly challenging, engineering software example.

Table 3
Application of the modified predictive model to test cases $np = 64$ on Cluster A $np = 32$ on Cluster B for multigrid code $m1$: measurements and predictions are given in seconds.

Cluster	Cluster A	Cluster A	Cluster B	Cluster B
nprocs	$np = 64$	$np = 64$	$np = 32$	$np = 32$
Switching	Ethernet	Myrinet	Ethernet	Myrinet
Measurement	1703.9	1014.9	443.0	259.5
Prediction	1692.3	1102.9	456.8	268.8
|error| (%)	0.68	8.67	3.12	3.58

Table 4
Application of the modified predictive model to test cases $np = 64$ on Cluster A $np = 64$ on Cluster B for multigrid code $m2$: measurements and predictions are given in seconds.

	Cluster A	Cluster A	Cluster B	Cluster B
Switching	Ethernet	Myrinet	Ethernet	Myrinet
Measurement	1008.9	248.0	268.2	162.1
Prediction	741.5	235.5	265.4	165.6
|error| (%)	26.5	5.04	1.04	2.16

6. Application to practical engineering software

We now test the predictive methodology described in the previous section on a practical engineering code, which is based on the use of the multigrid strategy. This code simulates the elastohydrodynamic lubrication (EHL) problem for a point contact, whose definition and parallel solution algorithm, are described briefly below. For a full discussion of the parallel implementation see [12].

6.1. Elastohydrodynamic lubrication

EHL plays an important role in many mechanical devices such as journal bearings or gears where, under very heavy loads, the extreme pressure in the lubricant causes elastic deformation of the contacting elements. This is typically modelled via a thin-film approximation for the lubricant flow, coupled with a film-thickness equation which captures the elastic deformation. With a suitable non-dimensionalization (see [34] for further details) the following equations are obtained on a two-dimensional domain $(X_{\min}, X_{\max}) \times (Y_{\min}, Y_{\max})$:

$$\frac{\partial}{\partial X}\left(\frac{\rho H^3}{\eta\lambda}\frac{\partial P}{\partial X}\right) + \frac{\partial}{\partial Y}\left(\frac{\rho H^3}{\eta\lambda}\frac{\partial P}{\partial Y}\right) - u_s\frac{\partial(\rho H)}{\partial X} = 0, \tag{7}$$

and

$$H(X,Y) = H_{00} + \frac{X^2}{2} + \frac{Y^2}{2} + \frac{2}{\pi^2}\int_{Y_{min}}^{Y_{max}}\int_{X_{min}}^{X_{max}} \times \frac{P(X',Y')}{\sqrt{(X-X')^2 + (Y-Y')^2}}\,dX'\,dY'. \tag{8}$$

Here P and H are the unknown pressure and film-thickness respectively, λ and u_s are constants, H_{00} is an unknown offset value which can be determined indirectly through a force balance constraint, such as

$$\int_{Y_{min}}^{Y_{max}}\int_{X_{min}}^{X_{max}} P(X,Y)\,dX\,dY = \frac{2\pi}{3}, \tag{9}$$

and the density ρ and viscosity η are given by the following empirical relations:

$$\rho(P) = \frac{0.59\times 10^9 + 1.34 p_h P}{0.59\times 10^9 + p_h P}$$

$$\eta(P) = \exp\left\{\frac{\alpha p_0}{z_i}\left[-1 + \left(1 + \frac{p_h P}{p_0}\right)^{z_i}\right]\right\}.$$

The coefficients p_h, p_0, α and z_i are assumed to be known and constant.

The code, that we refer to as *mEHL*, is based upon a finite difference approximation to (7) and a simple quadrature scheme for (8). The efficient solution of the resulting discrete system depends critically upon the use of multilevel methods:

- Parallel nonlinear multigrid (the FAS scheme [5], whose parallel implementation is described in [11,12]) is used for the solution of the discrete form of (7);
- Parallel multilevel multi-integration (MLMI), see [12,34] is used to evaluate the discrete form of (8).

The MLMI scheme is especially important since it allows the cost of evaluating the film thickness over the entire domain, approximated on a finest mesh of size $N \times N$, to be reduced from $O(N^4)$ to $O(N^2(\log N)^2)$. Note however that the parallel implementation of the MLMI requires each process to work with the entire computational domain at the coarsest mesh level. This computational step of *mEHL* therefore involves a multi-summation, for

approximating the double integral in (8), that is performed over the coarsest mesh of the multilevel scheme, $N^c \times N^c$ say, on all of the processors. For this reason, the *mEHL* algorithm is a little different to the benchmark parallel multigrid solvers that we have considered so far in this paper. This has an impact on the way in which the computational component of the parallel performance of the software should be predicted, as described below.

6.2. Predictive model

Since the software that we wish to model in this section has both a multigrid and a non-multigrid component, we can no longer assume the very straightforward evaluation of T_{comp} that was possible in the previous sections. Consequently, we now represent the term T_{comp} in the model as a sum of two components:

$$T_{comp} = T_{mgrid} + T_{nmgrid}, \tag{10}$$

where T_{mgrid} is the computational time associated with the multigrid operations and T_{nmgrid} is the rest of the computational cost of the code, principally due to the multi-summation computations required for the MLMI procedure. As before, assuming the parallel multigrid is run on *np* processors with a partition by rows, T_{mgrid} can be measured through a run on a single processor of the code, with size of the problem equal to $N \times \widetilde{N}$, with $\widetilde{N} = \frac{N}{np}$. The strategy for predicting T_{nmgrid} constitutes the fundamental contribution of this section, as a generalization of the multigrid methodology described previously, which only considers the case $T_{nmgrid} \approx 0$.

The cost T_{nmgrid} (required for each processor) has a multi-summation term that is quadratic with respect to the size of the overall coarsest mesh, $N^c \times N^c$, to leading order in powers of the dimension of the problem N^c:

$$T_{nmgrid} \propto \frac{(N^c N^c)^2}{np}. \tag{11}$$

A parallel run of *mEHL* performs the multi-summation over the coarsest mesh on each processor a certain number, *nsum*(*ngrids*,*ncoarse*), of times. This number depends of the number of grid levels, *ngrids*, and on the number, *ncoarse*, of smoothing sweeps at the coarsest level, as shown in (12) below. For a full explanation of this expression for *nsum* refer to [26].

$$nsum = nVC \cdot [ncoarse + (ngrids - 1) \times (npre + npost + 2)] \tag{12}$$

In (12): *ngrids* is the number of grids used; *ncoarse* is the number of smoothing sweeps at the coarsest level, *npre* is the number of pre-smoothing sweeps and *npost* is the number of post-smoothing sweeps in a single V-cycle.

In order to predict T_{nmgrid} we need to exploit its dependence with respect to both *nsum* and T_{comp} in (10). First, we observe that the same multi-summation work can be obtained across a sequence of serial runs of the code with a coarsest mesh $N^c \times N^c$ and different finest levels $2^l N^c \times 2^l N^c$ (for $l = 1,2,\ldots$), so long as we keep *nsum* constant through all these runs. The associated execution times obtained are denoted as T_l in the following part of the section. The parameter *nsum* is kept constant by appropriate variation of the parameter *ncoarse*, see [26]. In fact the possibility of changing this parameter in (12) permits us to obtain the same *nsum* (equal to the value used in the target problem that we wish to predict) through all the sequential runs (associated to $l = 1,2,\ldots$) where a different number of grids (*ngrids*) is used. Therefore, we can obtain T_{nmgrid} (as expressed in (11)) using the value extrapolated to $l = 0$ of the line plotted through the points (l, T_l) and dividing it by *np*.

The methodology for obtaining T_{nmgrid} can be therefore described through the following steps:

(1) run the code on a single processor with the coarsest mesh $N^c \times N^c$ and finest mesh $2^l N^c \times 2^l N^c$ for $l = 1,2,3$, in each case collect the execution time T_l obtained;
(2) determine the least square fitting line through the points (l,T_l) for $l = 1,2,3$;
(3) extrapolate the least square fitting line obtained in step *2* to $l = 0$;
(4) get as prediction for T_{nmgrid} the quotient obtained by dividing the extrapolated value obtained in step 3 by the number of processors *np* (see Eq. (11)).

In order to predict T_{mgrid} we need, as explained before, to run the code on a single processor with a finest mesh of size $N \times \widetilde{N}$. This is analogous of the methodology for predicting T_{comp} implemented for the pure multigrid solvers discussed previously. However, since now we are only interested to catch the computational cost associated with the multigrid, we do not consider terms due to the non-multigrid components of the software. These are therefore removed from the computational cost obtained.

The methodology for determining T_{mgrid} is then described through the following steps:

(1) run the code on a single processor with coarsest mesh $N^c \times \widetilde{N^c}$, with $\widetilde{N^c} = \frac{N^c}{np}$ and finest mesh $N \times \widetilde{N}$, saving the execution time as $T_{nlevels-1}$;
(2) run the code on a single processor with the same coarsest mesh as in step 1 and with the finest mesh equal to $2^l N^c \times 2^l \widetilde{N^c}$ for $l = 1, 2, 3$, in each case collect the execution time T_l obtained
(3) determine the least square fitting line through the points (l,T_l) for $l = 1, 2, 3$;
(4) extrapolate the least square fitting line obtained in step *2* to $l = 0$, obtaining the value T_0;
(5) get as prediction for T_{mgrid} the difference between $T_{nlevels-1}$ and T_0 (the former represents T_{mgrid} plus some MLMI work and T_0 is an estimate of this MLMI work, which must be therefore removed).

Finally, the computational time predicted, T_{comp}, is the sum of T_{nmgrid} and T_{mgrid} obtained from the methodology described.

Note that this approach may be used successfully regardless of the relative residual reduction that is to be used as the convergence criterion for the target problem. This is because, for an optimal multigrid implementation, the rate of convergence is independent of the mesh size and so the number of V-cycles (or W-cycles) required to reduce the norm of the residual by a constant factor (10^{-3} say) is independent of the level of mesh refinement. This is illustrated for the *mEHL* software in Fig. 7, which shows the residual reduction versus the number of V-cycles for runs using different maximum refinement levels. Consequently, so long as the convergence tolerance is always based upon a relative reduction in the residual (which is the only sensible approach to use, given that the residual can always be scaled through multiplication by a constant factor) our approach of basing the estimate of the run time of the target problem on a fixed number of cycles is certainly valid.

6.3. Numerical results

In Tables 5 and 6 we present results of the application of this predictive methodology to determine T_{comp} in *mEHL*, combined with the generalized methodology for T_{comm} (described in Section 5). The predictions obtained with this methodology are seen to be very accurate, with an error well below 10% in each combination of switching interconnect, problem size, processor number and cluster that is considered.

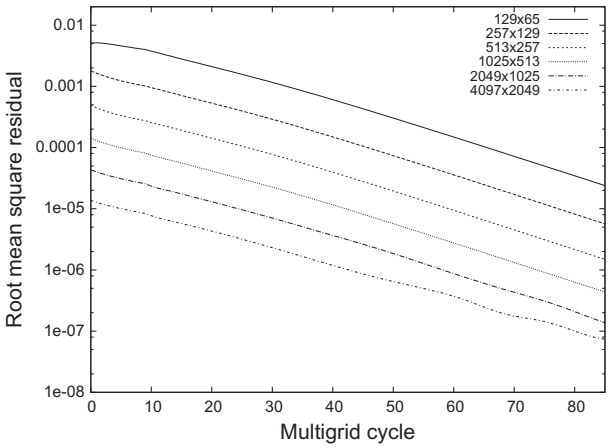

Fig. 7. Rate of convergence of the *mEHL* solver when applied using different maximum refinement levels: the constant slope of each graph implies that the rate of convergence is independent of the maximum refinement level.

Table 5
EHL predictions and measurements (both quoted in seconds) for Cluster A.

nprocs	$np = 32$	$np = 64$	$np = 64$	$np = 128$
Size	8193×2049	8193×4097	16385×8193	16385×16385
Switching	Ethernet	Ethernet	Myrinet	Myrinet
Measurement	250.5	361.5	1074.9	1260.2
Prediction	267.1	365.5	1051.3	1242.9
|error| (%)	6.63	0.97	2.20	1.37

Table 6
EHL predictions and measurements (both quoted in seconds) for Cluster B.

nprocs	$np = 32$	$np = 64$	$np = 64$	$np = 128$
Size	8193×2049	8193×4097	16385×8193	16385×16385
Switching	Ethernet	Ethernet	Myrinet	Myrinet
Measurement	177.3	241.8	908.4	1124.2
Prediction	172.1	222.3	904.4	1107.8
|error| (%)	2.93	8.06	0.44	1.46

7. Alternative partitioning strategies

So far in this paper we have only considered partitions of the computational mesh in which different numbers of rows are stored on the different processors. For the EHL code in the previous section this is the only partitioning strategy that is available, however for other parallel multigrid codes different partitioning strategies may be available. This leads to the possibility of adding a further level of complexity to our predictive methodology, in which we seek to estimate the effects of different partitioning strategies on the performance of multigrid codes on different HPC resources. Indeed, it is well known that, depending on the particular problem and hardware combination, different geometric partitionings of the computational work can show better performance than others, see for example [20,32]. In this section therefore, we consider the further generalization of our methodology to a block partitioning strategy, in which the unknowns may be split across the processors by column as well as by row, [27]. Such a partitioning strategy is possible for our code *m*1, so this is used in our numerical experiments in order to assess the effectiveness of our new predictions.

7.1. Block partitioning

We consider a rectangular computational mesh of dimension $N_a \times N_b$, together with a given homogeneous "block" distribution of the computational domain across the available processors. Specifically, we consider the mesh to be mapped onto *np* processors as a bi-dimensional grid $np_a \times np_b$, with

$$np = np_a \cdot np_b,$$

see, for example, Fig. 8. As in this figure, in the following we use the notation (np_a, np_b) to indicate the case where a grid of $np_a \times np_b$ processors is used to partition the computational mesh.

Following the previous sections, the first assumption that we make is that the parallel solution time (on *np* processors) may be represented as (1). Now, in (1), T_{comp} represents the computational time for a problem of size $\widetilde{N_a} \times \widetilde{N_b}$ on a single processor (where $\widetilde{N_a} = \frac{N_a}{np_a}$ and $\widetilde{N_b} = \frac{N_b}{np_b}$), and T_{comm} again represents all of the parallel overheads (primarily due to inter-processor communications). A typical parallel multigrid code, such as *m*1, has a computational work on each internal processor (i.e. each processor which has an interior subdomain mapped to it) that is proportional to $(\widetilde{N_a} + 2) \times (\widetilde{N_b} + 2)$, as shown in Fig. 9. However, when only a single processor is used the equivalent computational mesh is of dimension $\widetilde{N_a} \times \widetilde{N_b}$. The task of estimating T_{comp} reliably is complicated somewhat by this observation (compared to the model for strip partitions described previously for example) and so a more general approach is considered.

The computational time is now assumed to be equal to that associated with a solution on a (2,2) processor grid, where the size of the problem is scaled in such way that each processor in the (2,2) grid solves on a computational mesh of dimension $\widetilde{N_a} \times \widetilde{N_b}$. In this way we consider each combination of one ghost row with one ghost column as arises in a general parallel partition, see Fig. 8. Our approach is not to seek to measure T_{comp} explicitly, but instead to determine it implicitly as an expression involving the measured parallel time ($T_{(2,2)}$), across the (2,2) partition, through the relation

$$T_{(2,2)} = T_{comp} + T_{comm(2,2)}$$

(see below for further details).

7.2. Communication costs

The communication phase consists of a sequence of sends and receives between neighbouring processors through the use of the

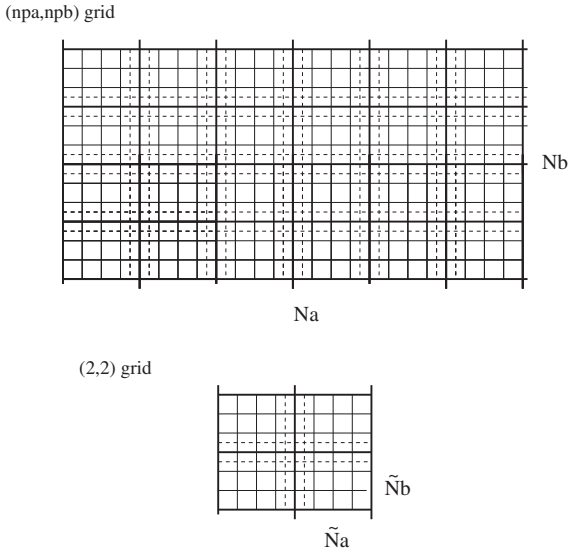

Fig. 8. Example of a computational grid for a target problem with a (np_a, np_b) partition and an equivalent sized problem for the (2,2) processor grid, as used in the predictive methodology. Note that each of the processors in both problems have the same work except for the variation in the number of ghost rows and columns.

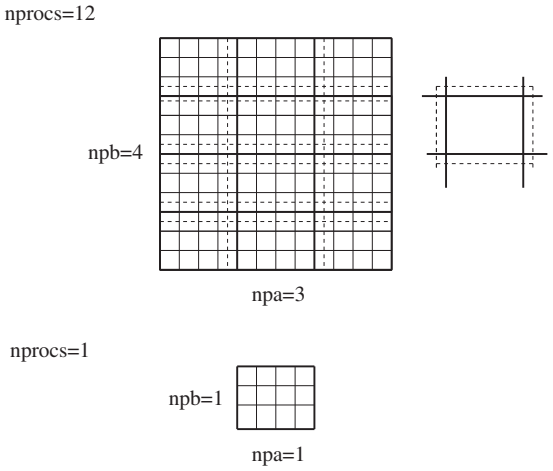

Fig. 9. Parallel distribution of the grid nodes in a block parallel strategy. Each computational node owns a section of the mesh enclosed by the solid lines: the dashed lines represent ghost rows and columns added for efficient parallel implementation. The left upper diagram illustrates the case *nprocs* = 12, with $np_a = 3$ and $np_b = 4$, whilst the right upper shows the effect of the ghost rows and columns on the section of the mesh that must be held for a computational node that is away from the boundary of the domain. The bottom diagram illustrates the computational grid when the code is implemented on a single processor (*nprocs* = 1): in this case no ghost rows or columns are used.

MPI library. When non-blocking communications are used exclusively, a theoretical full overlapping of the communications between processors of the same row and of the same column of the processor grid is expected. T_{comm} is then equal to the largest overhead time measured between processors in the same row and those in the same column. We assess that these overheads depend upon the number of processors in the same row and in the same column, respectively. This is a reasonable assessment due to the fact that, as described in the previous sections, the overhead in a strip partition depends upon to the number of processors used. Let $T_{comm(np_a,1)}$, and $T_{comm(1,np_b)}$, be the overhead times measured on the strip of processors (np_a,1), and in the column (1,np_b), respectively. Then, based upon the previous argument, we would have

$$T_{comm} = \max(T_{comm(np_a,1)}, T_{comm(1,np_b)}). \tag{13}$$

We remark however that Eq. (13) is not valid for real codes, because there is some asynchrony in the communication: both the complexity of the code and the access of memory induces an asynchronous term for the elapsed overhead time. The expression that represents this additional term, which we denote as T_{EXTRA}, depends upon the particular code being used. The next assumption that we make however is that this term is equal to that measured for the (2,2) grid of processors, where each processor solves on a $\widetilde{N_a} \times \widetilde{N_b}$ computational mesh. We then have that

$$T_{(2,2)} = T_{comp} + T_{EXTRA} + \max(T_{comm(2,1)}, T_{comm(1,2)}),$$

hence the resulting expression for T_{EXTRA} is

$$T_{EXTRA} = T_{(2,2)} - T_{comp} - \max(T_{comm(2,1)}, T_{comm(1,2)}). \tag{14}$$

Now, for the target problem on the (np_a,np_b) grid of processors, we have

$$T_{comm} = T_{EXTRA} + \max(T_{comm(np_a,1)}, T_{comm(1,np_b)}) \tag{15}$$

Hence using (1), (15) and (14)

$$\begin{aligned} T &= T_{comp} + T_{EXTRA} + \max(T_{comm(np_a,1)}, T_{comm(1,np_b)}) \\ &= T_{(2,2)} + \max(T_{comm(np_a,1)}, T_{comm(1,np_b)}) - \max(T_{comm(2,1)}, T_{comm(1,2)}) \\ &\approx (\leqslant) T_{(2,2)} + \max(T_{comm(np_a,1)} - T_{comm(2,1)}, T_{comm(1,np_b)} - T_{comm(1,2)}) \\ &= T_{(2,2)} + \max(T_{(np_a,1)} - T_{(2,1)}, T_{(1,np_b)} - T_{(1,2)}) \end{aligned}$$

Finally we have

$$T \approx T_{(2,2)} + \max(T_a, T_b), \tag{16}$$

where $T_a = T_{(np_a,1)} - T_{(2,1)}$ and $T_b = T_{(1,np_b)} - T_{(1,2)}$.

Based upon the evidence analysed in the previous sections both overheads T_a and T_b are assumed to satisfy a relation of the following type:

$$T_a = \alpha(np_a) + \gamma(np_a) \cdot work, \tag{17}$$

$$T_b = \alpha(np_b) + \gamma(np_b) \cdot work. \tag{18}$$

In (17) and (18) the term *work* is used to represent the problem size on each processor at the finest level which can be expressed in MBytes of the memory required since the computational work is proportional to the mesh size for a multigrid implementation. Also note that the length of the messages (that is N_b for the grid (np_a,1) and N_a for the grid (1,np_b)) does not appear in this formula since it is assumed that for a given size of target problem (e.g. a mesh of dimension $N_a \times N_b$ and a partition of dimension $np_a \times np_b$) the size of the messages is known *a priori*. Finally, following the generalized model of Section 5, we assume that the same nonlinear relations are sufficient:

$$\alpha(np) \approx c + d\log_2(np) + e(\log_2 np)^2, \tag{19}$$

$$\gamma(np) \approx \text{constant}. \tag{20}$$

A summary of the overall predictive methodology is provided by the following steps. We define as $N_a \times N_b$ and (np_a,np_b) the target problem size and target grid of processors respectively (i.e. we wish to predict a code's performance for these values). Also, let $\widetilde{N_a} = N_a/np_a$ and $\widetilde{N_b} = N_b/np_b$, and define $\widetilde{N_a} \times \widetilde{N_b}$ to be the size of problem (not considering the ghost rows) on each processor in the target configuration.

(1) Run the code on a (2,2) grid with a fine grid of dimension $(2\widetilde{N}_a) \times (2\widetilde{N}_b)$ and collect the parallel time $T_{(2,2)}$.
(2) Run the code on the grids (np,1) with $np = 2, 4, 8, 16$ processors, with a fine grid of dimension $(np * \frac{N_a}{l}) \times \widetilde{N_b}$ for $l = 1, 2, 4$. Define as *work* the memory allocated in each processor. In each case collect the parallel time $T_{(np,1)}$ and then compute $T_{(np^*,1)} - T_{(2,1)}$ with $np^* = 4, 8, 16$. Similar steps are computed to collect $T_{(1,np^*)} - T_{(1,2)}$, with $np^* = 4, 8, 16$.
(3) Fit a straight line (using a least squares fit), as in Eq. (17) or (18) (for each choice of $np = np^*$), through the data collected in step 2 in order to estimate $\alpha(np^*)$ and $\gamma(np^*)$ for both T_a and T_b.
(4) Fit a parabola, as in Eq. (20), through the points $(2,\alpha(4))$, $(3,\alpha(8))$ and $4,\alpha(16))$ to estimate c, d and e based upon Eq. (20): now compute $\alpha(np)$ for the required choice of np.
(5) Use the model in Eq. (17) to estimate the values of T_a (and use the model in Eq. (18) to estimate T_b) for the required choice of np (using the values $\gamma(np) = \gamma(16)$ and $\alpha(np)$ determined in steps 3 and 4 respectively).
(6) Determine $\max(T_a,T_b)$ from step 5 and combine with $T_{(2,2)}$ (determined in step 1) to estimate T as in Eq. (16).

7.3. Numerical results

Using the methodology described we are able to predict the performance of the code *m*1 for a selection of target configurations. The actual measured times once the target problems are run, the

Table 7
Measurements and predictions for Cluster A (both quoted in seconds).

np	(np_a, np_b)	Size	Mem. per proc. (GB)	Meas.	Predict.	|error| (%)
64	(8,8)	65,536 × 32,768	1	378.36	358.89	5.1
64	(4,16)	32,768 × 65,536	1	370.16	353.57	4.5
64	(2,32)	16,384 × 131,072	1	315.82	320.05	1.3
32	(8,4)	65,536 × 16,384	1	372.94	358.89	3.7
32	(4,8)	16,384 × 65,536	1	373.11	353.57	5.2

Table 8
Measurements and predictions for Cluster B (both quoted in seconds).

np	(np_a, np_b)	Size	Mem. per proc. (GB)	Meas.	Predict.	|error| (%)
128	(16,8)	131,072 × 65536	2	522.83	519.81	0.58
128	(8,16)	65,536 × 131072	2	493.71	506.43	2.6
128	(32,4)	262,144 × 32768	2	533.86	504.85	7.9
128	(4,32)	32,768 × 262144	2	512.18	507.28	0.96
64	(8,8)	65,536 × 65,536	2	510.59	506.43	0.81
64	(16,4)	131,072 × 32,768	2	478.07	519.81	8.7
64	(4,16)	32,768 × 131,072	2	507.28	474.76	6.4
64	(32,2)	262,144 × 16,384	2	564.54	533.86	5.4
64	(2,32)	16,384 × 262,144	2	534.80	496.75	7.1
32	(8,4)	65536 × 32,768	2	481.36	506.43	5.2
32	(4,8)	32768 × 65,536	2	507.28	495.66	2.3

predicted times and the resulting errors are then listed in the last three columns of the following tables (Tables 7 and 8 for runs on Clusters A and B respectively).

These results show a very accurate and robust prediction with respect to each of: the target problem size, the target number of processors, the target partition and the parallel architecture used. In fact the methodology can detect the performance of the multigrid code with an error below 10% for all the numerical tests considered. This level of accuracy is certainly sufficient to be able to guide decisions as to the scheduling of parallel jobs on available resources, although it may not always be sufficient to allow the optimal decomposition to be predicted. Specifically, when there is little to choose between the efficiency of different partitions, an error of up to 10% could lead to slightly a sub-optimal partition being selected. Highly inappropriate partitions will always come out worse in the computational model however.

8. Conclusions

In this paper we have presented a simple and general methodology for the performance of parallel engineering software for across a range of HPC resources. The simplicity of the approach stems from the use of empirical models for the communication overheads, which contain just a small number of parameters which may be determined from, carefully selected, short runs on small numbers of processors. The generality comes from the approach of separating out the calculation of T_{comp} from the extrapolated estimate of T_{comm}. Of course the precise manner in which T_{comp} is found must vary between applications (depending upon whether they are pure multigrid, in which case the total work scales as $O(N)$, or whether there are components that scale less well than this, as with the code considered in Section 6). This generality has been illustrated by the application of the methodology to a variety of multigrid-based software tools, two different data partitioning strategies, and a selection of processor and communication architectures. The results obtained show that the difference between predicted run times for the target problems (using large numbers of processors) and the subsequent measured run times are almost always within the range of variability of the execution time on the shared systems that have been used (e.g. a difference of less than 10%).

All of the examples considered so far have involved the use of structured grids in two dimensions. There appear to be no reasons why the extension to structured grids in three-dimensions, using strip or block partitions such as those considered here, should present any additional practical difficulties. As has already been discussed, the use of geometrically simple domains such as squares or cubes does not necessarily mean that the problem being simulated is also geometrically simple. Nevertheless, there are likely to be times where numerical methods involving more complex domains, and the use of unstructured grids, may be of interest. We believe that providing the partitioning strategy is based upon splitting the coarsest mesh across the parallel processors (e.g. [15,32]) then the approach described above can be extended to multigrid implementations based upon uniform refinements of this base grid. In particular, the use of partitioning strategies based upon coordinate bisection, as illustrated in [32] for example, should allow the techniques considered here to be applied successfully. One area where it will be difficult to extend this work however would be in cases where multigrid is used in combination with local mesh adaptivity and parallelism. This is because it would be difficult to predict in advance precisely what the size of the target problem would actually be, given that this would depend upon the *a posteriori* error estimate that would not be known until the full execution takes place. The work of [2] does appear to offer a possible approach to tackling this problem however considerable additional research will be required.

There are many potential applications of the predictive capability that has been developed here. Computational scientists and engineers will be able to reserve computational resources that are appropriate for the size of the computations that they wish to undertake: avoiding the potential delays associated with requesting excessive resources, or the need to resubmit jobs when insufficient resources were requested. They will also be able to make informed decisions regarding the additional costs and turnaround times, and the potential lack of available resources, associated with increasing the resolution of a given computational simulation. Furthermore, both individuals and software that are responsible for the scheduling of computational work across shared resources, such as a computational Grid for example, can benefit from having more reliable data as to the likely run times of codes prior to them being executed.

The primary theoretical underpinning for this work is based upon the scalability of multigrid algorithms with regard to the number of unknowns on the finest mesh, and the separation of the prediction of the computation and the communication costs. The simplicity of the approach that is then used to estimate these costs makes the technique widely applicable, however it is reasonable to expect that less empirical approaches should be able to provide more accurate predictions for specific codes and particular hardware architectures, albeit at the cost of some of this simplicity. This is clearly a trade-off that should be considered when selecting the most appropriate predictive tool for any given application.

Acknowledgements

We are grateful to the UK Engineering and Physical Sciences Research Council for supporting this work via grant EP/C010027/1.

References

[1] Adalsteinsson D, Sethian JA. A fast level set method for propogating interfaces. J Comput Phys 1995;118:269–77.
[2] Bank RE, Holst MJ. A new paradigm for parallel adaptive meshing algorithms. SIAM Rev 2003;45:292–323.

[3] Bank RE, Jimack PK. A new parallel domain decomposition method for the adaptive finite element solution of elliptic partial differential equations. Concurr Comput: Pract Exper 2001;13:327–50.
[4] Brandt A. Multi-level adaptive solutions to boundary value problems. Math Comput 1977;31:333–90.
[5] Briggs WL, Henson VE, McCormick SF. A multigrid tutorial. SIAM; 2000.
[6] Carrington L, Laurenzano M, Snavely A, Campbell R, Davis L. How well can simple metrics represent the performance of HPC applications? In: Proceedings of supercomputing 2005; 2005.
[7] Culler DE, Karp RM, Patterson DA, Sahay A, Schauser KE, Santos E, et al. LogP: towards a realistic model of parallel computation. SIGPLAN Not 1993;28: 1–12.
[8] Dew PM, Schmidt JG, Thompson M, Morris P. The white rose grid: practice and experience. In: Cox SJ, editor. Proceedings of the 2nd UK all hands e-science meeting. EPSRC; 2003.
[9] Gaskell PH, Jimack PK, Sellier M, Thompson HM. Efficient and accurate time adaptive multigrid simulations of droplet spreading. Int J Numer Methods Fluids 2004;45:1161–86.
[10] Gaskell PH, Jimack PK, Sellier M, et al. Gravity-driven flow of continuous thin liquid films on non-porous substrates with topography. J Fluid Mech 2004;509:253–80.
[11] Gaskell PH, Jimack PK, Koh YY, Thompson HM. Development and application of a parallel multigrid solver for the simulation of spreading droplets. Int J Numer Methods Fluids 2008;56:979–1002.
[12] Goodyer CE, Berzins M. Parallelization and scalability issues of a multilevel elastohydrodynamic lubrication solver. Concurr Comput: Pract Exper 2007;19:369–96.
[13] Goodyer CE, Berzins M, Jimack PK, Scales LE. A grid-enabled problem solving environment for parallel computational engineering design. Adv Eng Softw 2006;37:439–49.
[14] Hirt CW, Nichols BD. Volume of fluid (VOF) method for the dynamics of free boundaries. J Comput Phys 1981;39:201–25.
[15] Hodgson DC, Jimack PK. A domain decomposition preconditioner for a parallel finite element solver on distributed unstructured grids. Parallel Comput 1997;23:1157–81.
[16] Hu X, Li R, Tan T. A multi-mesh, adaptive finite element approximation to phase field models. Commun Comput Phys 2009;5:1012–29.
[17] Kerbyson DJ, Alme HJ, Hoisie A, Petrini F, Wasserman HJ, Gittings M. Predictive performance and scalability modeling of a large-scale application. In: Proceedings of supercomputing 2001; 2001.
[18] Koh YY. Efficient numerical solution of droplet spreading flows. Ph.D. Thesis. University of Leeds; 2007.
[19] Lang S, Wittum G. Large-scale density-driven flow simulations using parallel unstructured grid adaptation and local multigrid methods. Concurr Comput: Pract Exper 2005;17:1415–40.
[20] Pantelelis NG, Kanarachos AE. The parallel block adaptive multigrid method for the implicit solution of the Euler equations. Int J Numer Methods Fluids 1996;22:411–28.
[21] Pllana S, Brandic I, Benkner S. A survey of the state of the art in performance modeling and prediction of parallel and distributed computing systems. Int J Comput Intel Res (IJCIR) 2008;4:17–26.
[22] Rodriguez G, Badia RM, Labarta J. Generation of simple analytical models for message passing. In Danelutto M et al., editors. Euro-Par 2004 parallel processing. Springer LNCS, vol. 3149. Springer; 2004. p. 183–8.
[23] Romanazzi G, Jimack PK. Performance prediction for parallel numerical software on the white rose grid. In: Cox SJ, editor. Proceedings of UK e-science all hands meeting. 2007. p. 517–24 [ISBN 978-0-9553988-3-4].
[24] Romanazzi G, Jimack PK. Parallel performance prediction for multigrid codes on distributed memory architectures. In: Perrott R et al., editors. High performance computing and communications (HPCC-07). LNCS, vol. 4782. Springer; 2007. p. 647–58.
[25] Romanazzi G, Jimack PK. Parallel performance prediction for numerical codes in a multi-cluster environment. In: Ganzha M et al., editors. Proceedings of the 2008 international multiconference on computer science and information technology (IMCSIT'08). Katowice, Poland: PTI Press; 2008. p. 467–74.
[26] Romanazzi G, Jimack PK, Goodyer CE. Reliable performance prediction for parallel scientific software in a multi-cluster grid environment. In: Papadrakakis K, Topping BHV, editors. Proceedings of the sixth international conference on engineering computational technology. Civil-Comp Press; 2008 [paper 9].
[27] Romanazzi G, Jimack PK. Performance prediction for multigrid codes implemented with different parallel strategies. In: Topping BHV, Ivanyi P, editors. Proceedings of the first international conference on parallel, distributed and grid computing for engineering. Civil-Comp Press; 2009 [paper 43].
[28] Rosam J, Jimack PK, Mullis AM. A fully implicit, fully adaptive time and space discretization method for phase-field simulation of binary alloy solidification. J Comput Phys 2007;225:1271–87.
[29] Sanjay HA, Vadhiyer S. Performance modeling of parallel applications for grid scheduling. J Parallel Dist Comput 2008;68:1135–45.
[30] Schopf J, Berman F. Using stochastic information to predict application behavior on contended resources. Int J Found Comput Sci 2001;12:341–64.
[31] Snir M, Otto SW, Huss-Lederman S, Walker DW, Dongarra J. MPI – the complete reference. MIT Press; 1996.
[32] Touheed N, Selwood P, Jimack PK, Berzins M. A comparison of some dynamic load-balancing algorithms for a parallel adaptive flow solver. Parallel Comput 2000;26:1535–54.
[33] Trottenberg U, Oosterlee CW, Schüller A. Multigrid. Academic Press; 2003.
[34] Venner CH, Lubrecht AA. Multilevel methods in lubrication. Amsterdam: Elsevier; 2000.