

Parallel Performance Prediction for Multigrid Codes on Distributed Memory Architectures

Giuseppe Romanazzi and Peter K. Jimack

School of Computing, University of Leeds, Leeds LS2 9JT, UK
{roman,pkj}@comp.leeds.ac.uk

Abstract. We propose a model for describing the parallel performance of multigrid software on distributed memory architectures. 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. This has potential applications for the scheduling of jobs in a Grid computing environment where reliable predictions as to execution times on different systems will be valuable. The model is tested for two different multigrid codes running on two different parallel architectures and the results obtained are discussed.

Keywords: Parallel Distributed Algorithms; Grid Computing; Cluster Computing; Performance Evaluation and Prediction.

1 Introduction

Multigrid is one of the most powerful numerical techniques to have been developed over the last 30 years [1, 2, 13]. As such, state-of-the-art parallel numerical software is now increasingly incorporating multigrid implementation in a variety of application domains [6, 9, 11]. In this work we seek to model the performance of two typical parallel multigrid codes on distributed memory architectures. The goal is to be able to make accurate predictions of the performance of such codes on large numbers of processors, without actually executing them on all of these processors. This is of significant potential importance in an environment, such as that provided by Grid computing, where a user may have access to a range of shared resources, each with different costs and different levels of availability, [4, 7, 10].

A vast literature on performance models exists, varying from analytical models designed for a single application through to general frameworks that can be applied to many applications on a large range of HPC systems. This latter approach is typically based upon a convolution of an application trace with some benchmarks of the HPC system used. Both approaches have been demonstrated to be able to provide accurate and robust predictions, although each has its potential drawbacks too. In the former, for example, significant expertise is needed in deriving the analytic model, which is extremely code specific, whereas in the

latter approach, a large amount of computer time is typically required for tracing the application.

Considering these limitations, the choice between these two approaches will depend primarily on the goal of the predictions. For example, when it is most important to predict the run-time of large-scale applications on a given system, as opposed to just comparing their relative performance, it is preferable to build and apply a detailed analytic model for the available set of HPC systems, as in [8] for example. On the other hand, when it is more important to compare the performance of some real applications on different machines, the latter approach is preferable: in which case different benchmarks metrics can be used and convoluted with the application trace file, as in [3] or [5].

Our approach lies between these two extremes. We use relatively crude analytic models that are applicable to a general class of algorithms (multigrid) and through simulations of the application on a limited number of CPUs we attempt to evaluate the parameters of these models. In comparison with the first approach the sophistication of the analytic model is much less (but also much less dependent on the specific code or implementation). In comparison with the second approach, there is no need for tracing the application nor running large numbers of HPC benchmarks on the HPC facility: our benchmarking is simply based upon execution of the code on small numbers of processors of the HPC system.

The layout of the remainder of the paper is as follows. In the next section 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 performance of two such codes on an abstract distributed memory architecture. The analysis is then used to build a predictive model for this class of codes, 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. The paper concludes with a description of some numerical tests to assess the accuracy and robustness of these predictions and a discussion of the outcomes obtained. Further extensions of the work are also suggested.

2 Multigrid and Parallel Implementation

The general principal upon which multigrid is based is that when using many iterative solvers for the systems of algebraic equations that result from the discretization of PDEs, the component of the error that is damped most quickly is the high frequency part [2, 13]. This observation leads to the development of an algorithm which takes a very small number of iterations on the finite difference or finite element grid upon which the solution is sought, and then restricts the residual and equations to a coarse 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 implementation of such an algorithm requires a number of components to be implemented in parallel:

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

There is also a variant of the algorithm (primarily designed with nonlinear problems in mind), known as FAS (full approximation scheme) [13], which requires the solution as well as the residual to be restricted to the coarser grid at each level.

In this work we consider the parallel implementation of two multigrid codes: one is standard and the other uses the FAS approach. In both cases they partition a two-dimensional finite difference grid across a set of parallel processors by assigning blocks of rows to different processors. Note that if 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.

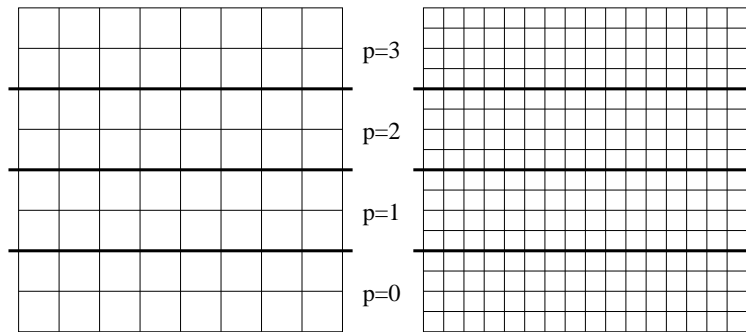


Fig. 1. Partitioning of a coarse and a fine mesh across four processors by assigning a block of rows to each processor

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, converge 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.

2.1 The algorithm m1

This algorithm solves the steady-state equation

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega, \\ \Omega &= [0, 1] \times [0, 1], \\ u|_{\partial\Omega} &= 0. \end{aligned}$$

The discretization is based upon a centred finite difference scheme on each grid. The iterative solver employed is the well-known Red-Black Gauss-Seidel (RBGS) method [9], 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 . Each processor also stores an extra, dummy, row of unknowns above and below its own top and bottom row respectively (see Figure 2): this is used to duplicate the contents of the corresponding row on each neighbour. After each red and black sweep of RBGS there is an inter-processor communication in which these dummy rows are updated. This is implemented with a series of non-blocking sends and receives in MPI. Similar inter-processor communication is required at the restriction step but the interpolation step does not require any message passing. A global reduction operation is required to test for convergence.

2.2 The algorithm m2

This algorithm, described in more detail in [9], uses an unconditionally-stable implicit time-stepping scheme to solve the transient problem

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

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 , see Fig. 2. This means that the total memory requirement is slightly less than with algorithm m1 but that, unlike m1, 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 [13] and so the current solution must also be interpolated from the fine to the coarser grid at each level.

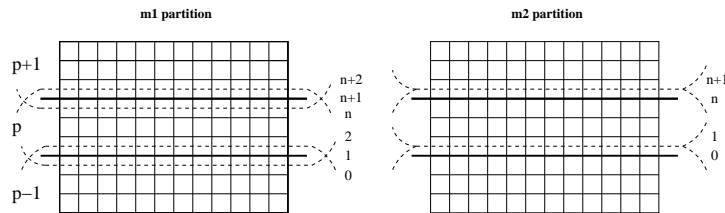


Fig. 2. Domain partitions for algorithms m1 and m2. The dummy rows of processor p have indexes 0, $n + 2$ and 0, $n + 1$ for m1 and m2, respectively

3 The Predictive Model

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 aim is to be able to predict these overheads in the situation where the size of the problem (the number of discrete unknowns, N^2 , on the finest mesh) scales in proportion to the number of processors. Consequently, the basic assumption that we make is that the parallel solution time (on np processors) may be represented as

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

In (1), T_{comp} represents the computational time for a problem of size N^2/np ($= N(1)^2$, say) on a single processor, 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 execution of a problem of size $N(1)^2$ on a single processor. One of the major attractions of the multigrid approach is that this time should be proportional to the size of the problem on the single processor. As demonstrated in [12] this is indeed the case for both of the implementations, m1 and m2, considered in this paper. The key consequence of this property is that in situations where we scale the problem size with the number of processors, np , then T_{comp} remains independent of np .

The more challenging task that we have, therefore, is to model T_{comm} in a manner that will allow predictions to be made for large values of np . Note that any additional work that is undertaken when computing in parallel is associated with the dummy rows that are stored on each processor, as is the communication overhead. When solving on a fine mesh of size N^2 the value of T_{comm} may be estimated, to leading order, as

$$T_{comm} \propto T_{start} + kN. \quad (2)$$

Here T_{start} relates to the start-up cost for the communications whilst kN represents the number of tasks of size N at each multigrid cycle (which will depend upon the number of communications and the number of additional iterative solves on duplicated rows). Furthermore, it is important to note that k itself may not necessarily be independent of N . In fact, slightly more than np times the work is undertaken by the parallel multigrid algorithm on an N^2 fine mesh than by the sequential multigrid solver on an N^2/np fine mesh. This is because in the former case either there are more mesh levels, or else the coarsest grid must be finer. In all of the examples discussed in this paper we consider each problem using the same coarse level N_0^2 (with N_0 a power of 2) and the same work per processor at the finest level (N^2 on np processors, where $N = \sqrt{np}N(1)$). Therefore, the number of grid levels used, $nlevels$, depends on the number of processors np , that is

$$nlevels = \log_2(N/N_0) + 1. \quad (3)$$

We consider two different models for T_{comm} , based upon (2). In the first of these we do assume that k is approximately constant to obtain

$$T_{comm} = a + bN. \quad (4)$$

In the second model we add a quadratic term $\gamma N(1)^2$ that allows a nonlinear growth of the overhead time

$$T_{comm} = \alpha + \beta N + \gamma N(1)^2. \quad (5)$$

The quadratic term introduced in (5) is designed to allow a degree of nonlinearity to the overhead model, reflecting the fact that k in (2) may not necessarily be a constant. As will be demonstrated below, the importance (or otherwise) of this nonlinear effect depends upon the specific characteristics of the processor and communication hardware that are present on each particular parallel architecture. In particular, the effects of caching and of a multicore architecture appear to require such a nonlinear model.

In order to obtain values for (a, b) and (α, β, γ) in (4) and (5) respectively, the parallel performance of a given code must be assessed on each target architecture. Note from Fig. 1 that the communication pattern is identical regardless of np : requiring only neighbour to neighbour communications. Hence our next assumption is that (a, b) or (α, β, γ) may be determined using just a small number of processors. In this work we choose $np = 4$ in order to approximate (a, b) or (α, β, γ) by fitting (4) or (5) respectively to a plot of $(T_{parallel} - T_{comp})$ against N . Note that, from (1), $T_{comm} = T_{parallel} - T_{comp}$ and T_{comp} is known from the data collected on a single processor.

A summary of the overall predictive methodology is provided by the following steps. In the following notation np is the target number of processors and N^2 is the largest problem that can be solved on these processors without swapping effects causing performance to be diminished. Similarly, $N(1)^2 = N^2/np$ is the largest such problem that can fit onto a single processor.

1. For $\ell = 1$ to m

- Run the code on a single processor with a fine grid of dimension $(2^{1-\ell}N(1))^2$.
 In each case collect T_{comp} based upon average timings over at least 5 runs.
2. For $\ell = 1$ to m

Run the code on 4 [or 8] processors, with a fine grid of dimension $(2^{2-\ell}N(1))^2$ [or $(2^{2-\ell}N(1) \times 2^{3-\ell}N(1))$].

In each case collect $T_{parallel}$ based upon average timings over at least 5 runs.
 3. Fit a straight line of the form (4) through the data collected in steps 1 and 2 to estimate a and b , or fit a quadratic curve through the data collected to estimate α , β and γ in (5).
 4. Use either model (4) or model (5) to estimate the value of T_{comm} for larger choices of np and combine this with T_{comp} (determined in step 1) to estimate $T_{parallel}$ as in (1).

4 Numerical Results

The approach derived in the previous section is now used to predict the performance of the two multigrid codes, m1 and m2, on the two parallel architectures WRG2 and WRG3. These computers form part of the University of Leeds' contribution to the White Rose Grid [4]

- WRG2 (White Rose Grid Node 2) is a cluster of 128 dual processor nodes, each based around 2.2 or 2.4GHz Intel Xeon processors with 2GBytes of memory and 512 KB of L2 cache. Myrinet switching is used to connect the nodes and Sun Grid Engine Enterprise Edition provides resource management.
- WRG3 (White Rose Grid Node 3) is a cluster of 87 Sun microsystem dual processor AMD nodes, each formed by two dual core 2.0GHz processors. Each of the $87 \times 4 = 348$ batched processors has L2 cache memory of size 512KB and 2GBytes of physical memory. Again, both Myrinet switching and Sun Grid Engine are used for communication and resource management respectively.

Table 1 shows values of (a, b) and (α, β, γ) obtained by following steps 1 to 4 described above, using $N(1) = 2048$ and $m = 4$. Each of the codes is run on each of the selected parallel architectures. Clearly the precise values obtained for (a, b) and (α, β, γ) depend upon a number of factors.

- Because users of WRG2 and WRG3 do not get exclusive access to the machines, or the Myrinet switches, there is always some variation in the solution times obtained in steps 1 and 2.
- On WRG2 there are (75) 2.4GHz and (53) 2.2GHz processor nodes, hence the parameters will depend on which processors are used to collect execution times in steps 1 and 2.
- On WRG3 the situation is made more complex by the fact that each node consists of two processors and 4 cores. Users may therefore have access to a core on a node where other users are running jobs or else they may have an entire node to themselves. This creates significant variations in the timings for both sequential and small parallel runs.

As outlined in steps 1 and 2 above, a simple way to reduce the effects of these variations is simply to take average timings over five runs (say). Such a crude approach, whilst accounting for the the relatively minor effects of sharing resources such as access to the communications technology, are not generally sufficient on their own however. For example, for a heterogeneous architecture such as WRG2, which has processors of two different speeds, it should also be recognised that a parallel job using a large number of processors will typically make at least some use of the slower 2.2GHz nodes. It is therefore essential to ensure that in step 1 a slower processor is used to compute the sequential timings, and in step 2 at least one slower processor should be used in the parallel runs. If only the faster processors are used in steps 1 and 2 above then the resulting model will inevitably under-predict solution times on large numbers of processors when any of these processors are 2.2GHz rather than 2.4GHz. This use of the slower processors has been imposed for the two WRG2 columns of Table 1.

		m2-WRG2	m2-WRG3	m1-WRG2	m1-WRG3
1 node	$a =$	0.1540	-0.1658	$-7.957e - 02$	-0.2250
	$b =$	$6.616e - 05$	$5.139e - 04$	$1.261e - 04$	$2.943e - 04$
2 nodes	$a =$		$-8.518e - 03$		-0.2316
	$b =$		$4.083e - 04$		$3.191e - 04$
1 node	$\alpha =$	-0.3825	-0.2079	$-9.245e - 02$	$6.012e - 02$
	$\beta =$	$7.863e - 04$	$5.704e - 04$	$1.434e - 04$	$-8.833e - 05$
	$\gamma =$	$-6.075e - 07$	$-4.765e - 08$	$-1.458e - 08$	$3.228e - 07$
	$\alpha =$		-0.4211		$-1.698e - 02$
2 nodes	$\beta =$		$9.621e - 04$		$3.106e - 05$
	$\gamma =$		$-4.671e - 07$		$2.430e - 07$

Table 1. Parameters (a, b) and (α, β, γ) of the T_{comm} models (4) and (5) respectively, obtained for $np = 4$ and for $N(4) = 512, \dots, 4096$ with coarse grid of size 32

Furthermore, in order to better control the effects of multiple cores on WRG3, we have chosen to undertake all of the sequential runs using four copies of the same code: all running on the same node. Again, this decision is made bearing in mind the situation that will exist for a large parallel run in which all of the available cores will be used. In addition to this, for WRG3, two sets of predictive timings are produced. The first of these is obtained by running the 4 process parallel job on a single (four core) node, whilst the second is obtained by running an 8 process parallel job across two (four core) nodes. The latter is designed to allow both intra-communication (communication between processes on the same node) and inter-communication (communication between processors on different nodes) overheads to be captured by our model (the former will only capture intra-communication costs). These two situations are denoted by “1 node” and “2 nodes” respectively in Table 1.

There are a number of interesting observations to make about the parameters shown in Table 1. As indicated in step 3 above, these are obtained by fitting curves (either linear or quadratic) through the averaged data collected in steps 1 and 2 (using (4) or (5) respectively, with $T_{comm} = T_{parallel} - T_{comp}$). For the linear model it is important to state that the fit to the data is not particularly good which provides a clear indication that the model may be deficient. For the quadratic cases the fits to the data are, perhaps not surprisingly, a lot better. For WRG3, the effect of undertaking the small parallel runs (step 2) using cores on two nodes rather than one is not particularly significant for the linear model but is much more noticeable for the quadratic model. This might suggest that other inaccuracies are dominant in the former case.

Given the values shown in Table 1, it is now possible to make predictions for the performance of the multigrid codes on greater numbers of processors. In this paper, we consider executing the codes on $np = 64$ processors (the maximum queue size available to us), with the grids scaled in size in proportion to np . Figure 3 shows results using the linear model, (4), whilst Figure 4 shows similar results based upon the quadratic model (5). In each case all four combinations of algorithms (m1,m2) and computer systems (WRG2,WRG3) are presented. Note that for the runs on WRG3, following Table 1, two predictions are presented (Tpredict1 and Tpredict2): these are based upon the best-fit parameters (α , β and γ) obtained when 1 node (4 cores) or 2 nodes (8 cores) are used in step 2 respectively.

It is clear from Fig. 3 that the linear model provides disappointing predictions in almost all cases. It is noticeable that this model under-predicts the solution times for algorithm m2 (which includes blocking as well as non-blocking communication) whilst it tends to over-predict the solution times for algorithm m1. It is hard to discern any obvious pattern from these results other than the fact that the qualitative behaviour seems to have been captured in each case, even though the quantitative predictions are unreliable. This suggests that nonlinear effects are important in the parallelization, either due to communication patterns (e.g. switch performance and/or the effects of non-blocking communication) or nonlinear cache effects (with multi-core processors for example).

From Fig. 4 it is apparent that using the quadratic model to capture the predicted nonlinear effects can be highly effective. This model provides significantly better predictions for both multigrid implementations and on both parallel architectures (provided that Tpredict2 is used on WRG3). In the case of the results obtained on WRG2 it is important to emphasize again that steps 1 and 2 of the algorithm were undertaken using at least one slower processor in each run. Without this restriction the predictions were of a much poorer quality: providing significant under-estimations of the parallel run times on 64 processors (see [12] for further details). In the case of WRG3 recall that Tpredict2 is based upon the use of 8 cores in step 2 of the methodology described in the previous section (as opposed to 4 cores for Tpredict1). This clearly demonstrates that, since the large parallel jobs typically use all of the available cores on each node, both intra- and inter-communication costs must be captured by the predictive model. Per-

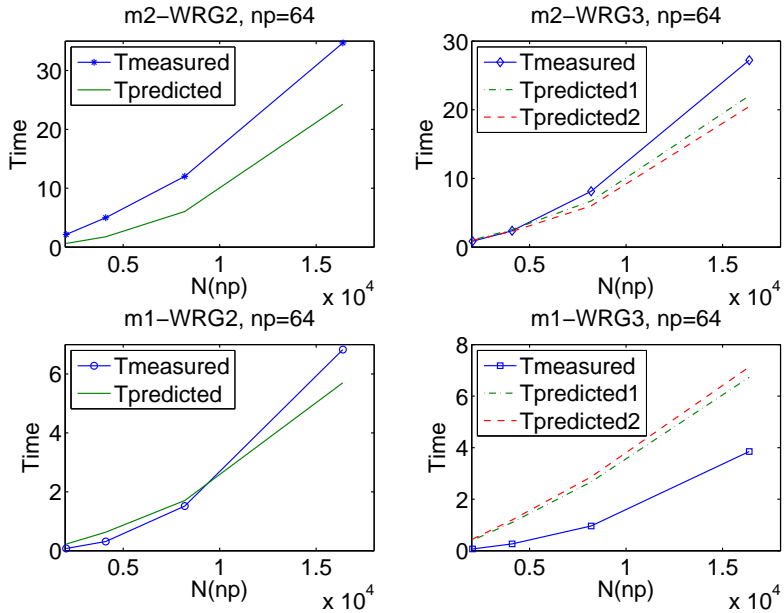


Fig. 3. Linear model $T = T_{comp} + a + bN$ for Time predicted, and Time measured (both in seconds) on $np = 64$ processors, $N(64) = 2048, \dots, 16384$

haps not surprisingly, when this model fails to capture all of the communication patterns present in the full parallel code the resulting predictions (Tpredicted1) become unreliable.

5 Conclusion and Future Work

In this paper we have proposed a simple methodology for predicting the performance of parallel multigrid codes based upon their characteristics when executed on small numbers of processors. The initial results presented are very encouraging, demonstrating that remarkably accurate and reliable predictions are possible provided that sufficient care is taken with the construction of the model and the evaluation of its parameters. In particular, it has been possible to demonstrate that the effects of both heterogeneous and multicore architectures can be captured, and that the models proposed can be applied to two quite different multigrid codes.

It is clear from the results presented in the previous section that, despite the communication costs being $O(N)$ and the $O(N)$ complexity of the multigrid approach (as illustrated in [12]), the simple linear model for the growth in the parallel overhead is not sufficient to capture the practical details of scalability to

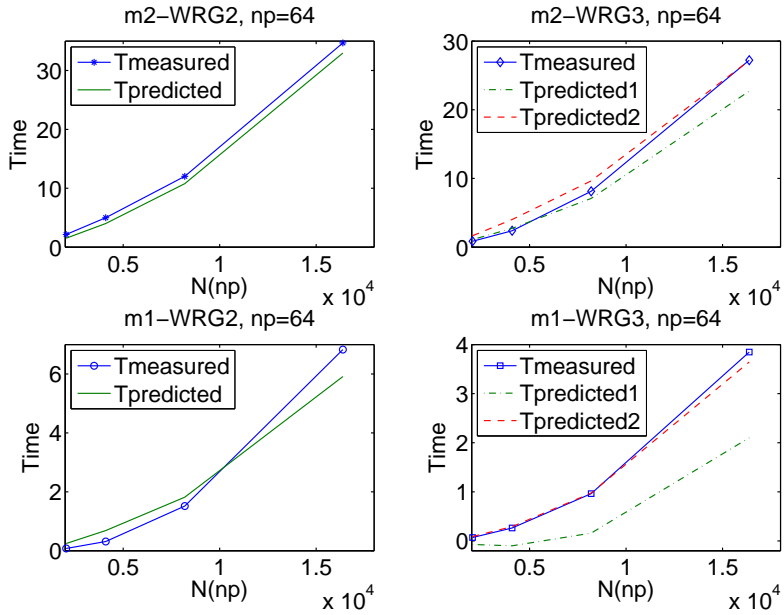


Fig. 4. Quadratic model $T = T_{comp} + \alpha + \beta N + \gamma N(1)^2$ for Time predicted, and Time measured (both in seconds) on $np = 64$ processors, $N(64) = 2048, \dots, 16384$

large numbers of processors. There are numerous possible causes of this (e.g. non-linear communication patterns, caching effects, etc.) however it is demonstrated that the addition of a quadratic term to the model, to capture the nonlinear effects to leading order, improves its predictive properties substantially. Unsurprisingly, care must be taken to deal with non-homogeneous architectures or multicore architectures in an appropriate manner. If these effects are ignored then even the quadratic model fails to yield realistic predictions.

It has been observed in this work that the quality of the best fit that is made in order to determine the values of the parameters in Table 1 appears to provide a useful indication as to the reliability of the resulting model. For example, the linear fits in the top half of the table are relatively poor, as are the predictions in Fig. 3. In future work it would be interesting to investigate this phenomenon further in order to attempt to produce a reliability metric for the predictions that are made. Recall that one of the primary motivations for this work is to provide information on expected run-times on different numbers of processors on different architectures in order to allow optimal (or improved) scheduling of jobs in a Grid-type environment where a variety of potential resources may be available. Clearly, providing additional information, such as error bounds for these expected run times, will further assist this process. An additional factor

that should also be included in this modelling process is the ability to consider different domain decomposition strategies (e.g. partitioning the data into blocks rather than strips, [6]) and predict their relative performance for a given problem on a given architecture.

Finally, we observe that, in addition to deciding which single computational resource to use in order to complete a given computational task, there will be occasions when multiple Grid resources are available and might be used together. Consequently, in future work we also intend to extend our models to provide predictions that will allow decisions to be made on how best to split the work across more than one resource and to determine the likely efficiency (and cost-effectiveness) of so doing.

References

1. Brandt, A. Multi-level adaptive solutions to boundary value problems. *Mathematics of Computation* **31**, 333–390 (1977).
2. Briggs, W.L., Henson, V.E. and McCormick, S.F. *A Multigrid Tutorial*. SIAM (2000).
3. Carrington, L., Laurenzano, M., Snively, A., Campbell, R., Davis, L.P. How well can simple metrics represent the performance of HPC applications?, In *Proceedings of SC—05* (2005).
4. Dew, P.M., Schmidt, J.G., Thompson, M. and Morris, P. The White Rose Grid: practice and experience. In: Cox, S J (editor) *Proceedings of the 2nd UK All Hands e-Science Meeting*. EPSRC (2003).
5. DIMEMAS: <http://www.cepba.upc.es/dimemas/>
6. Goodyer, C.E. and Berzins, M. Parallelization and scalability issues of a multi-level elastohydrodynamic lubrication solver. *Concurrency and Computation: Practice and Experience* **19**, 369–396 (2007).
7. Huedo, E., Montero, R.S. and Llorente, I.M. A module architecture for interfacing Pre-WS and WS Grid resource management services. *Future Generation Computing Systems* **23**, 252–261 (2007).
8. Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., and Gittings M. Predictive Performance and Scalability Modeling of a Large-Scale Application, In *Proceedings of SC2001* (2001)
9. Koh, Y.Y. *Efficient Numerical Solution of Droplet Spreading Flows*. Ph.D. Thesis, University of Leeds (2007).
10. Krauter, K., Buyya, R. and Maheswaran, M. A taxonomy and survey of Grid resource management systems. *Int. J. of Software: Practice and Experience* **32**, 135–164 (2002).
11. Lang, S. and Wittum, G. Large-scale density-driven flow simulations using parallel unstructured grid adaptation and local multigrid methods. *Concurrency and Computation: Practice and Experience* **17**, 1415–1440 (2005).
12. Romanazzi, G. and Jimack, P.K. Performance prediction for parallel numerical software on the White Rose Grid. In *Proceedings of UK e-Science All Hands Meeting*. 2007.
13. Trottenberg, U., Oosterlee, C.W. and Schüller, A. *Multigrid*. Academic Press (2003).