

# Asynchronous parallel solvers for linear systems arising in computational engineering

Peter K. Jimack and Mark A. Walkley

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

## Abstract

Modern trends in Computational Science and Engineering are moving towards the use of computer systems with ever increasing numbers of computational cores. A consequence of this is that over the next decade it will be necessary to develop and apply new numerical algorithms that are far more scalable than has historically been required. Ideally, such algorithms will be able to exploit many thousands of cores in an efficient manner in order to be able to tackle the most computationally demanding problems. This chapter explores these challenges in the context of the solution of large systems of algebraic equations arising from the discretization of partial differential equations. Particular emphasis is placed on the need to develop asynchronous and fault tolerant software that is built upon the most efficient underlying methods, such as multigrid and other multilevel algorithms.

**Keywords:** computational engineering, parallel processing, scalable parallel solution, asynchronous algorithms, multilevel methods.

## 1 Introduction

This chapter considers the parallel solution of linear systems of equations that arise from the discretization of one or more partial differential equations (PDEs). The motivation for this exposition is our desire to consider (i) very highly scalable parallelism that will be capable of exploiting future trends in multicore hardware, and (ii) the use of the most efficient numerical algorithms. The latter point is based upon the observation that the easiest algorithms to implement in parallel are not always the best algorithms in terms of their accuracy or their speed of convergence.

In the remainder of this introductory section we provide a brief summary of current hardware trends and also an overview of some of the key numerical algorithms

for the solution of linear systems, especially those arising from the discretization of PDEs. This overview considers some popular existing parallel implementations and discusses some of the main strengths and weaknesses of each. Particular attention is drawn to the use of multilevel algorithms, such as multigrid for example, due to their ability to deliver optimal time complexity for certain classes of problem. Section 2 then considers a typical parallel implementation of a multigrid solver in more detail. The parallel performance of such a solver is assessed and the need for significant improvements and developments is discussed. In particular, it is noted that the order in which operations are completed is of great importance and so the issues of synchronization and fault tolerance arise.

In Section 3 the fundamentals of asynchronous algorithms are introduced. There is a long and rich history of the development and analysis of asynchronous techniques and so this section is necessarily quite high level in its approach. Nevertheless, the aim is to convey some of the main classes of asynchronous methods and the theory that lies behind them. Section 4 then focuses more specifically on the use of asynchronous methods in the solution of linear systems of equations arising from the discretization of PDEs, with particular emphasis on multilevel methods. Finally, Section 5 presents a short summary and outlook for further research.

## 1.1 Recent hardware trends

As we approach the end of the first decade of the Twenty-First Century, we may observe some significant changes in the ways in which computer hardware, and processing units in particular, are developing. For the previous five decades we have benefited from a year-on-year increase in processor speeds that have allowed our software to run ever-faster without the need for algorithmic developments. Of course such developments have taken place but these have always served to provide *additional* speed and efficiency. For example the use of parallel computer architectures has typically allowed problems to be solved 100-200 times faster (say) than on a single processor, however an alternative to such algorithms and software development has always been to wait about a decade for this same speed-up to be delivered by increased CPU performance alone.

Today this is no longer the case. The speed of individual processing units has stopped increasing, and the need to operate in a more energy-efficient manner has even tended to cause processor speeds to decrease a little. Hence, the current trend is to seek to increase performance by increasing the number of cores per processor. These may be standard multicore architectures or more specialist processing units such as GPUs. In either case (or even with mixed architectures) it is clear that to maintain increases in performance over the coming decade it will be necessary, rather than optional, to fully exploit concurrency. Furthermore, to undertake state-of-the-art computations will require the effective use of large numbers of multicore units, potentially utilizing millions of cores on a single task.

To illustrate that this last claim is realistic it is informative to consider the World's

fastest supercomputers over the past decade<sup>1</sup>. In November 2010 there are 11 computers on the *Top500* list that have more than  $10^5$  cores, with the number one on the list (based at the National Supercomputing Center in Tianjin, China) achieving 2566 PFlops across 186368 GPU cores. Second on the list uses 224162 conventional cores (based around 6-core 2.6GHz processors) to achieve a performance of 1759 TFlops. Just five years previously (November 2005) only 1 system had more than  $10^5$  cores, whilst a decade before (November 2000) no system even had  $10^4$  cores (and only 18 systems had over  $10^3$  cores). In this ten year period the processor speeds have increased by a factor of less than 10, whilst the maximum performance has increased by over 500 times (from less than 5 TFlops to 2566 TFlops) as a result of the growth in the number of cores. Given that the fastest system in November 2000 would not get close to the bottom of the November 2010 *Top500* list (which is over 6 times faster in terms of maximum performance) it seems reasonable to assume that scalability to  $10^6$  cores or more will be a necessity for high performance computing in the coming decades.

Of course the benchmarks used to assess performance for the *Top500* list are based upon dense matrix operations. It is clear that many applications are not so naturally applicable to very large scale parallelism, and this particularly includes the computational solution of PDEs using techniques such as finite difference or finite element methods. In the next subsection we consider algebraic systems that arise from such discretizations.

## 1.2 Solution of linear systems

Discretization of a PDE ultimately leads to the need to solve one or more linear systems of algebraic equations. If problems are time dependent then the use of implicit time stepping leads to algebraic systems at each time step, whereas only a single system needs to be solved for a steady-state problem. If the PDE is nonlinear then the resulting algebraic equations will be nonlinear too, however some linearization of these systems is typically undertaken, yielding a sequence of linear systems to be solved.

Depending upon the discretization method that is used the resulting system(s) of linear algebraic equations may have one of a number of different properties. For example, using a finite element method on an unstructured grid yields a general sparse matrix system, where the sparsity pattern depends upon the geometry of the grid and the order of the finite elements used. Alternatively, application of finite differences on a regular lattice leads to a banded system of equations, where the bandwidth depends on the ordering of the unknowns and, most importantly, on the spatial dimension of the problem. Other techniques, such as boundary element methods for example, result in dense systems of algebraic equations. The focus of this chapter is on algorithms that are applicable to sparse or banded systems arising from finite element or finite difference discretizations of PDEs. For such systems there are a number of possible solution techniques. These include the use of banded or sparse direct solvers [1, 40],

---

<sup>1</sup>Source: [www.top500.org](http://www.top500.org)

the use of preconditioned iterative solvers [46, 25], and/or the use of more advanced techniques such as domain decomposition [34, 8] and multilevel methods [13, 47].

Banded solvers are often attractive for two-dimensional problems where, on an  $N^{\frac{1}{2}} \times N^{\frac{1}{2}}$  spatial grid, the half bandwidth of the linear system is approximately  $N^{\frac{1}{2}}$ . In three dimensions however, on an  $N^{\frac{1}{3}} \times N^{\frac{1}{3}} \times N^{\frac{1}{3}}$  spatial grid, the half bandwidth is  $N^{\frac{2}{3}}$ . This makes banded methods less attractive, especially since the vast majority of entries within the band are zero, and so could be ignored by a more sophisticated solver.

A sparse direct solver is, as the name suggests, a direct method such as LU factorization, but implemented in such a way as to attempt to minimize fill-in and to use a data structure which only stores the non-zero entries that arise. In the case of a symmetric positive-definite system (which typically arises following the discretization of a self-adjoint elliptic operator) it is known that pivoting is not required to ensure numerical stability. Hence an ordering of the unknowns may be selected with the sole aim of minimizing the fill-in (or, for a parallel implementation, minimizing the inter-processor dependencies). For all other systems however, the ordering must be based upon a trade-off between that required for computational efficiency and that required to maintain numerical stability. A number of sparse direct software libraries have been developed in recent years, including those targeted at parallel systems [1, 40].

It seems unlikely that sparse direct solvers will be able to scale to tens or hundreds of thousands of cores in the foreseeable future. This is partly due to the complexity of these algorithms and partly due to the strict order in which operations must occur, as well as the exact (as opposed to iterative) nature of these algorithms (making them intolerant of faults such as dropped packets of data between processors). Perhaps more likely candidates for the basis of highly scalable software are iteration-based methods.

Simple fixed point iterations, such as Jacobi or Gauss-Seidel iteration for example, are typically relatively straightforward to implement efficiently in parallel. Furthermore, they offer great potential for extreme scalability due to their ability to be modified into asynchronous methods, as outlined in Section 3 below. From a computational efficiency perspective however they are unacceptably slow to converge (and only converge at all for restricted classes of problem). Hence any practical, highly scalable, iterative algorithm must be based upon more advanced techniques such as Krylov subspace methods [46]. Even then, to make such techniques truly competitive as the level of mesh refinement (for finite difference or finite element schemes) increases, it is essential to include an effective preconditioner [25].

A number of possible preconditioning strategies are available. These may be divided into at least two classes: algebraic approaches and techniques based upon knowledge of the underlying differential operator that led to the discrete system. Examples of the former include incomplete factorization, [36, 42], and sparse approximate inverse (SPAI) methods, [30, 35]. The incomplete factorization approach is similar to a sparse direct method however not all of the fill-in that should occur is actually permitted, and so the factorization is not exact. Often however, this incomplete factorization makes an effective, and relatively cheap, preconditioner. (Note that allowing all of

the necessary fill-in leads to a perfect preconditioner but at great expense, whereas not allowing any fill-in to occur yields a preconditioner that is cheap to apply but is not necessarily that effective: the optimal choice typically lies between these two extremes.) Similarly, SPAI methods are based upon the assumption that the inverse of a sparse matrix, although not sparse itself, may have a sparse approximation which acts as a good and cost-effective preconditioner. Again, the art is in the construction of the SPAI matrix and, as with direct methods, it is not easy to achieve high parallel efficiencies for these approaches.

For these reasons, in this work we are primarily concerned with the use of preconditioners which exploit knowledge of the underlying PDE(s). Examples include domain decomposition and multigrid methods. In both cases the techniques seek to reduce the low frequency components of the error very rapidly by working on a much coarser discretization of the PDE than is required for the true approximation [55]. Algebraic versions of multigrid methods also exist, based upon the same multilevel principal but without the need to explicitly construct a hierarchy of discretizations of the PDE being solved, [23, 12, 49]. Note that although we may introduce these techniques here in the context of preconditioners for Krylov subspace solvers, they can often be applied as iterative methods in their own right.

Whilst the coarse grid approximation is a core ingredient of domain decomposition and multigrid approaches, it is this same coarse level step that is one of the major constraints in terms of parallel implementation. In particular, if we attempt to implement this in parallel then we are almost certain to have a very poor computation to communication ratio for this step. Furthermore, for algorithms such as multigrid the order in which this coarse grid solve occurs is highly significant, and so there are numerous synchronization points that cannot be crossed independently by different execution threads. These issues are discussed in more detail in the next section, whilst in Section 4 some techniques for overcoming such restrictions are discussed.

## **2 A Parallel Multigrid Algorithm**

In this section we describe a typical parallel implementation of an optimal multigrid algorithm. The algorithm is described in a reasonable level of detail so that subsequent observations regarding its parallel performance and scalability are can be made. In particular, we consider issues associated with synchronization and fault tolerance.

### **2.1 Algorithm overview**

The scalar multigrid algorithm was founded on analysis of simple iterative methods, such as Jacobi, showing that their convergence properties could be related to the Fourier components of the error [14, 33, 51, 53]. In particular a single grid iterative method cannot efficiently reduce the full spectrum of error components. The multigrid approach defines a recursive sequence of iterative methods on successively

coarser grids. Through approximating the problem at multiple scales a broader spectrum of error components can be efficiently reduced.

The first stage of the algorithm is to define a sequence of grid levels, the simplest such sequence being a uniform hierarchy of nested grids with mesh size  $h$ ,  $2h$ ,  $4h$ , etc. On any grid in this sequence the standard components of the basic multigrid algorithm are as follows.

1. Right-hand side of system received from next finer level (original right-hand side used on the finest level).
2. Solution update: usually undertaken via a small number of sweeps of an iteration such as Jacobi or Gauss-Seidel.
3. Residual computation: using the latest solution estimate.
4. Information transfer to the next coarser grid level: the residual is restricted either by injection or a weighted scheme and passed as the right-hand side to the next level.
5. Correction to the solution is received from the coarser level: this is computed by this same algorithm applied recursively, with an exact solve on the coarsest grid level.
6. Information transfer from the next coarser grid level: the correction is interpolated to the current level and used to update the solution.
7. Solution update: again undertaken via a small number of sweeps of an interaction such as Jacobi or Gauss-Seidel.

Each of the above operations can be implemented as a sweep across the entire set of grid points on that level. Hence the simplest way in which to parallelize the procedure is to only implement the parallelism *within* each of the steps above - whilst keeping the sequence of steps and grid levels unchanged.

Intuitively, such a parallel implementation is most easily achieved through a partition of the physical space across the set of available processors. In this case each processor is responsible for the set of nodes that corresponds to each of the nodes within its own physical region, on each grid in the sequence (see, for example, [28]). This leads to a simple implementation but, as is shown in the next section, sub-optimal scalability. This drawback has been well-documented, see for example the review of Jones and McCormick [37], but is also difficult to circumvent while maintaining the optimal multigrid performance.

There are two straightforward choices for the space partitioning: strip-based, in which space is evenly divided in only one coordinate direction; and block-based, in which space is evenly divided in every coordinate directions. Only strip-based partitioning is implemented here, but the relative merits of both approaches are discussed in [45] and, more briefly, in the following subsections.

## 2.2 Parallel performance

The algorithm implemented here solves the two-dimensional Poisson Equation on a rectangular domain using a nested sequence of regular finite difference approximations. Dirichlet boundary conditions are assumed. Since the focus here is on the parallel implementation it is not necessary at this point to consider more general mathematical forms or numerical methods. The problem to be solved is defined by several parameters:

- The number of processors  $p$ ;
- The grid dimension  $N$ , where we assume  $N$  is a power of 2 (the resulting discrete problem is based upon an  $N \times N$  grid of points with  $N^2$  unknowns);
- The finest grid  $f$  where  $N_f = 2^f$ ;
- The total number of grids in the multigrid process,  $nGrid$ ;
- The number of V-cycles used,  $nCycle$ ;
- The number of pre- and post-smoothing iterations used ( $nPre$  and  $nPost$  respectively).

We measure both weak and strong scalability of our parallel implementation of the algorithm. For weak scalability we fix the problem size per processor and scale up the number of processors and the problem size together, whereas for strong scalability we fix the overall problem size whilst the number of processors is increased. The corresponding parallel efficiencies for weak and strong scaling may be defined, respectively, as follows:

$$e_w(p) = \frac{T(p=1)}{T(p)} \quad \text{and} \quad e_s(p) = \frac{T(p=1)}{p T(p)}, \quad (1)$$

where  $T(p)$  is the computation times on the given number of processors  $p$

Two sets of computational experiments are described. The first of these is based upon a network of desktop workstations whilst the second is based upon a tightly coupled parallel cluster with a dedicated Infiniband interconnect. For the strong scalability experiment

$$f = 12, \quad nGrid = 7, \quad nCycle = 40, \quad nPre, nPost = 3, 2.$$

For the weak scalability experiment we fix the effective nodes per processor (work) at  $2^{10}$  and use the same multigrid control parameters.

Results for the network cluster are shown in Table 1. Note that, although the computers used for these experiments within the are quad-core, only 1 process was assigned to each machine in order to produce a genuinely distributed computation. It is

clear from these results that parallel efficiency is lost quite rapidly as  $p$  increases. The major reason for this is that the communication cost is large relative to the computation cost. Each processor performs  $\mathcal{O}(\frac{N^2}{p})$  computation but also  $\mathcal{O}(N)$  communication, and the relatively high latency of the Ethernet network is a serious bottleneck.

$p$	$N$	$T$	$e_w$	$p$	$N$	$T$	$e_s$
1	$2^{10}$	2.9	1	1	$2^{12}$	50.65	1
4	$2^{11}$	4.9	0.59	4	$2^{12}$	17.19	0.74
16	$2^{12}$	6.0	0.48	16	$2^{12}$	5.53	0.57

(a) Weak scalability      (b) Strong scalability

Table 1: Efficiencies on networked workstations

The computational experiments are repeated on the University of Leeds high performance computing (HPC) cluster, Arc-1, which uses infiniband hardware for low latency and high speed communication. The timing data that was collected is shown in Table 2. The faster communication offered by dedicated HPC hardware admits much improved scalability, compared to the non-dedicated cluster, for the same problem sizes. Nevertheless, the performance still falls away as the number of cores increases, and the parallel efficiency will clearly decrease unacceptably for very large numbers of cores. The degradation of the scalability as the problem size is increased can be traced to the communication overhead of the implementation but also, more fundamentally, to the nature of the algorithm itself. Two separate issues may be identified: the loss of efficiency of the algorithm due to communication and synchronization cost; and, the loss of convergence of the algorithm due to the coarsest grid approximate solve.

$p$	$N$	$T$	$e_w$	$p$	$N$	$T$	$e_s$
1	$2^{10}$	3.37	1	1	$2^{12}$	60.9	1
4	$2^{11}$	3.48	0.97	4	$2^{12}$	14.27	1
16	$2^{12}$	3.95	0.85	16	$2^{12}$	3.95	0.96
64	$2^{13}$	4.61	0.73				

(a) Weak scalability      (b) Strong scalability

Table 2: Efficiencies for ARC-1 computation

The two-dimensional grid, on which the PDE discretization is based imposes a certain relationship between communication and computational work. If one assumes a fine grid dimension of  $N = 2^f$  in each coordinate direction and that  $p = 2^k$  MPI processes are used then the strip-partitioned algorithm described above employs an  $N \times 2^{f-k}$  grid on each processor. An immediate observation is that this limits the number of coarser grids to at most  $f - k$ , with  $f - k - 2$  required in practice. Even for large problem sizes this will severely limit the number of grid levels possible in a multigrid cycle and consequently reduce the overall convergence rate of the multigrid algorithm. This also directly affects the strong scalability of the parallel algorithm.

Block-based partitioning alleviates this problem, in the sense that there are now only  $2^{\frac{k}{2}}$  processors in each direction and hence the restriction on the number of grid levels is reduced to  $f - \frac{k}{2}$ . Nevertheless, a limit is still implied and this will result in poor scalability at large numbers of cores. Furthermore, as noted in [45] the total number of communications is increased with the block-based approach to partitioning, which can be problematic when latency or synchronization are issues.

The theoretical multigrid algorithm requires an exact solution at the coarsest grid level to achieve the expected, grid-independent, convergence rate. Parallel direct solution algorithms do not scale optimally, even for the sparse discrete PDE systems considered here and hence this can be considered a bottleneck to optimal parallel scalability. One possible approach for the coarsest level is to use an alternative iterative method, for example Preconditioned GMRES [46], that could be implemented optimally in parallel, to provide a high-quality, efficient approximation on the coarsest level. This requires a parallel preconditioning strategy. So far simple block preconditioning has been investigated but this does not provide enough convergence acceleration. Alternative preconditioners based upon additive domain decomposition approaches are appropriate for parallel implementation [47] and are discussed in Section 4.

## 2.3 Synchronization

The fully synchronous algorithm requires that a processor has completed communications with its neighbours at every smoothing stage and during intergrid transfer operations. (For the algorithm implemented here the fine to coarse grid transfer has no communication but for more general transfer operations this must also be considered.) Some of this overhead can be reduced through the use of MPI non-blocking communication. In this case each grid level can be partitioned into an interior part, that is not dependent on communicated data, and an exterior layer that is dependent. Each processor first initiates the sending and receiving of data at the exterior, before computing on the interior. Only after the interior is complete must the receiving of data be completed and that part of the grid data computed. It is at this final stage that synchronization is enforced. The effect is to overlap inter-processor communication work with on-processor computational work. Synchronization must still be enforced by the end of the particular multigrid operation but a certain amount of idle time is saved.

Unfortunately, even with overlap of communication with computation, the fact that parallelism is only implemented *within* each of the seven steps outlined in the previous subsection, and that these steps must be carried out in a sequential manner, means that there are still numerous synchronization points in the multigrid algorithm. When the work is not perfectly distributed, or when the number of processors means that data transfers may be delayed (or lost altogether), these synchronization points will ultimately ruin the parallel scalability of the approach.

A second important issue is convergence control of the multigrid algorithm. Typ-

ically a global norm of the residual is monitored, implying a gather operation after the residual computation on each processor and synchronization at this point. When multigrid is used as a preconditioning tool for an outer iteration [25] this synchronization issue may be avoided, since typically a fixed number of cycles is employed, but in general it will also lead to processor idle time. This issue of convergence detection is discussed as part of the following section.

### 3 Fundamentals of asynchronous algorithms

The previous section clearly demonstrates some of the main scalability issues associated with the development of a parallel implementation of a state-of-the-art multigrid algorithm. The need to synchronize at the end of each sweep of the Jacobi or Gauss-Seidel smoother, and at the end of each inter-grid transfer operation, makes scalability to very large numbers of cores virtually impossible. This is one of the key motivations for taking a closer look at asynchronous algorithms in this section, and is primarily covered in Subsection 3.1. An additional motivation for asynchronous methods comes through the need for fault tolerance once the number of computational cores becomes very high. Such fault tolerance is required in order to be able to survive minor faults, such as delayed or lost packets of inter-processor data, as well as more major faults such as loss of nodes or connectivity. Hence Subsection 3.2 focuses more specifically on research into fault tolerant algorithms and asynchronous algorithms for computational Grids. Finally, we note that there are additional difficulties that are specific to multilevel solvers, such as multigrid, when implemented in parallel. As seen above, the order in which operations (e.g. coarse grid corrections) are computed at each level can prove to be a significant parallel bottleneck. These issues and the development of asynchronous multilevel algorithms are considered in the next section, Section 4.

#### 3.1 Introduction to asynchronous algorithms

In keeping with the overall focus of this chapter the discussion in this section is primarily based around the solution of sparse systems of linear equations arising from the discretization of an elliptic PDE. We note however that the theory of asynchronous iterative algorithms is substantially broader than this and point the reader to the excellent text [10], which provides a comprehensive introduction.

As described in [10] the defining characteristic of an asynchronous algorithm is that it is made up from a set of local algorithms that do not need to wait “at predetermined points for predetermined messages to become available”. Data is still transferred between the local execution threads however few assumptions are made on the order in which messages arrive or the relative execution speeds of the different tasks. Two subclasses of algorithm may be defined: *totally asynchronous* and *partially asynchronous*. The former are also referred to as *chaotic relaxation* algorithms and are designed to tolerate arbitrarily large communications delays (provided that no processor quits for-

ever). The latter are based upon the stronger assumption of an upper bound on such delays, although this bound may be arbitrarily large in many cases. For each of these sub-classes it is possible to prove convergence results for general fixed point iterations. For example, a fully asynchronous version of

$$\underline{x} := A\underline{x} + \underline{b} \quad (2)$$

may be shown to converge provided that

$$\rho(|A|) < 1, \quad (3)$$

where  $|A|$  is the matrix whose entries are the absolute values of the corresponding entries of  $A$  and  $\rho(\cdot)$  denotes the spectral radius. Note that the usual, synchronized, versions of this iteration (e.g. Jacobi or Gauss-Seidel) only require  $\rho(A) < 1$  for convergence. Furthermore, cases may be constructed where even very small delays can lead to divergence of the asynchronous process when  $\rho(|A|) > 1$  even if  $\rho(A) < 1$ .

As another example, also from [10], consider the iteration

$$\underline{x} := \underline{x} - \gamma(A\underline{x} - \underline{b}) \quad (4)$$

in the case where  $A$  is weakly diagonally dominant. That is, for each row  $i$  we have

$$|1 - a_{ii}| + \sum_{j \neq i} |a_{ij}| \leq 1. \quad (5)$$

Noting that (5) implies

$$\|I - \gamma A\|_{\infty} \leq (1 - \gamma) + \gamma \|I - A\|_{\infty} \leq (1 - \gamma) + 1 = 1, \quad (6)$$

if this last inequality were strict then the totally asynchronous version of the iteration would converge. However, even when the inequality is not strict it may be shown that the partially asynchronous assumption is sufficient to prove convergence (assuming, of course, that a solution exists).

Analysis may also be undertaken as to the relative speeds of convergence of the synchronized and asynchronous versions of these fixed point iterations. Perhaps not surprisingly, if no assumptions are made on the maximum delay in communications then convergence could be arbitrarily slow. With appropriate assumptions on these delays however geometric rates of convergence may be proved for the asynchronous algorithm.

Interest in asynchronous iterative algorithms may be traced back to the 1960s, where it was realized that such an approach to the parallel solution of linear systems of equations could significantly reduce “programming and processor time of a bookkeeping nature” [18]. This particular work considers a partially asynchronous approach, despite the title of the paper being “Chaotic Relaxation”. With this assumption of a bounded maximum delay [18] proves convergence results of the form described

above for symmetric strictly diagonally dominant and for irreducibly diagonally dominant systems. Although slightly weaker than the optimal results, these proofs appear to be the earliest of this type.

In [9] the proofs of [18] are extended to cover the fully asynchronous case, based upon three very mild conditions: (a) only components from previous iterations are used to evaluate a component of the new iteration; (b) eventually all components of an iteration will no longer be used for any subsequent iterations, and; (c) no component is abandoned forever. This paper also presents a selection of results on the convergence rates and the efficiency of asynchronous iterations, although the assumption of a bounded maximum delay is now used: capturing the fact that the speed of convergence deteriorates as this bound grows. Implementation issues are also discussed in this work, with a “Purely Asynchronous Method” being preferred over asynchronous Jacobi and Gauss-Seidel methods, both from an implementation and a performance view point.

An alternative approach to the direct solution of an elliptic problem is to solve an appropriate parabolic problem to steady state (often referred to as time-marching or pseudo-time-stepping schemes). In [4] the use of asynchronous finite difference methods for parabolic PDEs is considered. The most straightforward asynchronous scheme is only suitable for steady-state problems, however variants of this scheme are also proposed where time accuracy is required (although only first order). The better of these variants, referred to as the “time stabilizing” scheme, selects a local time step on each processor in a manner that guarantees that it “keeps up” with its neighbours.

Finally in this subsection we consider a set of papers by Szyld *et al* [26, 11, 27]. In [26] two asynchronous versions of the two-stage block Jacobi algorithm are considered. This two stage approach modifies the usual block Jacobi algorithm by only using an approximation to the inverse of the block diagonal system via the use of an inner iteration (as opposed to an exact block diagonal solve). In a parallel implementation this inner iteration is distributed amongst the processors, and in an asynchronous parallel implementation results from inner solves on other processors are not waited for before each processor moves on to the next outer iteration. The two asynchronous variants either assume that both the inner and the outer iterations are asynchronous (“totally asynchronous”) or just that the outer iteration is asynchronous. In each case convergence results are proved, based upon similar assumptions to those already described above. Note however that the extension of the above results (whilst straightforward for standard block iterative versions of Jacobi, etc.) does not follow immediately for these two-stage methods due to the inexact inner solvers.

In [11] some more practical aspects of the implementation and performance of asynchronous iterative algorithms are considered. A model problem that is similar to the finite difference approximation of the PDE in Section 2 above is considered. It is noted that a partition into stripes requires fewer communication instances than a partition into tiles and it is suggested that this could be an advantage even though the total communication data volume is greater. Perhaps most significantly [11] notes the requirement for a message passing library to provide one-sided communication

routines such as *Read*, *Put* and *Get*. At the time that this article was published (1999) MPI did not provide such routines, although they are now defined in MPI2. Clearly the efficiency of their implementation will be of great importance as the size of parallel systems, and their reliance on asynchronous algorithms, continues to grow.

We conclude this subsection by remarking on the survey paper [27]. This provides an excellent overview of the state-of-the-art on asynchronous iterations by the end of the last Century (including many significant references that are omitted from this chapter due to lack of space). As already noted in the introduction to this chapter, by the year 2000 there were very few computers on which it was possible to make use of more than a few hundred cores and so research up to this time tended to be more theoretical in nature. By this point however the key convergence results for fixed point iterations, for linear and nonlinear systems, had been derived. For the next decade or so, with the advent of Grid Computing, interest in asynchronous algorithms tended to move towards the issue of fault tolerance. This is therefore the topic of the following subsection.

### **3.2 Grid Computing, Fault Tolerance and Asynchronous Iterations**

The term “metacomputing” has been used to describe the application of two or more distant parallel computers, linked by a relatively slow network, for the solution of a single computational problem, e.g. [29]. As noted in [29] the common parallel programming tool of overlapping communication by computation is an essential component of metacomputing algorithms. Furthermore, it is observed that asynchronous communication is the key to a “more flexible and efficient way of using the network between two [or more] machines”. Indeed, relatively good parallel efficiencies are demonstrated across a selection of test problems, including the solution of a system of time-dependent PDEs.

More recent work by Bahi *et al* [5] focuses directly on the issues of applying asynchronous iterative algorithms in a Grid Computing context. This experimental study contrasts the use of synchronous and asynchronous algorithms when run across heterogeneous distant machines. The results demonstrate that the latter approach is superior in all cases that were considered, including both linear and nonlinear iterations. It is also noted that, in addition to being more naturally suited to Grid Computing environments, asynchronous algorithms are generally simpler to implement in software: the only significant challenge coming in convergence detection. Subsequent research by the same group, [6], developed a multisplitting Newton method for the asynchronous parallel solution of a three-dimensional advection-diffusion-reaction system. The implementation is based upon the development of a Java Asynchronous Computing Environment (JACE) which is run on a set of multiple machines across a number of distant sites. Asynchronization is used to support the efficient overlap of communication and computation and, as before, this allows the asynchronous version to be more efficient than proves to be possible in the synchronous case.

Other groups have also developed software for computational Grid environments based upon the use of asynchronous iterations. For example the GREMLINS (GRid Efficient Method for LINear Systems) solver is introduced in [22]. As with JACE, this solver may be executed either in a synchronous or an asynchronous mode. The parallelism is achieved using a multilevel splitting algorithms that may be applied to an arbitrary sparse matrix system: with each processor being responsible for a block of rows of the system. The overall iteration is again closely related to block Jacobi iteration, with either exact inner solves based upon sparse direct solvers, or inexact iterative inner solvers. In each case these are simply standard sequential solvers. Communication is between each processor and its two neighbours – with the exception of the convergence detection phase which is based upon two different possible methods (a centralized and a decentralized version). Tests undertaken on the French GRID’5000 architecture show that the asynchronous version is often, but not always, faster than the synchronous case.

A slightly different approach to the multisplitting technique is based upon the Schwarz alternating method. This is applied to the solution of a nonlinear diffusion PDE in [48], where the link with multisplitting is also discussed (Schwarz alternating being shown to be a special case of the latter under certain conditions on the discretization). Monotone convergence is demonstrated for the parallel asynchronous iteration and a sample implementation is presented and analysed. This approach is also applied to a challenging three-dimensional flow problem, governed by Navier-Stokes equations coupled with transport and potential equations, in [17]. Convergence of the asynchronous iteration is again proved, under appropriate conditions, however the computational demonstrations are only performed on a modest number of processors.

Subsequent work by the same authors, [43], returns to the recurrent issue of convergence detection for asynchronous iterations. Their approach defines the concept of a macro-iteration by looking back at the sequence of events that led to local updates: by ensuring that all convergence tests are made within the same macro-iteration it is possible to provide guarantees on a global convergence criterion being satisfied, albeit at the expense of additional data exchanges. More recently still, the research described in [15] also addresses the question of convergence detection for asynchronous iterations. The authors present a decentralized algorithm that aims to detect global convergence as part of a fault tolerant solution procedure. Their paper focuses on fault tolerance as well as global convergence detection and presents an implementation using their JaceP2P platform, which is designed for asynchronous algorithms (and is described in more detail in [16]). The convergence test incorporates local and global phases, which are described in detail.

Interest in asynchronous algorithms, in terms of both fundamental theory and practical application, continues to be an active area of research. Other recent contributions include the analysis of another asynchronous parallel multisplitting iteration, [19], that is derived from the weighted multisplitting schemes first introduced in [54]. This analysis proves convergence of the proposed method, which is a generalization of earlier work on the same topic in [52, 7]. Other approaches include the use of asynchronous

preconditioners as part of a preconditioned conjugate gradient solver, [21], which itself still has a synchronization step at each iteration. The goal is to devote “the bulk of the computational effort to the preconditioner [so that] the computation to communication ratio can be improved significantly, while considerably reducing the number of expensive (outer) synchronisations”. This is achieved by using a flexible subspace iteration as the asynchronous preconditioner, so as to accelerate convergence – but without the need to introduce any additional synchronization points. By choosing this asynchronous preconditioner so as to focus on coarse subspace corrections it is possible to accelerate convergence when solving problems arising from the discretization of elliptic PDEs, [20]. This approach may be viewed as a crude multilevel method, in which there is a single coarse space correction at each Krylov subspace iteration. The following section considers asynchronous multilevel methods in more detail.

## 4 Asynchronous multilevel methods

As discussed in Sections 1 and 2 the fastest sequential solvers are based upon the use of multilevel techniques such as multigrid. Parallel implementations of multigrid are possible however, as we have seen, these require frequent synchronizations and this has an adverse impact on their parallel performance. Other multilevel algorithms do exist: either possessing the optimal (i.e.  $O(n)$ ) complexity or very close to this (e.g.  $O(n \log n)$ ), [24, 56]. In [24] a multilevel version of the additive Schwarz approach is developed, which is well-suited to parallel implementation, whilst in [56] a different multilevel splitting is proposed, based upon hierarchical bases.

Perhaps the best known parallel multilevel solvers are the preconditioners developed in [13], and references therein. Note that this approach, usually referred to as BPX, is only applicable as a preconditioner and does not generally converge as a stand-alone iteration. It is based upon the use of an additive decomposition which allows different subspace problems to be solved simultaneously, rather than sequentially. Unlike the parallel multigrid described in Section 2 above, which is based upon parallelism within levels, the BPX technique is therefore a genuinely concurrent multilevel algorithm. The paper [13] describes and analyses this algorithm in an abstract setting which allows a quite general theory to be developed. Nevertheless, the basic preconditioning step is remarkably straightforward:

$$\mathcal{B}v = \sum_{k=1}^J \sum_{\ell} \phi_k^{\ell} \int_{\Omega} v \phi_k^{\ell} dx, \quad (7)$$

where  $\{\phi_k^{\ell}\}$  denotes the usual nodal basis for the piecewise linear finite element space obtained from  $k - 1$  refinements of a coarse triangulation  $T_1$  and  $v$  is a function in the finest space. It may be shown that when this preconditioner is used in the solution of a finite element discretization of a suitable linear elliptic PDE, the resulting condition number is at worst  $O(J^2)$ , where  $J$  is the maximum number of refinement levels in the triangulation. The simplicity of the parallel implementation is immediately evident

from (7), for which the terms in the summation may be accumulated in any order, and the combination with local mesh refinement requires no further extensions to the theory (and only minor extensions to the algorithmic realization).

Later results are able to improve further on the work of [13]. For example, [57] is able to analyse a closely related additive Schwarz framework in which the preconditioned system condition number is bounded independently of the number of levels of refinement. More recently, [50] presents a generalization of the BPX approach which specifically allows for asynchronous updates, with convergence results that are again independent of the mesh parameters under suitable conditions. Furthermore, this approach is applicable to convex minimization problems and is therefore more general than just for linear systems arising from elliptic PDEs (which correspond to minimizations of quadratic forms).

A different approach, that may also be thought of as another generalization of the BPX technique, is the asynchronous fast adaptive composite (AFAC) method of [39, 38]. As the name suggests, this is specifically aimed at problems for which local mesh refinement is appropriate, and it is an asynchronous generalization of the fast adaptive composite (FAC) methods from the same group (e.g. [32]). To illustrate the approach, consider the finite element solution of a linear elliptic PDE in a domain  $\Omega$  which is covered by a coarse grid,  $\Omega^H$  say, and which has a locally refined region whose grid is denoted by  $\Omega^h$ . Note that both  $\Omega^H$  and  $\Omega^h$  cover the refined region and so the BPX preconditioner will involve two simultaneous corrections to the solution on this region. The AFAC method adds an additional grid, which has the coarse grid resolution but is only present in the refined region: this “is used to resolve and eliminate the error components that are duplicated in the original pair of grids” [37]. Hence the AFAC approach involves additional subspace solves, compared to BPX, but this extra cost is small. Further details and performance tests may also be found in [31, 41]. It is interesting to note however that none of these multilevel asynchronous techniques have yet been applied to extreme computing environments using many thousands of processors.

Indeed, the generalizations that have been developed most recently tend to have focused on broadening the class of problem that may be solved or on the production of general software frameworks. Examples of the former include [44], who apply a multilevel additive Schwarz preconditioner to the solution of a model of bioelectrical activity of heart tissue. Examples of the latter include [3] who provide a package of multilevel domain decomposition parallel preconditioners, including additive Schwarz preconditioners as described in [2]. In none of these cases has there been an explicit attempt to exploit the asynchronous potential of these algorithms however.

## 5 Outlook

As computational hardware continues to evolve in the years ahead there will be a need to develop new parallel numerical algorithms that are capable of exploiting many thou-

sands of cores concurrently. Historically, good parallel efficiency has only been possible with very large numbers of cores for certain classes of algorithm, such as those based upon dense matrices. These algorithms have a large amount of computational work per unit of data and a large amount of computation relative to communication.

Many important problems in computational engineering take a different form however, ultimately reducing to the solution of sparse systems of linear equations. For the solution of partial differential equations for example, the finite element or finite difference discretization schemes lead to precisely such sparse systems. The current state-of-the-art provides a choice between parallelizing a fast and accurate sequential algorithm, but with only limited parallel scalability, or else implementing a highly scalable asynchronous iteration but with only a slow rate of convergence. Neither approach is likely to prove satisfactory in the future, where there will be a need to solve complex PDE models using large numbers of degrees of freedom, for high accuracy, and large numbers of computational cores, for high speed.

An important topic for current research therefore is the development of algorithms that possess both excellent convergence properties (e.g. multilevel solvers) and excellent parallel scalability (e.g. fault tolerant and asynchronous). In this paper we have discussed some of the options that are available for multilevel solvers and for asynchronous iterations. The next step therefore must be to find ways in which to develop these ideas further for the practical realization of highly scalable but near-optimal iterative techniques.

## Acknowledgements

The authors acknowledge the Engineering and Physical Science Research Council (EPSRC) for their financial support for this work through grant number EP/I006737/1.

## References

- [1] E. Agullo, A. Guermouche and J.-Y. L'Excellent. A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory. *Parallel Computing* 34:296–317, 2008.
- [2] P. D'Ambra, D. di Serafino and S. Filippone. On the development of PSBLAS-based parallel two-level Schwarz preconditioners. *Appl. Numer. Math.* 57:1181–1196, 2007.
- [3] P. D'Ambra, D. di Serafino and S. Filippone. MLD2P4: A package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. *ACM Trans. Math. Software* 37, article 30, 2010.
- [4] D. Amitai, A. Averbuch, M. Israeli, S. Itzikowitz and E. Turkel. A survey of asynchronous finite-difference methods for parabolic PDEs on multiprocessors. *App. Num. Math.* 12:27–45, 1993.

- [5] J.M. Bahi, S.Contassot-Vivier and R.Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing* 31:439-461, 2005.
- [6] J.M. Bahi, R. Couturier, K. Mazouzi and M. Salomon. Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. *Appl. Math. Modelling* 30:616-628, 2006.
- [7] Z.-Z. Bai. A new generalized asynchronous parallel multisplitting iteration method. *J. Comput. Math.* 17:449-456, 1999.
- [8] R.E. Bank and P.K. Jimack. A new parallel domain decomposition method for the adaptive finite element solution of elliptic partial differential equations. *Concurrency and Computation: Practice and Experience* 13:327-350, 2001.
- [9] G.M. Baudet. Asynchronous iterative methods for multiprocessors. *J. Assoc. Comp. Mach.* 25(2):226-244, 1978.
- [10] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*, Athena Scientific, 1997.
- [11] K. Blathras, D.B. Szyld and Y. Shi. Timing models and local stopping criteria for asynchronous iterative algorithms. *J. Par. Dist. Comp.* 58:446-465, 1999.
- [12] D. Braess. Towards algebraic multigrid for elliptic problems of second order. *Computing* 55:379-393, 1995.
- [13] J.H. Bramble, J.E. Pasciak and J. Xu. Parallel multilevel preconditioners. *Math. Comp.* 55: 1-22, 1990.
- [14] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Math. Comp.* 31:333-390, 1977.
- [15] J.-C. Charr, R. Couturier and D. Laiymani. A decentralized and fault tolerant convergence detection algorithm for asynchronous iterative algorithms. *J. Supercomput.* 53:269-292, 2010.
- [16] J.-C. Charr, R. Couturier and D. Laiymani. A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures. *Future Generation Comput. Systems*, doi:10.1016/j.future.2010.04.013, 2010.
- [17] M. Chau, P. Spiteri, R. Guivarch and H.C. Boisson. Parallel asynchronous iterations for the solution of a 3D continuous flow electrophoresis problem. *Computers and Fluids* 37:1126-1137, 2008.
- [18] D. Chazan and W. Miranker. Chaotic relaxation. *Lin. Alg. Apps* 2:199-222, 1969.
- [19] F. Chen. Asynchronous multisplitting iteration with different weighting schemes. *Appl. Math. Comput.* 216: 1771-1776, 2010.
- [20] T.P. Collignon and M.B. van Gijzen. Parallel scientific computing on loosely coupled networks of computers. In *Advanced Computational Methods in Science and Engineering*, B.Koren and C.Vuik (Eds.), Lecture Notes in Computational Science and Engineering (Springer, Berlin) 71: 76-106, 2010.
- [21] T.P. Collignon and M.B. van Gijzen. Two implementations of the preconditioned conjugate gradient method on heterogeneous computing grids. *Int. J. Math. Comput. Sci.* 20: 109-121, 2010.

- [22] R. Couturier, C. Denis and F. Jézéquel. GREMLINS: a large sparse linear solver for grid environment. *Parallel Computing* 34:380-391, 2008.
- [23] J.E. Dendy. Black box multigrid. *J. Comput. Phys.* 48:366–386, 1982.
- [24] M. Dryja and O.B. Widlund. Multilevel additive methods for elliptic finite element problems. In *Parallel Algorithms for Partial Differential Equations, Proceedings of the Sixth GAMM-Seminar*, W.Hackbusch (Ed.), Kiel, January 1990 (Vieweg, Braunschweig), 1991.
- [25] H.C. Elman, D.J. Silvester and A.J. Wathen. *Finite Elements and Fast Iterative Solvers with applications in incompressible fluid dynamics*, OUP, 2005.
- [26] A. Frommer and D.B. Szyld. Asynchronous two-stage iterative methods. *Num. Math.* 69:141–153, 1994.
- [27] A. Frommer and D.B. Szyld. On asynchronous iterations. *J. Comp. App. Math.* 123:201–216, 2000.
- [28] P.H. Gaskell, P.K. Jimack, Y.-Y. Koh and H.M. Thompson. Development and application of a parallel multigrid solver for the simulation of spreading droplets. *Int. J. Numer. Meth. Fluids* 56:979–1002, 2008.
- [29] M. Garbey and D. Tromeur-Dervout. A parallel adaptive coupling algorithm for systems of differential equations. *J. Comput. Phys.* 161:401–427, 2000.
- [30] M.J. Grote and T. Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM J. Sci. Comp.* 18:838–853, 1997.
- [31] L. Hart and S.F. McCormick. Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Basic ideas. *Parallel Comp.* 12:131–144, 1989.
- [32] L. Hart, S.F. McCormick and A. O’Gallagher. The fast adaptive composite-grid method (FAC): Algorithms for advanced computers. *Appl. Math. Comp.* 19:103–125, 1986.
- [33] V.E. Henson, W.L. Briggs and S.F. McCormick. *A Multigrid Tutorial*, SIAM, 2000.
- [34] D.C. Hodgson and P.K. Jimack. A domain decomposition preconditioner for a parallel finite element solver on distributed unstructured grids. *Parallel Computing* 23:1157–1181, 1997.
- [35] T. Huckle, A. Kallischko, A. Roy, M. Sedlacek and T. Weinzierl. An efficient parallel implementation of the MSPAI preconditioner *Parallel Computing* 36:273–284, 2010.
- [36] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning *SIAM J.Sci. Comp.* 22:2194–2215, 2001.
- [37] J.E. Jones and S.F. McCormick. Parallel multigrid methods. In *Parallel Numerical Algorithms*, D.Keyes, A.Sameh and V. Venkatakrishnan (Eds.), NASA/LaRC Interdisciplinary Ser. in Sci. and Engrg. (Kluwer), 1997.
- [38] B. Lee, S.F. McCormick, B. Philip and D.J. Quinlan. Asynchronous fast adaptive composite-grid methods for elliptic problems: Numerical results. *SIAM J. Sci. Comp.* 25:682–700, 2003.
- [39] B. Lee, S.F. McCormick, B. Philip and D.J. Quinlan. Asynchronous fast adaptive composite-grid methods for elliptic problems: Theoretical foundations. *SIAM J.*

- Numer. Anal.* 42:130–152, 2004.
- [40] X.Y.S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. on Math. Software* 31:302–325, 2005.
- [41] S.F. McCormick and D. Quinlan. Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Performance results. *Parallel Comp.* 12:145–156, 1989.
- [42] J. Mayer. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing* 86:291–312, 2009.
- [43] J.C. Miellou, P. Spiteri and D. El Baz. A new stopping criterion for linear perturbed asynchronous iterations. *J. Comput. Appl. Math.* 219:471–483, 2008.
- [44] L.F. Pavarino and S. Scacchi. Multilevel additive Schwarz preconditioners for the bidomain reaction-diffusion system. *SIAM J. Sci. Comput.* 31:420–443, 2008.
- [45] G. Romanazzi, P.K. Jimack and C.E. Goodyer. Reliable performance prediction for multigrid software on distributed memory systems. *Adv. Engrg. Software* doi:10.1016/j.advengsoft.2010.10.005, 2011.
- [46] Y. Saad and M.H. Schultz. GMRES - A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* 7:856–869, 1986.
- [47] B. Smith, P. Bjørstad and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, 1996.
- [48] P. Spiteri, J.-C. Miellou and D. El Baz. Parallel asynchronous Schwarz and multisplitting methods for a nonlinear diffusion problem. *Numerical Algorithms* 33:461–474, 2003.
- [49] P.S. Summant, A.C. Cangellaris and N.R. Aluru. A node-based agglomeration AMG solver for linear elasticity in thin bodies. *Comm. Numer. Meth. Engrg.* 25:219–236, 2009.
- [50] X.-C. Tai and P. Tseng. Convergence rate analysis of an asynchronous space decomposition method for convex minimization. *Math. Comp.* 71: 1105–1135, 2002.
- [51] U. Trottenberg, C.W. Oosterlee and A. Schüller. *Multigrid*, Academic Press, 2000.
- [52] D.-R. Wang, Z.-Z. Bai and D.J. Evans. A class of asynchronous parallel matrix multisplitting relaxation methods. *Parallel Algorithms Appl.* 2: 173–192, 1994.
- [53] P. Wesseling. *Introduction to Multigrid Methods*, Wiley, 1992.
- [54] R.E. White. Multisplitting with different weighting schemes. *SIAM J. Matrix Anal. Appl.* 10: 481–493, 1989.
- [55] J.C. Xu. Iterative methods by space decomposition and subspace correction. *SIAM Review* 34:581–613, 1992.
- [56] H. Yserentant. On the multi-level splitting of finite element spaces. *Numer. Mathematik* 49:379–412, 1986.
- [57] X. Zhang. Multilevel Schwarz methods. *Numer. Mathematik* 63: 521–539, 1992.