Investigating the Performance of Asynchronous Jacobi's Method for Solving Systems of Linear Equations

Bethune, Iain and Bull, J. Mark and Dingle, Nicholas J. and Higham, Nicholas J.

2011

MIMS EPrint: **2011.82**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Investigating the Performance of Asynchronous Jacobi's Method for Solving Systems of Linear Equations

Iain Bethune J. Mark Bull

EPCC, James Clark Maxwell Building,
The King's Buildings, University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK
{ibethune,markb}@epcc.ed.ac.uk

Nicholas J. Dingle Nicholas J. Higham

School of Mathematics, University of Manchester,
Oxford Road, Manchester M13 9PL, UK
{nicholas.dingle,nicholas.j.higham}@manchester.ac.uk

ABSTRACT

Ever-increasing core counts create the need to develop parallel algorithms that avoid closely-coupled execution across cores. In this paper we present two case studies investigating the performance of several parallel asynchronous implementations of Jacobi's method for solving systems of linear equations. Although conditions for the convergence of asynchronous Jacobi are well known, what drives its rate of convergence is less well understood. The first case study investigates the algorithm's performance when executed on large numbers of processors on a Cray XE6, while the second explores the effect of varying the number of synchronous and asynchronous processors. We observe that the performance of parallel asynchronous Jacobi is highly implementation, problem and architecture-dependent.

1. INTRODUCTION

Modern high-performance computing systems are typically composed of many thousands of cores linked together by high bandwidth and low latency interconnects such as Infiniband. Over the coming decade core counts will continue to grow as efforts are made to reach Exaflop performance. In order to continue to exploit these resources efficiently, new software algorithms and implementations will be required that avoid tightly-coupled synchronisation between participating cores and that are resilient in the event of failure.

This paper investigates one such class of algorithms. The solution of sets of linear equations $Ax = b$, where A is a large, sparse $n \times n$ matrix and x and b are vectors, lies at the heart of a large number of scientific computing kernels, and so efficient solution implementations are crucial. Existing iterative techniques for solving such systems in parallel are typically *synchronous*, in that all processors must exchange updated vector information at the end of every iteration, and this creates a barrier beyond which computation cannot proceed until all participating processors have reached that point. Such approaches will not scale.

Instead, we are interested in developing *asynchronous* techniques that avoid this blocking behaviour by permitting processors to operate on whatever data they have, even if new data has not yet arrived from other processors. To date there has been work on both the theoretical [1, 2, 5] and the practical [4, 6] aspects of such algorithms. In order to be able to reason about these algorithms we need to understand what drives the speed of their convergence, but existing results merely provide sufficient conditions for the algorithms to converge. Here, we look for insights by investigating the performance of the algorithms experimentally.

Through two case studies we investigate some of the issues that can manifest themselves with asynchronous algorithms. In the first case study, we look at the performance of the algorithm on a Cray XE6 and compare a synchronous implementation against two asynchronous versions. In the second case study we explore what can happen when synchronous and asynchronous approaches are mixed together. In particular, we highlight how simple measures of algorithm performance, such as the mean number of iterations, can hide surprising behaviour in an asynchronous context.

2. JACOBI'S METHOD

Jacobi's method for the system of linear equations $Ax = b$, where the $n \times n$ matrix A is assumed to be nonsingular and to have nonzero diagonal, computes the sequence of vectors $x^{(k)}$, where

$$x_i^{(k)} = \frac{1}{a_{ii}}\Big(b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)}\Big), \qquad i = 1\colon n.$$

The $x_i^{(k)}$, $i = 1\colon n$, are independent, which means that vector element updates can be performed in parallel. Jacobi's method is also amenable to an asynchronous parallel implementation in which newly-computed vector updates are exchanged when they become available rather than by all processors at the end of each iteration. This asynchronous scheme is known to converge if $\rho(|M|) < 1$ [5], where $\rho(|M|)$ denotes the spectral radius (maximum absolute value of the eigenvalues) of the matrix $|M| = (|m_{ij}|)$ and $M = -D^{-1}(L+U)$. Here, D, L, U are, respectively, the diagonal and strictly lower and upper triangular parts of A. In contrast, the synchronous version of Jacobi's method converges if the weaker condition $\rho(M) < 1$ holds.

These known conditions for convergence in the synchronous and asynchronous cases are the main reason why we have chosen to base our case studies on Jacobi's method. In practice, synchronous parallel versions of other iterative schemes (e.g. conjugate gradient methods) are often preferred, but asynchronous parallel versions of these approaches are not yet well developed.

3. CASE STUDY 1

In order to investigate the convergence and performance of Jacobi's method at large scale, we have implemented three variants - Synchronous Jacobi (*sync*), and two Asynchronous

Jacobi (*async* and *async_racy*) - all parallelised using MPI and written in Fortran 90. The behaviour of these programs was studied on HECToR, a Cray XE6 supercomputer, using up to 24 576 processor cores. HECToR is composed of 1 856 compute nodes, each with two 12-core AMD Opteron 2.1GHz processors and 32GB of memory. A 3D torus interconnect exists between nodes, which are connected to it via Cray's Gemini router chip.

3.1 Implementation

Instead of developing a general Jacobi solver with explicit A matrix, we solve the 3D diffusion problem $\nabla^2 u = 0$ using a 6-point stencil over a 3D grid. In all cases, we have fixed the grid size for each process at 50^3, and so as we increase the number of participating processes the global problem size is *weak-scaled* to match. The boundary conditions for the problem are set to zero, with the exception of a circular region on the bottom of the global grid defined by $e^{-((0.5-x)^2+(0.5-y)^2)}$. The grid is initialised to zero at the start of the iteration, and convergence is declared when the ℓ_2-norm of the residual (normalised by the source) is less than 10^{-4}.

In common with many grid-based distributed memory applications, a 'halo swap' operation is required, since the update of a local grid point requires the data from each of the 6 neighbouring points in 3D. If a point lies on the boundary of a process's local grid, then data is required from a neighbouring process. To achieve this, each process stores a single-element 'halo' surrounding its own local grid, and this is updated with new data from the neighbouring processes' boundary regions at each iteration (in the synchronous case), and vice versa, hence 'swap'.

The structure of all three programs is similar (see Figure 1), but the implementation of the halo swap and global residual calculation vary between versions.

- **sync** Halo swaps are performed using `MPI_ISend` and `MPI_IRecv` followed by a single `MPI_Waitall` for all the sends and receives. Once all halo swap communication has completed, a process may proceed. Global summation of the residual is done every 100 iterations via `MPI_Allreduce`, which is a blocking collective operation. In this implementation, all processes are synchronised by communication, and therefore proceed in lockstep.

- **async** This implementation allows multiple halo swaps to be 'in flight' at any one time (up to 100 between each pair of processes). This is done by means of a circular buffer storing `MPI_Request`s. When a process wishes to send halo data it uses up one of the 100 MPI requests and sends the halo data to a corresponding receive buffer on the neighbouring process. If all 100 MPI requests are active (i.e. messages have been sent but not yet received) it will simply skip the halo send for that iteration and carry on iterating. On the receiving side, a process will check for arrival of messages, and if new data has arrived, copy the data into the halo cells of its u array and continue (even if no new data was received in that iteration). By using multiple receive buffers we ensure that the data in the u array halos on each process is a consistent image of the halo data that was sent at some iteration in the past by the neighbouring process.

  In addition, we also replace the blocking reduction with an asynchronous binary-tree based scheme, where each process calculates its local residual, and inputs this value into the reduction tree. These local contributions are summed and sent on up the tree until reaching the root, at which point the global residual is broadcast (asynchronously) down the same reduction tree. Since the reduction takes place over a number of iterations (the minimum time being the message latency $\times \log_2 p$), as soon as a process receives the global residual it immediately starts another reduction.

- **async_racy** A potential performance optimisation is that instead of having 100 individual buffers to receive halo data, we instead have a single buffer to which all in-flight messages are sent. This introduces a deliberate race condition in that as data is read out of the buffer into the u array, other messages may be arriving simultaneously, depending on the operation of the MPI library. In this case, assuming atomic access to individual data elements (in this case double-precision floats), we are no longer guaranteed that we have a consistent set of halo data from some iteration in the past, but in general will have some combination of data from several iterations. It is hoped that this reduces the amount of memory overhead (and cache space) needed for multiple buffers, without harming convergence in practice.

3.2 Timing Results

Table 1 shows results for each of the three versions on a range of processor counts on HECToR. The smallest run is done with 24 processes, since this is the number of cores on a HECToR compute node; using fewer than this would increase the amount of memory bandwidth available to the processes, and artificially inflate the iteration rates in comparison to runs on larger numbers of processes using full nodes.

The three different processor counts (24, 1 536 and 24 576) correspond to A having 3 million, 192 million and 3.072 billion rows respectively. Due to the expense of running 24 576 processor jobs, we report results from a single run of each asynchronous variant for each processor count. We report the maximum and minimum number of iterations across all processors, as well as the average number of iterations and the total runtime.

Table 1: HECToR results summary.

Processes	Iterations			Execution
Version	Min.	Mean	Max.	Time (s)
24				
sync		8200		24.1
async	10479	10991	12024	31.2
async_racy	7984	8476	9025	23.7
1536				
sync	43870	52318	61932	164.0
async		44200		151.3
async_racy	39624	45678	53384	143.5
24576				
sync		101400		351.2
async	94350	109460	137636	348.8
async_racy	85777	102536	132904	319.0

```
do

   swap a one-element-thick halo with each neighbouring process

   every 100 steps
      calculate local residual
      sum global residual
      if global residual < 10^(-4) then stop

   for all local points
      u_new(i,j,k) = 1/6*(u(i+1,j,k)+u(i-1,j,k)+u(i,j+1,k)+u(i,j-1,k)+u(i,j,k+1)+u(i,j,k-1))

   for all local points
      u(i,j,k) = u_new(i,j,k)

end do
```

Figure 1: Pseudocode of parallel Jacobi implementation on HECToR.

Consider, first, the results for 24 processes. It is clear that both asynchronous versions achieve higher average iteration rates (353 and 357 iterations per second, compared with 341 iterations per second for the synchronous case). Performance profiling using the CrayPAT tools (Figure 2(a)) showed that the most significant extra cost in the synchronous code was the `MPI_Waitall` call, whose cost is entirely avoided in the asynchronous codes. It is also apparent that the *async* version takes around 25% more iterations to converge than than the synchronous version. While this might be expected, since in the absence of new data, extra iterations might be performed that do not improve (very much) the overall convergence, it is not clear why this is not the case for the *async_racy* version, which takes approximately the same number of iterations on average as the synchronous implementation. Looking at the total execution time, *async_racy* slightly outperforms *sync*, while *async* is significantly slower.

1 536 processes (Figure 2(b)) is a medium-sized simulation, which is representative of typical job sizes run on HECToR today. Here we begin to see the greater scalability of the asynchronous versions. In the synchronous case the cost of the `MPI_Waitall` has grown even further (since messages now have to travel over the network, thus increasing latency compared to the intra-node case), reducing the iteration rate to only 292 iterations per second (c.f. 318-319 for both asynchronous versions). The increased latency cost is effectively masked by allowing multiple messages to be in flight at once. While the *async* version is still overall slower than the *sync* version, for this problem size, *async_racy* is already 5% faster.

The largest job allowed on HECToR is 24 576 processes (Figure 2(c)), taking over half of the machine. Again, the cost of the `MPI_Waitall` has grown, but we also for the first time we see the synchronisation cost of the `MPI_Allreduce` used for the global residual calculation. These effects combine to cause *sync* to run slower than the *async* code. However, as before, the *async_racy* version is faster still, up to 10% faster than the synchronous code.

Looking at these results as a whole, two points are immediately apparent. First, the relaxation of data consistency in the *async_racy* code does provide a substantial performance improvement, such that for all 3 processor counts it always outperforms the synchronous implementation. Secondly, the more careful *async* version does obtain higher iteration rates than the synchronous version, but because it takes more iterations to converge it only outperforms it for very large processor counts, and is never faster than *async_racy*.

3.3 Convergence

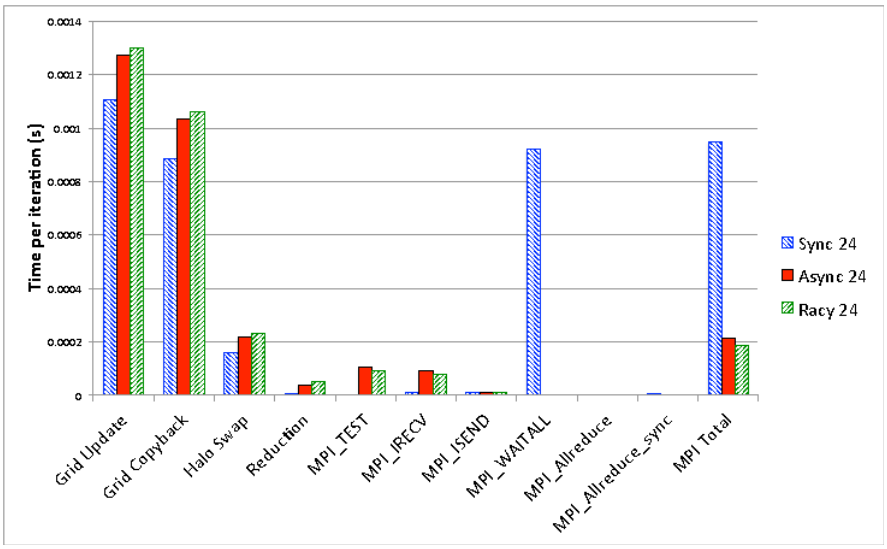
As can be seen in Figure 3, while the *sync* and *async_racy* versions seem to share similar convergence profiles (although *async_racy* is around 5% faster) the *async* version exhibits two features that are different. First, it begins to converge faster than either of the other two methods, but gradually shallows out. This indicates that the choice of stopping criterion may influence which version is in fact fastest for a given problem size. In addition, there are several 'jagged' sections of the curve, where for several iterations the residual increases, before settling back towards convergence again. We hypothesise that these are caused by multiple iterations occurring without new data arriving from neighbours (so each process continues to converge with respect to its own halo data), followed by the arrival of data, which is now so 'old' that when the residual is calculated, artificially high values are reached since the local data is far from convergence with respect to the new halo data. This behaviour is not observed in *async_racy*, but the reason for this is not yet understood.

4. CASE STUDY 2

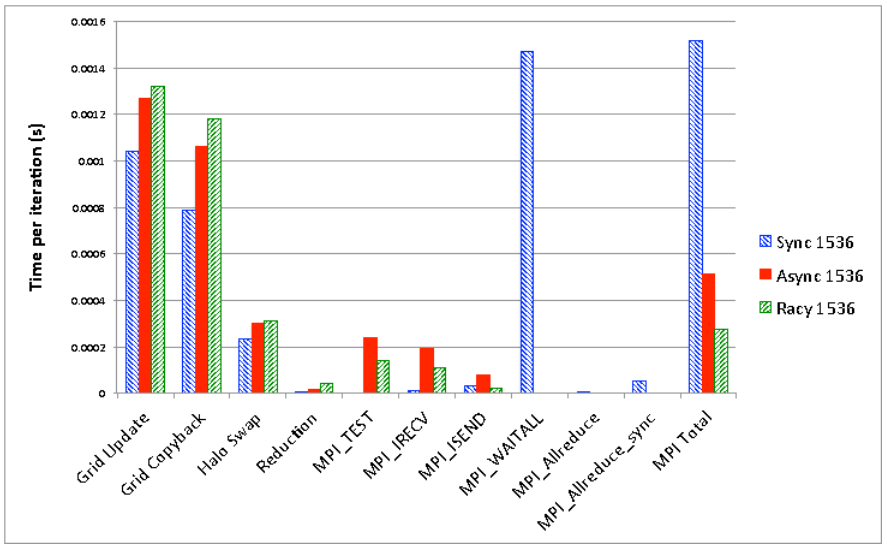
In this case study we investigate the convergence behaviour of an asynchronous parallel implementation of Jacobi's method when subsets of the participating processors synchronise at the end of every iteration. Such a situation could occur in a hybrid solver where a shared-memory library such as OpenMP is used to parallelise computations within a multicore node and a message-passing library like MPI is used to parallelise computations across nodes. Synchronous computation might be used within a node, but vector update exchanges between nodes might be performed asynchronously.

4.1 Implementation

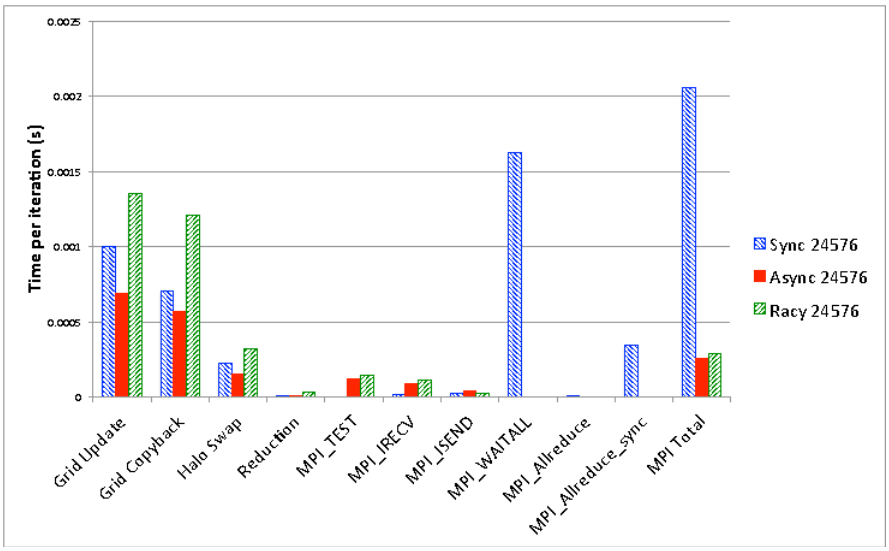
Figure 4 shows a pseudocode representation of our parallel asynchronous solver. The original is written using C++ and MPI. The intention is not to present a fully-fledged solver, but rather to produce an example implementation to investigate the key dynamics that such solvers may display. Functionality such as efficient communication patterns (e.g. only sending updated vector elements to those processes which



(a) 24 processes



(b) 1 536 processes



(c) 24 576 processes

Figure 2: Profiles showing time per iteration for Jacobi on HECToR.

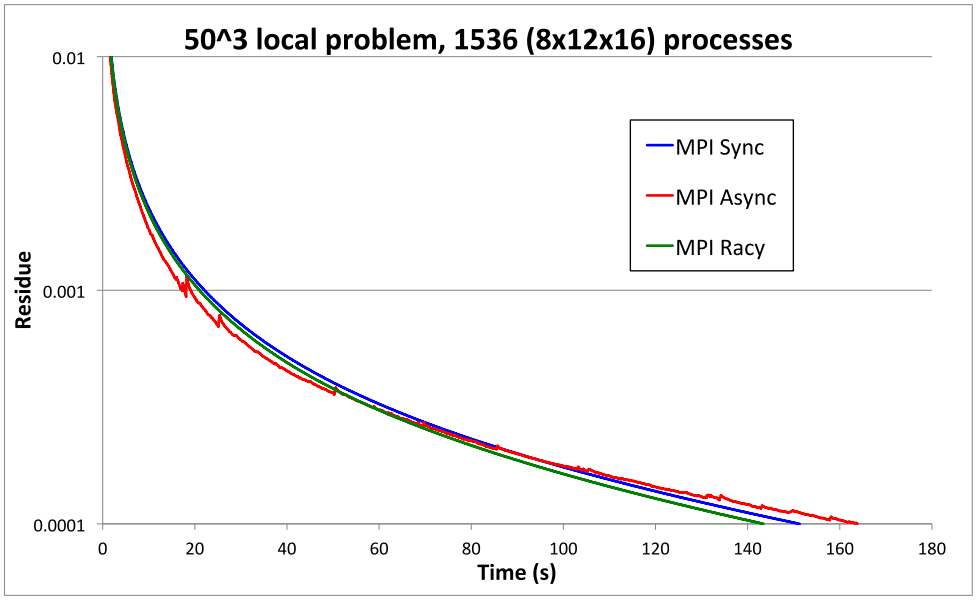


Figure 3: Convergence for 1 536 processes on HECToR.

```
while(!converged)

  //compute Jacobi on locally held rows using local and remote vector elements
  for (every matrix row i)
    for (every row entry j)
      sum += A[i][j]*x[j];
    end
    new_x[i] = (b[i] - sum)/A[i][i];
  end

  //work out our local convergence data and asynchronously send it to rank 0
  convergence[0] = max(new_x - x);
  convergence[1] = max(new_x);
  MPI_Put(convergence, 0);

  //varying the members of "subset" allows us to adjust the number of
  //synchronising processors
  MPI_Barrier(subset);
  for (every other processor p)
    MPI_Put(new_x, p);
  end
  MPI_Barrier(subset);

  //rank 0: work out global convergence from current convergence data
  //        and asynchronously update "converged" on all processors
  if (root processor)
    max_diff = max(convergence[0]);
    max_soln = max(convergence[1]);

    if (max_diff/max_soln < epsilon)
      converged = false;
      for (every other processor p)
        MPI_Put(converged, p);
      end
    end
  end

end
```

Figure 4: Pseudocode of parallel Jacobi implementation where the degree of synchronisation can be adjusted.

require them) and node failure detection and recovery has not yet been implemented.

To achieve asynchronous exchange of data we use MPI-2 remote memory operations to enable processes to put data (with `MPI_Put`) directly into eachother's address spaces. This avoids the need to match `MPI_Sends` and `MPI_Recvs` on both sides, and also avoids the need to explicitly track the completion of asynchronous `MPI_Isends` or `MPI_Irecvs`. Each process maintains a buffer of remote vector data for every other process (i.e. when there are p processes in total, each one will have $p-1$ of these buffers), and so as in previous case study's *async_racy* implementation we may encounter race conditions where data is read from a buffer at the same time as new data is being written into it.

We also use this remote memory functionality to transmit information about the subvectors held on each processor to the master (rank 0) to check for global convergence, and also for the master to signal when convergence has been achieved.

Synchronisation of subsets of processes is achieved with `MPI_Barriers` around the transmission of vector updates. Only those processes which belong to the MPI communicator `subset` are blocked by these, and this allows us to control the number of synchronising processes. Processors which do not block at these barriers proceed asynchronously.

4.2 Markov Chain Results

We first investigate the convergence of asynchronous parallel Jacobi when solving systems of linear equations arising from the steady-state solution of Continuous Time Markov Chains (CTMCs). In this context A is an explicitly-stored transposed generator matrix of the CTMC, x is a probability vector ($\sum_{i=1}^{n} x_i = 1$) and $b = 0$. Note that here convergence occurs even though $\rho(|M|) = 1$ [7].

Table 2: CTMC matrix results.

Synchronous Processors	Iterations			Execution Time (s)
	Min.	Average	Max.	
0	680	735	944	168
2	628	742	965	181
4	636	763	989	205
6	632	747	949	182
8	660	730	896	161
10	661	746	1004	179
12	680	748	1067	183
14	705	755	987	187
16	714	763	1046	199
18	722	749	1061	179
20	726	762	1191	201
22	731	758	1422	219
24	733	733	733	154

Table 2 presents the number of iterations and overall runtime observed for increasing numbers of synchronising processors. Note that 0 synchronising processors corresponds to fully asynchronous parallel solution, and 24 synchronising processors corresponds to fully synchronous. All of the results were gathered on the Computational Shared Facility (CSF) at the University of Manchester using 2 nodes comprising 24 processors in total. Each CSF node has two 6-core Xeon X5650 2.66GHz processors and 48GB of memory, and they are connected with Infiniband. A has 1 639 440 rows and 13 552 968 nonzeros, and was partitioned across the participating processors by assigning a block of contiguous rows to each. Iterations were continued until $\frac{||x^{(k)}-x^{(k-1)}||_\infty}{||x^{(k)}||_\infty} < \varepsilon$; in these experiments we set $\varepsilon = 10^{-06}$. We performed 5 repeated runs for each number of synchronising processors, and in each case we report the minimum and maximum number of iterations observed across all 5 runs and all processors, and the mean of the 5 iteration counts and runtimes.

As in Case Study 1, we observe that fully asynchronous execution requires more iterations to converge than fully synchronous. This observation can be attributed to the use of outdated vector elements delaying the achievement of convergence and so requiring processors to do more work. Unlike in Case Study 1, however, we do not see a reduction in runtime for asynchronous execution versus synchronous; indeed, here the fully synchronous execution is actually the fastest. We believe that this is because, for this size of matrix on this architecture, the computation time completely dominates the communication time. As such, any saving in not synchronising is lost because of the need to perform extra iterations to achieve convergence.

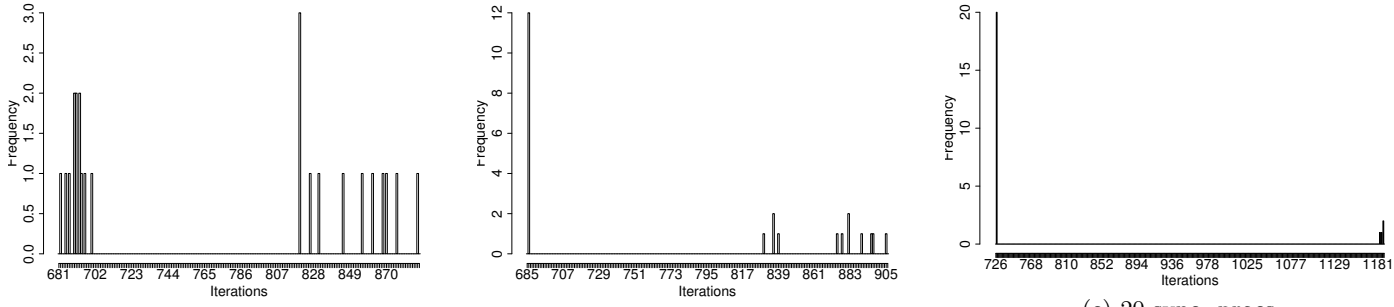
The figures presented in the table do not tell the full story, however. In order to reveal more about the behaviour of the implementation we have plotted the distribution of iteration counts for a variety of different matrix sizes. Figure 5 shows the iteration count distributions for a 1 639 440 row matrix (the same size as featured in Table 2). These are representative of the distributions observed over repeated runs, and display the behaviour we would expect: as the number of synchronising processors increases, the distribution of iterations becomes increasingly bi-modal with the first peak corresponding to the number of iterations performed by the synchronous processors and the higher iteration counts occurring on the asynchronous processors.

Figure 5(a) is interesting as it shows two groups of processors: one group of 12 performed fewer than 700 iterations and the other over 800. Examining the log files, we find that these correspond to MPI ranks 0-11 and 12-23 respectively. It would be interesting to know to which nodes these ranks were assigned, as we may be observing that one node was slower than the other. Had the processors been operating synchronously, the overall execution of the solver would have been limited by the speed of the slow node.

We do not always observe the expected behaviour, however. On an number of runs we found that the widely-spread iteration count distributions seen in the previous figure were no longer evident and instead all participating processors have carried out approximately the same number of iterations regardless of whether they were operating synchronously or asynchronously. It is unclear if this behaviour becomes more likely to occur as matrix size grows, or whether it is a manifestation of the interference that occurs from other jobs running on the shared system. Larger systems will take longer to solve and so are more vulnerable to clashing with other jobs.

4.3 Random Matrix Results

Table 3 shows the results of solving a set of 200 000 000 linear equations (the matrix A had 2 000 045 816 nonzero entries in total). A was diagonally dominant with off-diagonal elements randomly sampled from a uniform distribution on $[0, 1)$. The column indices of the off-diagonal elements were also randomly selected from a uniform distribution across all



(a) 0 sync. procs. (b) 12 sync. procs. (c) 20 sync. procs.

Figure 5: 1 639 440 row matrix iteration count distributions (0-20 sync. procs.).

Table 3: Random matrix results.

Synchronous Processors	Iterations			Execution Time (s)
	Min.	Average	Max.	
0	113	118	127	5709
2	73	100	136	4975
4	60	98	130	4663
6	83	110	133	5264
8	62	95	150	4323
10	74	115	142	5563
12	77	108	138	5451
14	141	141	143	6630
16	116	134	174	6676
18	112	139	170	6739
20	137	142	201	6629
22	127	142	288	6841
24	216	216	216	7205

columns, excluding the diagonal. The right-hand side vector b was constructed such that $x_i = 1\ \forall i$ and we took $\varepsilon = 10^{-08}$. The results were gathered on the CSF at the University of Manchester using 24 processors over 5 repeated runs.

We note that fully asynchronous execution is faster and requires fewer iterations than the fully synchronous case. This matches our experience in Case Study 1, but differs from what was observed on the CSF for the smaller CTMC matrix. It is interesting, however, that here the fully asynchronous case is actually slower than the mixed cases up to and including those with 12 synchronous processors. The sudden increase in runtimes and iteration counts that occurs at 14 synchronous processors is probably due to the fact that at this point it is guaranteed that the synchronising subset will span both participating nodes (each of which has 12 cores) and so communication across the network must occur. After this point the observed execution times are largely the same as the number of synchronising processors increases, and only increase again when the fully synchronous case is reached.

It may seem somewhat surprising that the fully asynchronous execution requires fewer iterations than the fully synchronous version. One possible explanation might be the existence of a Gauss-Seidel effect in the asynchronous version, as has been observed elsewhere [3]. In the fully synchronous case we have implemented pure Jacobi where locally-updated vector elements are not used in calculations until the next iteration. In contrast, asynchronous processors use remotely-computed updated vector elements as soon as they are received, which means that they may be using newly-computed elements within an iteration. The fully synchronous execution could perhaps converge faster if newly-computed local vector elements were used as soon as they were available.

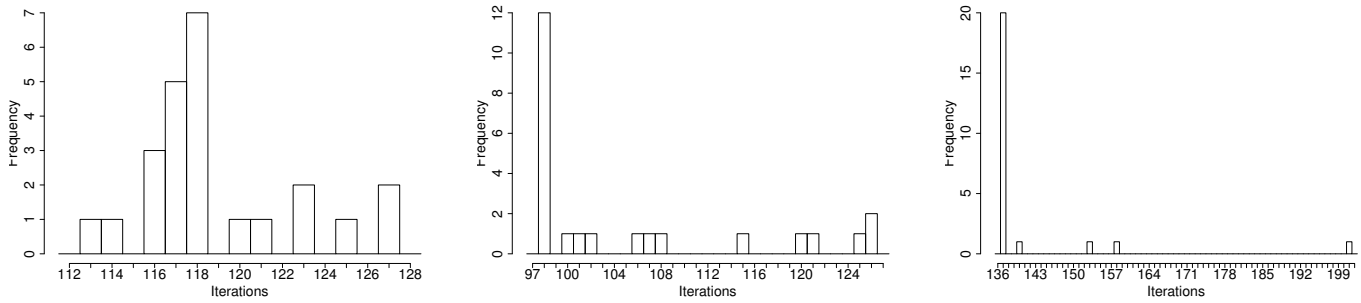
Figure 6 shows the iteration count distributions for a number of these experiments. Again, we observe that as the number of synchronising processors increases the spread of the iteration counts is increased. This is most marked in the case of 20 synchronous processors (Figure 6(c)).

We also observed unusual behaviour for some executions of the solver, exactly as we did for the CTMC matrix experiments. Most surprising were the cases for which all participating processors (synchronous and asynchronous) performed exactly the same number of iterations. In one case with 22 synchronous processors we observed that all processors converged to the solution after 127 iterations, when we would have expected the spread of iteration counts to be very wide indeed. This number of iterations is a little more than half the number required by the fully synchronous case; we theorise that this is a manifestation of the Gauss-Seidel effect noted above.

5. CONCLUSION

We have implemented a number of variations of asynchronous parallel Jacobi and evaluated their behaviour experimentally on several scientific problems. We observed differing behaviour between the two case studies and also within Case Study 2, which suggests that the performance of parallel asynchronous Jacobi is highly implementation, problem and architecture-dependent. For larger matrices, we observed that the fully asynchronous versions ran faster than the fully synchronous versions, especially for large numbers of processors on HECToR, and this reinforces our contention that asynchronous algorithms will be vital in efficiently harnessing the computing power of parallel machines with millions of cores.

Comparing iteration counts, our experiments showed that using out-of-date information can lead to an increase in the number of iterations required to reach convergence. In Case Study 2, however, we also observed that asynchronous parallel Jacobi can sometimes require fewer iterations to converge than its synchronous counterpart. Varying the amount



(a) 0 sync. procs. (b) 12 sync. procs. (c) 20 sync. procs.

Figure 6: Random matrix iteration count distributions (0-20 sync. procs.).

of synchronisation could have surprising effects, for example reducing runtime in the random matrix case when small numbers of synchronous processors were used. Some of this behaviour only became apparent when we looked into the distributions of iteration counts rather than just the means.

In the future we intend to conduct further experiments to investigate those effects which we noted but could not explain, for example the lack of sudden increases in the residuals for *async_racy* and the extension of the Gauss-Seidel effect to synchronous parallel Jacobi. We hope to combine the results of all these experiments with appropriate theoretical analysis to improve our understanding of the convergence of asynchronous Jacobi and thereby to assess its suitability for the massively parallel machines of the future. In addition, having implemented asynchronous Jacobi using MPI (both single-sided and two-sided messaging), we intend to explore a range of other parallel programming paradigms such as OpenMP, SHMEM, or Partitioned Global Address Space (PGAS) languages, to understand both performance and suitability for expressing asynchronous algorithms.

Acknowledgements

This work made use of the facilities of HECToR, the UK's national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme.

The authors would like to acknowledge the assistance given by IT Services for Research at the University of Manchester regarding the use of the Computational Shared Facility.

The authors' work was supported by EPSRC grants EP/I006680/1 and EP/I006702/1 "Novel Asynchronous Algorithms and Software for Large Sparse Systems". The work of the fourth author was also supported by EPSRC grants EP/E050441/1 "CICADA: Centre for Interdisciplinary Computational and Dynamical Analysis" and EP/I006702/1, and European Research Council Advanced Grant MATFUN (267526).

6. REFERENCES

[1] G. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the Association for Computing Machinery*, 25(2):226–244, 1978.

[2] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, 1989.

[3] D. Bertsekas and J. Tsitsiklis. Some aspects of parallel and distributed iterative algorithms – a survey. *Automatica*, 27(1):3–21, 1991.

[4] J. Bull and T. Freeman. Numerical performance of an asynchronous Jacobi iteration. In *Proceedings of the 2nd Joint International Conference on Vector and Parallel Processing (CONPAR'92)*, pages 361–366, 1992.

[5] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2:199–222, 1960.

[6] D. de Jager and J. Bradley. Extracting state-based performance metrics using asynchronous iterative techniques. *Performance Evaluation*, 67(12):1353–1372, 2010.

[7] D. Szyld. The mystery of asynchronous iterations convergence when the spectral radius is one. Technical Report 98-102, Department of Mathematics, Temple University, Philadelphia, PA, October 1998.