

Performance Prediction for Multigrid Codes Implemented with Different Parallel Strategies

Giuseppe Romanazzi¹ and Peter K. Jimack²

¹CMUC, Departamento de Matematica, Universidade de Coimbra, Coimbra
3001-454, Portugal

²School of Computing, University of Leeds, Leeds LS2 9JT, UK

Keywords: Parallel Distributed Algorithms, Performance Evaluation and Prediction, Multigrid Numerical Software.

We propose a model for describing and predicting the parallel performance of multigrid numerical software on distributed memory architectures for which different data partitioning and mapping strategies may be used. 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. Despite its relative simplicity the model is shown to be accurate and robust with respect to both the parallel architectures and the data partitioning strategies that are used.

Keywords: Performance Evaluation and Prediction, Parallel Distributed Algorithms, Multigrid Numerical Software.

1 Introduction

Computational science and engineering research is of growing importance across a large range of application sectors and frequently research teams have potential access to a wide variety of computational resources. consequently, when a computationally demanding numerical simulation is to be undertaken there is frequently a choice to be made as to the most appropriate resource on which to execute the task. The optimal selection of resource will depend upon a number of factors such as availability, cost, turn-around time, etc. In order to make informed decisions therefore it is desirable that accurate predictions can be made in advance as to the execution time of a given job on a given computer system. An important feature of this decision-making process is that different parallelization strategies are generally possible for typical numerical codes, and so it is also necessary to be able to predict the performance of a given code on a given architecture for a selection of different partitions and mappings of

the computational data. This is because, depending on the particular problem and hardware combination different geometric partitionings of the computational work can show better performance than others, see for example [5]. In order to make such decisions in a reliable way therefore, it is necessary that computational researchers are able to predict the performance of their software across different combinations of these resources and parallelization strategies.

We propose a methodology for describing and predicting the performance of parallel numerical software that can be implemented with different geometric partitionings of the computational work within a distributed memory architecture. The methodology is tested for a particular class of numerical codes, based upon the multigrid solution of discretized partial differential equations, [1]. Our goal 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 parallel strategies, and the parallel architectures considered.

2 Background

There is a large body of related work into performance modelling [8] that varies from analytical models designed for a single application through to general frameworks that can be applied to many applications on a large range of high performance computing (HPC) systems. For example, in [9] detailed models of a particular application are built for a range of target HPC systems, whereas in [10] or [11] an application trace is combined with some benchmarks of the HPC system that is being used in order to produce performance predictions.

Both approaches have been demonstrated to be able to provide accurate and robust predictions, although each has its potential drawbacks: significant code-specific knowledge being required for deriving the analytic models, whereas the trace approach may require significant computational effort. Moreover, in the former approach, when a different HPC system is used it would generally be necessary to change the model, adding new parameters for example. Instead, in the latter, we need to add or to find new benchmarks when a new code is used. Considering these limits, the choice between the two approaches can depend also on other factors. For example, when it is more important to predict the run-time of a large-scale application on a given set of systems, as opposed to comparing the performance of the systems in general, researchers (like those in the LANL group [9]) prefer to study deeply their application in order to obtain its own analytic model for the available set of HPC systems. On the other hand, when it is more interesting to compare performances of different machines on some real-applications, the latter approach is preferable; in that case different benchmark metrics can be used and convoluted with the application trace file.

Our approach lies between these two extremes. We use relatively simple analytic models (compared to the LogP model [12] for example), that are applicable to a gen-

eral class of multigrid algorithms and then make use of a small number of simulations of the application on a limited number of CPUs of the target architecture in order to obtain values for the parameters of these models. Predictions as to performance of the application on larger numbers of processors may then be made.

3 Previous Research and Parallel Strategies

In our recent work [2, 3, 4], we use a prediction model that accurately describes the performance of different multi-level parallel numerical codes within a multi-cluster environment. The model is based on the assumption that a symmetric partition by rows of the computational domain is used. In this case each processor can send and receive MPI messages only to and from the neighbouring processors immediately above and below it in the partition, see Figure 1.

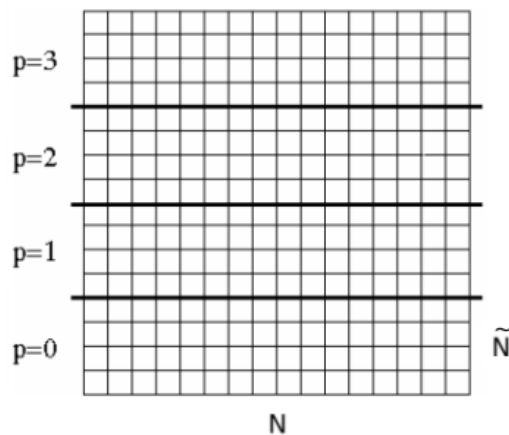


Figure 1: Parallel distribution of the grid nodes in a strip partitioning strategy. Each processor sends and receives messages of length N and has a computational load that is proportional to $N \times \tilde{N}$

The model used in this earlier work is based on the relation

$$T_{parallel} = T_{comp} + T_{comm},$$

where T_{comp} is the computational time on the slowest processor and T_{comm} is the total parallel overhead, primarily due to inter-processor communications. The calculation of T_{comp} is relatively straightforward since it simply requires the execution of a properly scaled problem on a single processor: 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 provides the same memory access and contention patterns, relating to cache and multicore effects for example, as those experienced in the parallel runs.

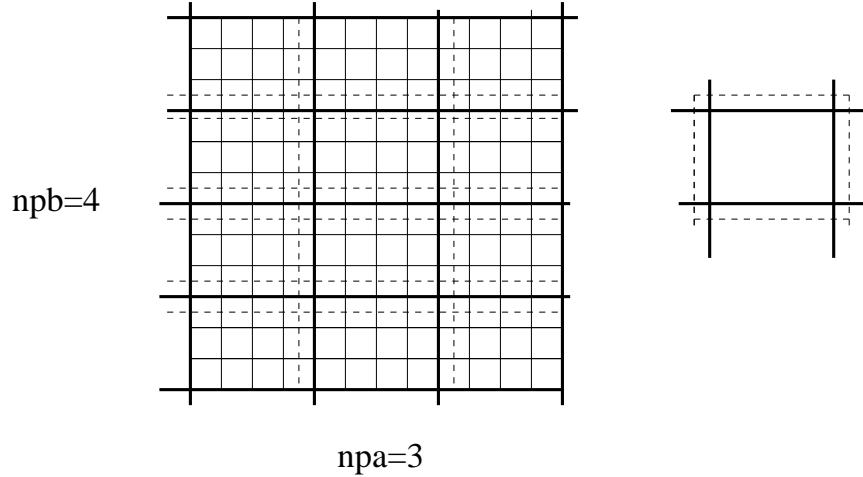
The calculation of T_{comm} is rather more complex than that for T_{comp} however. The methodology proposed in [2, 3] is based on a limited sequence of parallel runs across a limited number of parallel resources. The overhead measured on these parallel runs is used to extrapolate information for the overhead expected when a larger number of processor are used. The comparison of the parallel times measured with respect to those predicted shows a high accuracy (an error below 10%) for each combination of parallel architecture and numerical code used. In our recent work [4], we have exploited this methodology for predicting the performance of a parallel engineering multi-level numerical code. Also in this situation an extrapolation of the parallel overhead based upon a few runs of the code on a small number of processors has shown its reliability.

This reliability has been confirmed for a number of different codes based upon the multilevel solution of discretized partial differential equations (PDEs) [2, 3, 4], with each code involving slightly different parallel implementations. In each case however, the partitioning strategy used is based upon grouping together consecutive rows as illustrated in Figure 1. The differences in the parallel implementations tend to come about at the interface between subdomains where there may be additional “ghost” rows in order to reduce the frequency of communication but at the expense of some additional computational work per processor. In this work we have focused exclusively on a single software implementation in the first instance (the code *m1* in [2]) which also allows the partition of the computational mesh into blocks as well as strips. This is illustrated in Figure 2 for a partition into 12 subdomains. The figure also seeks to illustrate the use of ghost rows within the code *m1*, where each subdomain is augmented by a ghost column and a ghost row for each of its left-right and above-below neighbours respectively.

Note that the use of ghost rows means that slightly different quantities of computational work are assigned to each processor depending upon the partitioning strategy that is used. For example, as shown in Figures 2 and 3, even when the block sizes are the same, having different numbers of neighbours can mean a different amount of computational work per processor and a different maximum amount of work per processor in particular.

The model described in our previous work, for both T_{comp} and T_{comm} is no longer sufficient for predicting the performance of these more general numerical codes for which different partitioning strategies, such as block partitioning, are implemented in order to improve the performance of the parallel run. However, starting from our previous work it is possible to obtain a model that can accurately describe the communication patterns for different parallel block partitioning strategies, as demonstrated in the following sections.

$nprocs=12$



$nprocs=1$

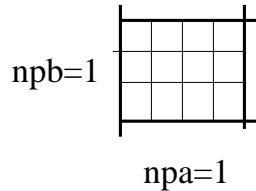


Figure 2: Parallel distribution of the grid nodes in a block partitioning strategy. Each computational node lies at the vertices of the grid however the dashed lines represent additional ghost rows or columns added for efficient parallel implementation. Each internal block in the partition has two ghost rows and two ghost columns in addition to its internal unknowns, see the two diagrams at the top (case $nprocs = 12$, $npa = 4$, $npb = 3$). The diagram at the bottom represents the computational grid when the code is implemented on a single processor ($nprocs = 1$) for which no ghost lines are used.

4 The Predictive Model

We describe a model for predicting the performance of parallel runs of multigrid codes when used to solve a large “target” problem across np processors. This target problem is defined on 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. In particular, we consider the mesh to be mapped onto np processors as a bi-dimensional grid $np_a \times np_b$, with

$$np = np_a \cdot np_b, \quad \frac{N_a}{np_a} = \frac{N_b}{np_b},$$

see Figures 2 and 3. As in these figures, 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 com-

nprocs=12

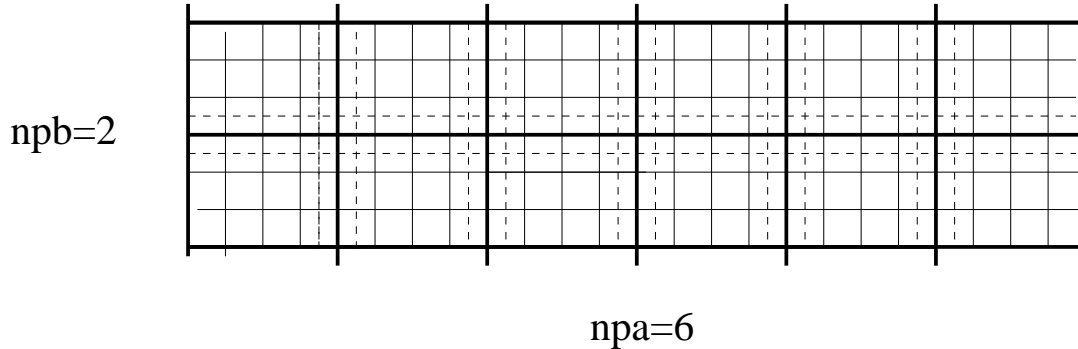


Figure 3: Parallel distribution of the computational work across a grid 6×2 of 12 processors. Note that the partitioning strategy has changed from that of Figure 2 even though the number of processors is unchanged.

putational mesh.

Following our previous work, [2, 3, 4], the first 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 $\widetilde{N}_a \times \widetilde{N}_b$ 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} represents all of the parallel overheads (primarily due to inter-processor communications). The exact shape of the computational domain per processor may vary, as described in the previous section, with respect to the partitioning strategy used. The MPI parallel code *m1* (see [2] for details) 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 Figure 2. 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 [2, 3] 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 Figure 4. 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).

The communication phase consists of a sequence of sends and receives between neighbouring processors through the use of the 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 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 our previous work [3], the overhead in a strip partition, where the processors are in a single column (or row), 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

$$T_{comm} = \max(T_{comm(np_a,1)}, T_{comm(1,np_b)}). \quad (2)$$

We remark however that equation (2) 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 an $\widetilde{N}_a \times \widetilde{N}_b$ 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

$$T_{EXTRA} = T_{(2,2)} - T_{comp} - \max(T_{comm(2,1)}, T_{comm(1,2)}). \quad (3)$$

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

$$T_{comm} = T_{EXTRA} + \max(T_{comm(np_a,1)}, T_{comm(1,np_b)}) \quad (4)$$

Hence using (1), (4) and (3)

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

Finally we have

$$T \approx T_{(2,2)} + \max(T_a, T_b), \quad (5)$$

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

Based upon the evidence of [4] both overheads T_a and T_b are assumed to satisfy a relation of the following type:

$$T_a = \alpha(np_a) + \gamma(np_a) \cdot work; \quad (6)$$

$$T_b = \alpha(np_b) + \gamma(np_b) \cdot work. \quad (7)$$

In (6), (7) 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 $(np_a, 1)$ or 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*.

Furthermore, following [4], we assume that the following relations are valid:

$$\alpha(np) \approx c + d \log_2(np) + e \log_2(np)^2, \quad (8)$$

$$\gamma(np) \approx \text{constant}. \quad (9)$$

This assumption is discussed in detail in [4], where only partitions by strips are considered. In practice, in order to evaluate the parameters c, d, e and γ , we use measurements taken for $np = 4, 8, 16$ and pose: $\gamma = \gamma(16)$; whilst c, d and e are obtained using a simple linear fit through the three data points $(2, \alpha(4))$, $(3, \alpha(8))$ and $(4, \alpha(16))$.

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 $(np * \frac{\widetilde{N}_a}{l}) \times \widetilde{N}_b$ 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. (6) or (7) (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 straight line, as in Eq. (8), through the points $(2, \alpha(4))$, $(3, \alpha(8))$ and $(4, \alpha(16))$ to estimate c, d and e based upon Eq. (8): now compute $\alpha(np)$ for the required choice of np .

5. Use the model in Eq. (6) to estimate the values of T_a (and use the model in Eq. (7) 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. (5).

5 Numerical Results

The model described in the previous section is now used to predict the performance of the multigrid numerical code `m1` [2], running on two different clusters of the White Rose Grid [13] environment.

- Cluster A 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 in the tests to connect the nodes.
- Cluster B 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 access to 8GBytes of physical memory. Again, Myrinet switching is used.

A selection of typical test results, using Clusters A and B, are shown in Tables 1 and 2, respectively. The first three columns of both tables describe the target problem: the total number of processors, the grid of processors/partition geometry, and the overall problem size $N_a \times N_b$. The fourth column shows the memory required by each processor associated with each target problem: we use 2GB for cluster A and 1GB for Cluster B. Using the methodology described in the previous section, we predict the code's performance for each of these target configurations. The actual measured times once the target problems are run, the predicted times and the resulting errors are then listed in the last three columns of each table.

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.

Table 1: Measurements and predictions for Cluster A (both quoted in seconds)

np	(np_a, np_b)	size	mem. per proc.	meas.	predict.	$\ error\ $
64	(8, 8)	65536×32768	1GB	378.36	358.89	5.1%
64	(4, 16)	32768×65536	1GB	370.16	353.57	4.5%
64	(2, 32)	16384×131072	1GB	315.82	320.05	1.3%
32	(8, 4)	65536×16384	1GB	372.94	358.89	3.7%
32	(4, 8)	16384×65536	1GB	373.11	353.57	5.2%

Table 2: Measurements and predictions for Cluster B (both quoted in seconds)

np	(np_a, np_b)	size	mem. per proc.	meas.	predict.	$\ error\ $
128	(16, 8)	131072×65536	2GB	522.83	519.81	0.58%
128	(8, 16)	65536×131072	2GB	493.71	506.43	2.6%
128	(32, 4)	262144×32768	2GB	533.86	504.85	7.9%
128	(4, 32)	32768×262144	2GB	512.18	507.28	0.96%
64	(8, 8)	65536×65536	2GB	510.59	506.43	0.81%
64	(16, 4)	131072×32768	2GB	478.07	519.81	8.7%
64	(4, 16)	32768×131072	2GB	507.28	474.76	6.4%
64	(32, 2)	262144×16384	2GB	564.54	533.86	5.4%
64	(2, 32)	16384×262144	2GB	534.80	496.75	7.1%
32	(8, 4)	65536×32768	2GB	481.36	506.43	5.2%
32	(4, 8)	32768×65536	2GB	507.28	495.66	2.3%

6 Discussion

In this paper we have proposed a simple methodology for predicting the performance of parallel multigrid codes across different parallel architectures based upon distributed memory. The philosophy upon which our methodology is based is to produce a general empirical model that involves a minimal number of parameters, and then to determine appropriate values for these parameters for any given combination of code, partition and hardware resources. These parameter values are determined based upon the characteristics of the code when it is executed on much smaller numbers of processors than are ultimately required. This allows resources that are not currently available to be reserved for future execution based upon the predicted need. Results presented in the previous section demonstrate that the methodology is both robust and accurate across the two parallel architectures considered, as well as coping satisfactorily with different partitioning strategies and different problem sizes, at least for the multigrid code so far considered.

Although the results presented in this work are encouraging, it is clear that when the elapsed times for different partitioning strategies are very close the methodology is unable to predict these differences reliably. The immediate future work that we propose is to test the methodology on a broader selection of codes to assess the generality of our approach to wider classes of partitioning strategy and parallel software.

Acknowledgements

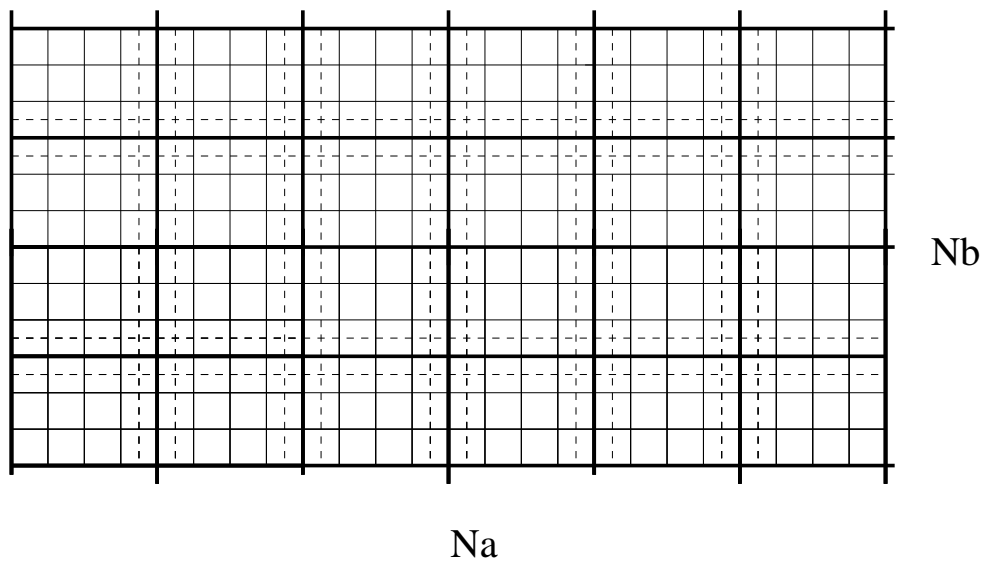
This work was supported by EPSRC grant EP/C010027/1.

References

- [1] U. Trottenberg, C.W. Oosterlee, and A. Schüller, *Multigrid*, Academic Press (2003).
- [2] G. Romanazzi and P.K. Jimack, “Parallel performance prediction for multigrid codes on distributed memory architectures”, in *High Performance Computing and Communications (HPCC-07)*, R. Perrott et al. (editors), LNCS 4782, Springer, 2007, pp. 647–658.
- [3] G. Romanazzi and P.K. Jimack, “Parallel performance prediction for numerical codes in a multi-cluster environment”, in *Proceedings of the 2008 International Multiconference on Computer Science and Information Technology (IMCSIT’08)*, M. Ganzha et.al. (editors), PTI Press, Katowice, Poland, pp. 467–474, 2008.
- [4] G. Romanazzi, P.K. Jimack and C.E. Goodyer, “Reliable performance prediction for parallel scientific software in a multi-cluster gridenvironment”, in *Proceedings of the Sixth International Conference on Engineering Computational Technology*, M. Papadrakakis and B.H.V. Topping (editors), Civil-Comp Press, Paper 4, 2008.
- [5] N.G. Pantelelis and A.E. Kanarachos, “The parallel block adaptive multigrid method for the implicit solution of the Euler equations”, *Int. J. Numer. Meth. Fluids*, vol. 22, 1996, pp. 411–428.
- [6] C. E. Goodyer, and M. Berzins, “Parallelization and scalability issues of a multilevel elastohydrodynamic lubrication solver,” *Concurrency and Computation*, vol. 19, 2007, pp. 369–396.
- [7] P. Ladeveze, A. Nouy, and O. Loiseau, “A multiscale computational approach for contact problems”, *Comput. Meth. Appl. Mech. Engrg.*, vol. 191, 2002, pp. 4869–4891.
- [8] S. Pllana, I. Brandic and S. Benkner, “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)*, vol. 4, 2008, pp. 17–26.
- [9] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman and M. Gittings, “Predictive performance and scalability modeling of a large-scale application”, in *Proceedings of SuperComputing 2001*, 2001.
- [10] G. Rodriguez, R. M. Badia, and J. Labarta, “Generation of simple analytical models for message passing”, in *Euro-Par 2004 Parallel Processing*, M. Danelutto et al. (editors), LNCS 3149, Springer, 2004, pp. 183–188.

- [11] L. Carrington, M. Laurenzano, A. Snavely, R. Campbell and L. Davis, “How well can simple metrics represent the performance of HPC applications?”, in *Proceedings of SuperComputing 2005*, 2005.
- [12] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian and T. von Eicken, “LogP: towards a Realistic Model of Parallel Computation”, *SIGPLAN Not.*, vol. 28, 1993, pp. 1–12.
- [13] P. M. Dew, J. G. Schmidt, M. Thompson, and P. Morris, “The White Rose Grid: practice and experience,” in *Proceedings of the 2nd UK All Hands e-Science Meeting*, S.J. Cox (editor), EPSRC, 2003.

(np_a, np_b) grid



$(2,2)$ grid

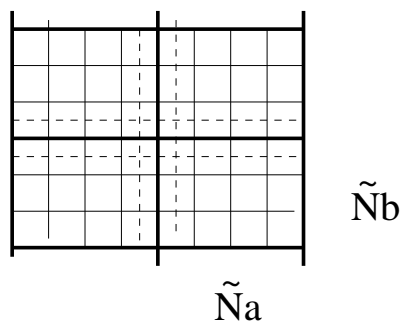


Figure 4: Example of a computational grid for a target problem with a (np_a, np_b) partition and an equivalent sized problem for the $(2, 2)$ processor grid, as used in the predictive methodology. Note that each of the processors in both problems have the same work except for the variation in the number of ghost rows and columns.