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

Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP

Iain Bethune[1], J. Mark Bull[1], Nicholas Dingle[2], and Nicholas Higham[2]

[1]EPCC, The University of Edinburgh

[2]School of Mathematics, University of Manchester

May 13, 2013

Running head: "Perf analysis of async Jacobi"

Iain Bethune[1], J.Mark Bull

EPCC, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK

{ibethune,markb}@epcc.ed.ac.uk

Nicholas Dingle, Nicholas Higham

School of Mathematics, University of Manchester, Oxford Road, Manchester, M13 9PL, UK

{nicholas.dingle,nicholas.j.higham}@manchester.ac.uk

[1]Corresponding Author

Abstract

Ever-increasing core counts create the need to develop parallel algorithms that avoid closely-coupled execution across all cores. In this paper we present performance analysis of several parallel asynchronous implementations of Jacobi's method for solving systems of linear equations, using MPI, SHMEM and OpenMP. In particular we have solved systems of over 4 billion unknowns using up to 32,768 processes on a Cray XE6 supercomputer. We show that the precise implementation details of asynchronous algorithms can strongly affect the resulting performance and convergence behaviour of our solvers in unexpected ways, discuss how our specific implementations could be generalised to other classes of problem, and how existing parallel programming models might be extended to allow asynchronous algorithms to be expressed more easily.

1 Introduction

Modern high-performance computing systems are typically composed of many thousands of cores linked together by high bandwidth and low latency interconnects. 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 scalar reductions may be required by the algorithm. This creates barriers beyond which computation cannot proceed until all participating processors have reached that point, i.e. the computation is globally sychronised at each iteration. Such approaches are unlikely to scale to millions of cores.

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 [4, 5, 9] and the practical [3, 6, 11] aspects of such algorithms. To reason about these algo-

rithms we need to understand what drives the speed of their convergence, but existing results merely provide sufficient conditions for the algorithms to converge, and do not help us to answer some of questions arising in the use of asynchronous techniques in large, tightly coupled parallel systems of relevance to Exascale computing. Here, we look for insights by investigating the performance of the algorithms experimentally.

Taking Jacobi's method, one of the simplest iterative algorithms, we implement one traditional sychronous and two asychronous variants, using three parallel programming models - MPI [17], SHMEM [10], and OpenMP [18]. We comment on the programmability or ease of expression of asynchronous schemes in each of the programming models, and based on our experiences how they might be adapted to allow easier implementation of similar schemes in future. We investigate in detail the performance of our implementations at scale on a Cray XE6, and discuss some counter-intuitive properties which are of great interest when implementing such methods. Finally, we consider how our communication schemes could be applied to other iterative solution methods.

2 Jacobi's Method

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

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

The $x_i^{(k)}$, $i = 1{:}\,n$, 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 the spectral radius $\rho(|M|) < 1$ with $M = -D^{-1}(L + U)$ where D, L, U are the diagonal and strictly lower and upper triangular parts of A. In contrast, the synchronous version of Jacobi's method converges if $\rho(M) < 1$ [9].

3 Implementation

To investigate the convergence and performance of Jacobi's method, we have implemented three variants of the algorithm - Synchronous Jacobi (*sync*) and two Asychronous Jacobi (*async* and *racy*). The *sync* algorithm falls into the SISC (Synchronous Iterations Synchronous Communications) classification proposed by Bahi *et al* [2] i.e., all processes carry out the same number of iterations in lock-step, and communication does not overlap computation, but takes place in a block at the start of each iteration. The *async* and *racy* algorithms are AIAC (Asynchronous Iterations Asynchronous Communications), since processes proceed through the iterative algorithm without synchronisation, and so may iterate at different rates depending on a variety of factors (see Section 6.2). Communication may take place at each iteration, but is overlapped with computation, and crucially, the receiving process will continue iterating with the data it has, incorporating new data

as it is received. We note that these schemes are more general than simply overlapping communication and computation within a single iteration (e.g. using MPI non-blocking communication) as messages may be (and in general, will be) received at at different iteration to which they are sent, removing all synchronisation between processes. In *async* we ensure that a process only reads the most recently received complete set of data from another process, i.e., for each element $x_i^{(k)}$ received from a particular process p, we ensure that all such x_i read during a single iteration were generated at the same iteration on the sender. In *racy* this restriction is relaxed, allowing processes to read elements x_i potentially from multiple different iterations. As is shown in Section 5, this reduces communication overhead, but relies on the atomic delivery of data elements to the receiving process, so that every element we read existed at some point in the past on a remote process. Investigation of this type of communication scheme has not been reported in the literature to date.

Instead of a general Jacobi solver with explicit A matrix, we have chosen to solve the 3D diffusion problem $\nabla^2 u = 0$ using a 6-point stencil over a 3D grid. This greatly simplified the implementation, since there was no load-imbalance or complex communication patterns needed, and it allowed us to easily develop multiple versions of our test code. 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, allowing much easier comparison of the relative scalability of each communication scheme, since the computational cost per iteration in each case is identical. 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 $e^{-((0.5-x)^2+(0.5-y)^2)}$, where the global grid is $0 \leq x, y, z \leq 1$. Physically, this could be thought of as a region of concentrated pollutant entering a volume of liquid or gas, and we solve for the steady state solution as the pollution diffuses over the region. The interior of 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 practice a smaller error tolerance might be chosen to stop the calculation, but this allows us to clearly see trends in convergence without the calculation taking excessively long. For our system, the iteration matrix $M \geq 0$ so $\rho(|M|) = \rho(M)$. The spectral radius was found to be strictly less than one, so both synchronous and asynchronous Jacobi's algorithm are guaranteed to converge.

3.1 MPI

MPI is perhaps the most widely used programming model for HPC today, and the de-facto standard for implementing distributed memory applications. The MPI standard [17] contains both two-sided (sender and receiver cooperate to pass messages), and single-sided (sender puts data directly into the receiver's address space without explicit cooperation) communication calls. However, as single-sided MPI usually gives poor performance [16], we choose to use only two-sided and collective communication - representing the typical programming model of parallel HPC applications today.

In common with many grid-based applications, when implemented using a distributed memory model 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' 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 overall structure of the program is shown in Figure 1, which is common between all three variants of the algorithm. However, the implementation of the halo swap and the global residual calculation vary as follows:

```
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 Synchronous Jacobi using MPI or SHMEM

3.1.1 sync

Halo swaps are performed using `MPI_Issend` 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.

3.1.2 async

This implementation allows multiple halo swaps to be 'in flight' at any one time (up to R between each pair of processes). This is done by means of a circular buffer storing `MPI_Requests`. When a process wishes to send halo data it uses up one of the R MPI requests and sends the halo data to a corresponding receive buffer on the neighbouring process. If all R 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 with the computation, until one or more of the outstanding sends has completed. We chose $R = 100$ for initial experiments and consider the effects of different values in Section 6.3. On the receiving side, a process will check for arrival of messages and, if new data has arrived, copy the newest data from the receive buffer into the halo cells of its u array (discarding any older data which may also have arrived). If no new data was received during that iteration the calculation continues using whatever data was already in the halos. By using multiple receive buffers (one for each message in-flight) we ensure that the data in the u array halos on each process is a consistent image of halo data that was sent at some iteration in the past by the neighbouring process.

In addition, since non-blocking collectives do not exist in the widely-implemented MPI 2.1 standard (although they exist in MPI 3.0) 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 number being $2 \log_2 p$), as soon as a process receives the global residual it immediately starts another reduction. In fact, even on 32768 cores (the largest run size tested), this asynchronous reduction took only around 50 iterations to complete. Compared with the synchronous reduction (every 100 iterations), this gives the asynchronous implementations a slight advantage in potentially being able to terminate sooner after convergence is reached. This could of course also be achieved in the synchronous case, but at a higher communication cost.

One side-effect of the asynchronous reduction is that by the time processes receive a value for the global residual indicating that convergence is reached, they will have performed some number of further iterations. Since convergence in the asynchronous case is not necessarily monotonic (see Section 6.2), it is possible that the calculation may stop in an unconverged state (although we did not observe this in practice in any of our tests). In addition, since the residual is calculated piecewise locally, with respect to current halo data, rather than the data instantaneously on a neighbouring process, the converged solution may have discontinuities along process' grid boundaries. To overcome this we propose that on reaching asychronous convergence a small number of synchronous iterations could then be performed to guarantee true global convergence, but we have not yet implemented this extension to the algorithm.

3.1.3 racy

While similar in spirit to the *async* algorithm, here we introduce a performance optimisation in that instead of having R individual buffers to receive halo data, we use a single buffer to which all in-flight messages are sent, with the aim of reducing the memory footprint of the buffers and hence improve cache performance of the halo swap operation, and allowing access to the most recently received data (even if the message has only partially arrived). This is a deliberate race condition because 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. If access to individual elements is not atomic (e.g. in the case of complex numbers - a pair of floats - or more complex data structures), such a scheme would not be appropriate as we might read an element which is incorrect, composed of partial data from multiple iterations.

3.2 SHMEM

As a second programming model for evaluation, we chose SHMEM as a representative of the Partitioned Global Address Space (PGAS) paradigm. Each process has its own local address space, which allows SHMEM codes

to run on massively parallel distributed memory machines, but can directly access memory locations in remote processes' memory spaces via single-sided API calls. There are many SHMEM implementations available and effort is ongoing to standardise these through the OpenSHMEM project[7]. In our case we use Cray's SHMEM library which is available on the Cray XT/XE series, although it is not yet compliant with Version 1.0 of the OpenSHMEM standard[1]. Conceptually, the single-sided nature of SHMEM should map well to our asynchronous implementation of Jacobi's algorithm, and since on the Cray XE remote memory access (RMA) is supported directly by the network hardware, we should expect good performance.

3.2.1 sync

We implement synchronous Jacobi with SHMEM following the same structure as the MPI implementation (Figure 1). However, whereas we use the implicit pair-wise synchronisation of `MPI_Issend` for the MPI implementation, there is no direct equivalent available in SHMEM, only a global barrier (`SHMEM_barrier_all`), which does not scale well to large numbers of processes. To achieve pair-wise sychronisation in SHMEM we use the following idioms. On the sending side:

```
call shmem_double_put(remote_buf, ...)
call shmem_fence()
local_flag = 1
call shmem_integer_put(remote_flag, 1, ...)
...
call shmem_wait_until(local_flag, &
                      SHMEM_CMP_EQ, 0)
```

And on the receiver:

```
call shmem_wait_until(remote_flag, &
                      SHMEM_CMP_EQ, 1)
read data from buffer
remote_flag = 0
call shmem_integer_put(local_flag, 0, ...)
```

In this way the sender cannot progress from the halo swap until the receiving process has acknowledged receipt of the data. The receiver does not read the receive buffer until the data is guaranteed to have arrived since the `shmem_fence()` call ensures the flag is not set until the entire data is visible to the receiving process.

The global reduction is almost identical to the MPI implementation, using the routine `shmem_real8_sum_to_all()` in place of the `MPI_Allreduce`.

3.2.2 async

Implementing the asynchronous variant in SHMEM has similar difficulties to sychronous Jacobi. However, in this case, instead of waiting for a single flag to be set before reading the buffer, here the process tests the next flag in a circular buffer of R flags, and if it is set, reads the corresponding receive buffer, before notifying the receiver that the buffer has been read and can be reused. The asynchronous reduction is performed using a binary tree, exactly as for the MPI implementation, but using SHMEM point-to-point synchronisation via the `put-fence-put` method.

3.2.3 racy

This variant is very straightforward to implement using SHMEM. At each iteration, a process initiates a `shmem_put()` operation to send its halo to the appropriate neighbour processes, and carries on. The receiver reads data from its receive buffer every iteration without any check to see if the data is new or any attempt to ensure the buffer contains data from a single iteration. We recognise that this is not a direct equivalent to the MPI *racy* code (as is the case for *sync* and *async*) since there is no upper limit on the number of messages in flight, but it seems to be the most natural way of expressing an asynchronous algorithm using SHMEM.

3.3 OpenMP

As one of the objectives of this work is to compare the ease with which asynchronous algorithms can be expressed in a variety of programming models, we chose to also implement the three variants of Jacobi's method using OpenMP. As a shared memory programming model, OpenMP is limited in scalability to the size of a shared-memory node (in our case, 32 cores), but direct access to shared memory without API calls (such as SHMEM), or explicit message passing (MPI), should map well to asynchronous algorithms. Hybrid programming using MPI between nodes and OpenMP within a node is also becoming a popular programming model. Because of the relatively small number of cores we cannot draw direct performance comparisons with the other two implementations so we chose to use a 1-D domain decomposition using OpenMP parallel loops rather than an explicit 3D domain

decomposition. The global domain increases in the z-dimension with the number of threads to maintain weak scaling. Because of the shared memory nature of OpenMP, the explicit halo swap step is unecessary, and is replaced with direct access to the neighbouring cells during the calculation loops (see Figure 2).

```
do

   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 2: Pseudocode of parallel Synchronous Jacobi using OpenMP

3.3.1 sync

In the synchronous case the global residual is simply calculated using an OpenMP reduction clause, formed by the summation of each thread's local residual. Once this is done, a single thread normalises the shared residual value, and all threads test to see if convergence has been obtained. The two calculation loops are done in separate OpenMP parallel loops, with an implicit synchronisation point at the end of the first loop, ensuring that each process reads consistent data from the neighbouring threads' u array, while u_{new} is being updated.

3.3.2 async

In this variant, we allow each thread to proceed though its iterations without global synchronisation. To avoid a race condition, whenever a thread reads neighbouring threads' data, the neighbouring thread must not modify that data concurrenently. In addition, we ensure (as per the MPI and SHMEM implementations) that when we read boundary cells from a neighbouring thread we read a complete set of values from a single iteration. To do this, we create OpenMP lock variables associated which each inter-thread boundary, and the lock is taken while a thread reads or writes to data at the boundary of its domain. To allow threads to progress asynchronously, we split the loops over local points into three parts:

```
call omp_set_lock(ithread)
   loop over lower boundary plane
call omp_unset_lock(ithread)

loop over interior points

call omp_set_lock(ithread+1)
   loop over upper boundary plane
call omp_unset_lock(ithread+1)
```

Each of these parts can proceed independently, and with 50 planes of data per thread, we expect lock collisions to be infrequent. This model is used for both calculating local residuals and updating the u and u_{new} arrays.

To assemble a global residual asynchronously, instead of summing local contributions, we keep a running global (squared) residual value in a shared

variable, and each thread calculates the difference in its local residual since the last iteration, and subtracts this difference from the shared global residual, protecting access via an OpenMP critical region. To avoid a race condition during the convergence test, the thread also takes a private copy of the global residual inside the same critical region. This critical region could be replaced by an atomic operation in future versions of OpenMP which support atomic add-and-capture. If the local copy of the global residual is seen to be less than the termination criteria, then a shared convergence flag is set to true. Any thread which subsequently sees the flag set to true exits the calculation loop.

3.3.3 racy

As for SHMEM, allowing deliberate race conditions simplifies the code somewhat. In this case we can simply remove the locks present in the *async* implementation. We can also remove the critical sections around the global residual calculation - simply an OpenMP `atomic` subtraction is enough to ensure no deltas are 'lost'. Here we are again relying on hardware atomicity of reads and writes to individual data elements (double precision floats) to ensure that we always read valid data from a particular iteration on a per-element basis.

One of the aims of our study was to characterise the ease with which one can express asynchronous algorithms in each of the programming models, and secondly to understand how easy it is to convert an existing synchronous implementation into an asynchronous one.

MPI was found to be easy to work with, due in part to the authors' existing familiarity with it, as well as widely available documentation and published standard. While the synchronous algorithms are very easy to implement in MPI as the two-sided point-to-point semantics automatically provide the pairwise synchronisation needed, converting to an asynchronous implementation is challenging. Due to all asynchrony being achieved by the use of 'non-blocking' communications such as `MPI_Issend`, a great deal of extra book-keeping code has to be written to ensure all outstanding message handles are eventually completed, via corresponding calls to `MPI_Wait`. In addition, collective communications must be replaced by hand-written non-blocking alternatives - in our implementation, over 60 lines of code in place of a single call to `MPI_Allreduce`. The addition of non-blocking collective operations was proposed [14] and has recently been standardised in MPI 3.0, but this is not yet widely implemented. In addition, termination of the asynchronous algorithm is problematic, since when processes reach convergence there may be sends outstanding for which no matching receive will ever by posted and vice-versa. MPI provides a subroutine `MPI_Cancel` which can be used to clean up any such outstanding communications, but in practice cancelling of outstanding sends (specified in MPI-2.2) was not supported by the Cray MPT implementation. It would be possible - and necessary for use in a real code - to have processes exchange data on the number of outstanding sends remaining, and post matching `MPI_Recv` calls to terminate in a clean manner.

By contrast, SHMEM was found to be much more challenging to work with, mostly due to lack of good documentation (only `man` pages, which by

nature miss the bigger picture of how to use the API as a whole) and lack of adherance to the published standard. The interface is much simpler than MPI, but lacks point-to-point synchronisation, making synchronous Jacobi difficult to implement without using a global barrier. We had to implement explicit acknowledgement of message receipt, something which MPI handles within the library. Our *async* implementation was similarly complicated by the need for the sender to know when a message had been delivered in order to target future `shmem_put` operations at the correct buffers. However, one advantage of the single-sided model in SHMEM is that there is no need to 'complete' non-blocking operations as per MPI - by default remote writes complete asynchronously to the caller, and any resources allocated to process the message are freed automatically by SHMEM. This made the *racy* version very simple to implement, as no book-keeping code is required. SHMEM also suffered the same disadvantage as MPI with respect to non-blocking collective operations. Perhaps due to the fact that SHMEM is less used and less tested than MPI, we found a bug where remote writes to an area of memory being concurrently read by the remote process would sometimes never be seen at the remote side (despite further writes completing beyond a `shmem_fence`, which guarantees visibility of prior writes). This was observed when using the PGI compiler, and is incorrect behaviour according to the OpenSHMEM standard (although as mentioned earlier, the Cray implementation is non-compliant) . We switched to using the gfortran compiler, which behaved as we expected.

Finally, OpenMP was much easier for implementing both synchronous and asynchronous variants. The directives-based approach resulted in rela-

tively little code change over a serial implementation, and even when locking was required for *async* this was not as difficult as the buffering schemes implemented in MPI and SHMEM. One caveat for OpenMP is that the default `static` loop schedules map well onto a 1-D domain decomposition, since we simply divide up the grid into equal chunks, one per thread. Implementing a 2-D or 3-D decomposition would be possible but would require more explicit control of the loop bounds, somewhat reducing the elegance of the OpenMP implementation.

4.2 A proposed API for asynchronous communication channels

Based on the implementation experiences described above we derived the following set of requirements for a simple API designed to provide the functionality needed to implement our *async* and *racy* algorithms on a distributed memory system:

- The ability to initiate and terminate bidirectional asynchronous communication channels between pairs of processes, specifying the maximum number of in-flight messages on the channel (in each direction), the size of each message, and *async* or *racy* communication mode (with the semantics described in Section 3).

- End-to-end flow control for point-to-point communication i.e. messages can only be sent concurrently through the channel up to the specified maximum number and the sender may only insert further messages only when the message has been acknowledged by the re-

cieving end of the channel (although it may not have been read by the application).

- In *async* mode the programmer must be able to explicitly receive new data from the channel to guarantee that a single, entire message is delivered to the receive buffer. In *racy* mode, the channel may write new data directly into the receive buffer at any time.

- Non-blocking global reduction operations (for convergence check).

- Handles for individual communication operations are not exposed to the programmer - as in SHMEM, completion of underlying communications is handled below the API. In particular the implementation in either SHMEM or MPI (or potentially other programming models) should be transparent to the programmer.

The non-blocking `MPI_IAllReduce()` operation included in MPI 3.0 is ideally suited to our application, and we do not propose a replacement. Even if using SHMEM, so this routine could still be used by intermixing MPI and SHMEM operations (assuming the installed MPI library implements MPI 3.0). If not, one could easily implement the operation manually, as we describe earlier. To meet the other requirements we propose the following API (in C-like pseudocode):

- ```
  create_channel(int *handle, int rank, int message_size,
                 void *sendbuf, void *recvbuf,
                 int max_messages, bool racy)
  ```
  ```

Sets up a asynchronous communication channel between the caller and process `rank`, with `max_messages` allowed in-flight in each direction and communication sematics either `racy` or not. The `sendbuf` and `recvbuf` must be pointers to buffers of at least `message_size` bytes, which will be used to read and write data during send and recv operations. A call to `create_channel` will block until the process `rank` makes a matching call. On exit a unique integer handle for the channel is returned in `handle` and the channel is ready to be used.

- `destroy_channel(int handle)`

  Destroys the communication channel `handle`, ensuring outstanding communications are completed. Blocks until a matching call is made by the process at the other end of the channel.

- `bool can_send(int handle)`

  Returns true if there are fewer than `max_messages` in-flight i.e. a subsequent send operation can be completed immediately, otherwise false.

- `send(int handle)`

  Sends `message_size` bytes read from `sendbuf` via the specified channel. If the channel is full, the call blocks until the message can be sent. Callers should use `can_send()` to ensure this is not the case. On return, the `sendbuf` is free to be reused.

- `bool can_recv(int handle)`

In *async* mode returns true if new messages have arrived in the channel since the latest call to `recv()`, otherwise false. In *racy* mode always returns true.

- `recv(int handle)`

  Receives the newly arrived data from the channel. If the communication mode is *async* `recvbuf` is filled with `message_size` bytes of data from the most recently arrived message. Any other messages that may have arrived since the previous call to `recv()` are discarded. In *racy* mode, calls to `recv()` return immediately as data may be written to `recvbuf` at any time by the channel.

An implementation of this API would be of assistance to programmers who wish to express this type of asynchronous point-to-point communication in similar linear solvers (see Section 6.4) and the concept may be useful in other contexts where iterative schemes with point-to-point communication are used.

5 Performance

To investigate the performance of our implementations of Jacobi's algorithm, we used HECToR, a Cray XE6 supercomputer. HECToR is composed of 2816 compute nodes, each with two 16-core AMD Opteron 2.3 GHz processors and 32 GB of memory. A 3D torus interconnect exists between nodes, which are connected to it via Cray's Gemini router chip. We performed runs using 32 cores (4 million unknowns) up to a maximum of 32768 cores (4.1

billion unknowns) in powers of two. We do not report any runs on fewer than 32 cores, since with a partially-occupied node, each core has access to a greater fraction of the shared memory bandwidth on a node, and thus the performance of the grid operations increases, rendering weak-scaling comparison impossible. To limit the amount of computational times used, we performed only a single run for each core count and variant of the code, but by testing a wide range of core counts we can look at performance trends without variability in individual results being a concern. A sample of results for MPI (Table 1), SHMEM (Table 2), and OpenMP (Table 3) are shown below:

Processes	Iterations			Execution
Version	Min.	Mean	Max.	Time (s)
32				
sync		9100		28.4
async	10737	11245	12231	33.8
racy	8952	9307	10140	27.7
512				
sync		37500		132.1
async	36861	44776	51198	146.9
racy	32377	38763	44623	126.4
8912				
sync		68500		247.1
async	60612	82381	96700	272.5
racy	61453	77952	90099	264.9
32768				
sync		112000		405.2
async	101053	136337	165366	454.4
racy	86814	118967	144902	419.4

Table 1: HECToR MPI results summary

Firstly it is worth stating that our results are in broad agreement with previous work ([6]), that the asynchronous algorithms perform better than synchronous in some cases, and worse in others.

More specifically, we see that for MPI, the *async* implementation is sig-

Processes **Version**	Iterations Min.	Mean	Max.	Execution Time (s)
32				
sync		9100		27.6
async	8724	9136	10784	26.7
racy	8496	9088	10725	26.3
512				
sync		37500		130.1
async	32356	38285	43178	117.9
racy	32377	38416	46291	117.9
8912				
sync		68500		251.5
async	32612	67417	77382	215.5
racy	47231	70087	80620	219.3
32768				
sync		112000		491.1
async	11265	229266	331664	858.5
racy	10173	231919	314208	817.6

Table 2: HECToR SHMEM results summary

Threads **Version**	Iterations Min.	Mean	Max.	Execution Time (s)
32				
sync		3487		11.1
async	3383	3456	3530	7.4
racy	3402	3475	3524	7.5

Table 3: HECToR OpenMP results summary

nificantly worse than *racy* in all cases, so much so that it is also slower than *sync*. *Racy* was found to outperform the synchronous version for all but the largest runs.

For SHMEM, there is little difference between the two asynchronous implementations, and both are consistently faster than the synchronous code, except for the 32,768 core runs, where the performance drops off dramatically (see Section 5.1 for further discussion). Comparing MPI with SHMEM, we find that notwithstanding the performance problems at very high core counts, SHMEM is always somewhat faster than the equivalent MPI implementation.

As mentioned earlier, we cannot directly compare performance between OpenMP and the other two programming models, since the problem being solved is slightly different. However, it is interesting to note here that the asynchronous variants are consistently and significantly (63%) faster than the synchronous case.

The relative performance of each implementation is a function of two variables: the iteration rate - how much overhead does the communication scheme cause, and the number of iterations - how an asynchronous scheme affects progress towards convergence at each iteration. These factors are discussed in the following section.

5.1 Analysis

To understand the scalability of each implementation, we compared the mean iteration rate against the number of processes (Figure 3). This shows clearly that both synchronous variants iterate more slowly than either *async*

or *racy*. The MPI asynchronous codes have similar performance except at very high process counts, and the asynchronous SHMEM implementations are fastest of all, indicating that where an efficient implementation exists (such as on the Cray), SHMEM can provide lower overhead communications than MPI.

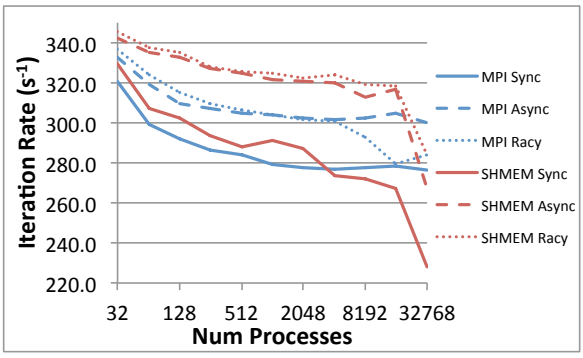
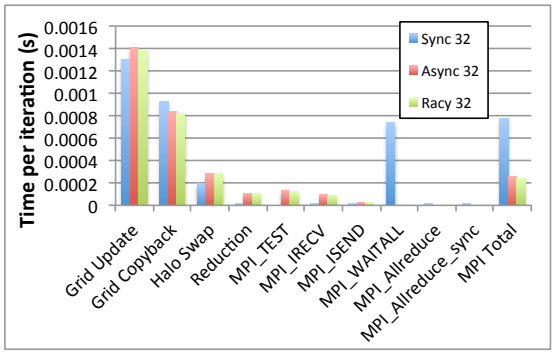


Figure 3: Comparing iteration rates of MPI and SHMEM implementations

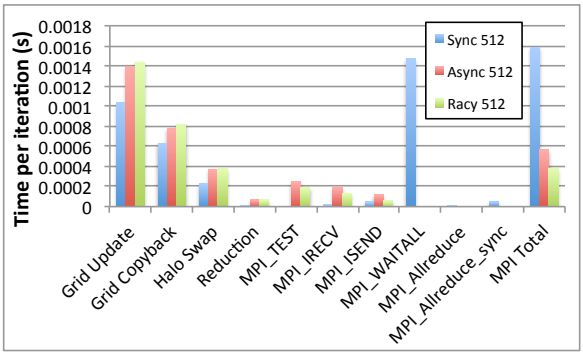
To understand what drives these differences, we performed detailed profiling of our codes using the Cray Performance Analysis Toolkit (CrayPAT). Figure 4 shows the mean time per iteration taken by the various steps of the algorithm, as well as the associated communications cost. Very similar data was also seen for the SHMEM implementation, but is omitted for the sake of brevity. By comparing the three graphs, we see that the time taken for grid operations is essentially constant as the number of processes increases. In the synchronous case, however, there is a much larger amount of time spent in communication - principally `MPI_Waitall` - due to the halo swap communication. Surprisingly, the global sum for the convergence check is not a significant contribution, although on 8192 cores, the synchronisation cost of this operation starts to show up in the profile. We note that al-

though most of the communication (excepting the small contribution from the global sum) is nearest-neighbour only, the cost of this does in fact grow with the number of processes, although we might expect it to be constant. The reason for this is due to the mapping of our 3D grid of processes onto the physical interconnect topology. By default we simply assign the first 32 processes (which are assigned to the 3D grid in a row-major order) to the first node. This means that in general 'neighbours' on the 3D grid are likely to only be close to each other in the machine in one dimension, and the others may be significantly further away, depending on which nodes are allocated to the job. This effect increases with the number of processes. Investigation of more efficient mappings of the 3D grid onto the network topology (e.g. as discussed in [20]) might well increase overall performance for both synchronous and asynchronous implementations.

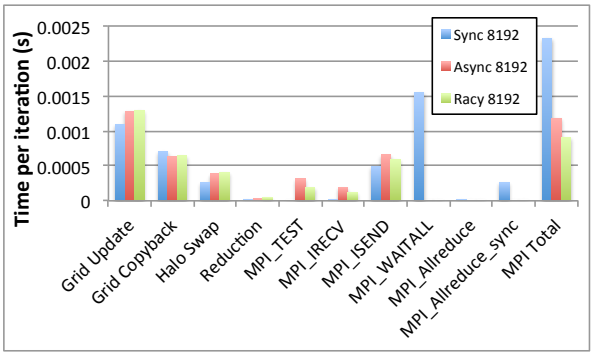
We have already shown that the SHMEM implementations suffer from very poor performance at high processor counts. This is illustated in Figure 5, where we can clearly see that the key cause of this is a sudden jump in the cost of `shmem_put` operations. The other components (grid operations, other SHMEM calls) are constant. Closer examination of the profiling data indicates that a small relatively small number of processes are suffering from very poor communications performance - in fact they are spending up to 3.2 times longer than average in `shmem_put` despite only making one-sixth as many calls as an average process. We have been unable to determine the cause of this behaviour, but have observed that it affects groups of 32 processes on the same node, and that these are outliers in terms of the number of iterations completed - since the communication cost is so large,



(a) 32 processes



(b) 512 processes



(c) 8192 processes

Figure 4: Profiles showing time per iteration for MPI on HECToR.

these processes complete very few iterations. The vast majority of processes complete a number of iterations more tightly clustered around the mean (see Figure 7(b)).

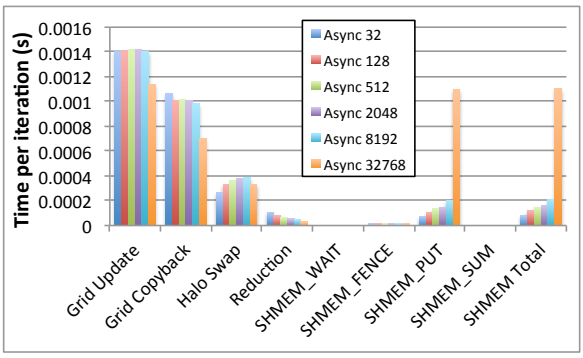


Figure 5: Profile of SHMEM *async* at varying process counts.

The other key factor affecting performance is the total number of iterations needed to converge. In the synchronous case, all processes execute the same number of iterations, but in the AIAC model, processes are free to iterate at different rates, so in Figure 6 we show only the mean number of iterations across all processes at the point where convergence is reached. However, as shown in Tables 1, 2 and 3, the maximum and minimum iteration counts are usually within 20% of the mean, and can be even higher in the case of SHMEM on 32,768 processes. The increasing trend in the graph is due to the fact that as we increase the number of processes, the problem size is also increased to match, thus it takes longer to reach convergence for larger problem sizes. The 'stepped' pattern in Figure 6 is due to the fact that this increase is done by first increasing the problem size in the x dimension, then the y and z dimensions at each doubling. Since the problem is anisotropic (the non-convergence is centred on the bottom z plane), some

changes cause very little increase in the amount of work needed to reach convergence.

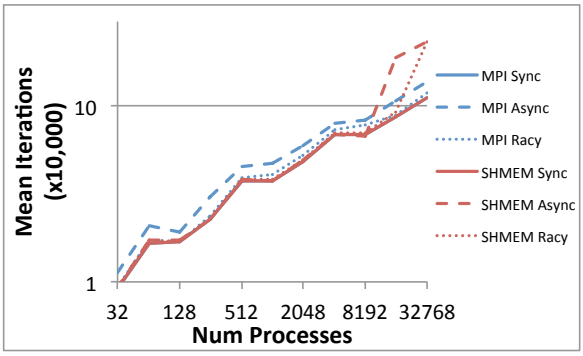


Figure 6: Comparing mean number of iterations to convergence in MPI and SHMEM implementations

All the asynchronous implementations take slightly more iterations to converge than than the synchronous implementation - as might be expected, since iterations working with out-of-date halo data will make less progress than if the data was current - but this effect is much more pronounced in the case of MPI. This appears to be as a result of the timing of MPI message arrival at the receiver (see Section 6.2). The mean iteration counts of SHMEM *async* and *racy* are seen to increase rapidly at high core counts as a result of the varying node performance mentioned earlier.

We plotted histograms of the number of iterations completed by each process for both MPI (Figure 7(a)) and SHMEM (Figure 7(b)), and observed some interesting behaviour. In both cases, the distribution of iterations has two main peaks - most of the processes making up the lower peak come from the first 8 cores on each node, with the remaining 24 cores performing more iterations and making up the higher peak. This is due to the physical

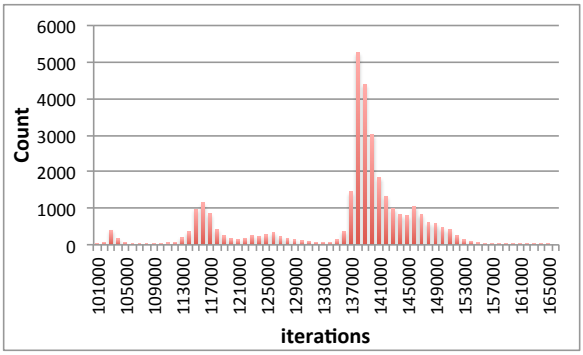
configuration of the compute node, where the connection to the network is attached directly to the first NUMA node (first 8 cores). As a result, message traffic to and from all the other cores on the node are routed over HyperTransport links through this NUMA node and onto the network, resulting in those nodes processing each iteration slightly slower. The main difference between MPI and SHMEM results, however, is the appearance of a 'long tail' of processes doing very few iterations in the SHMEM case. These are the same processes mentioned above that suffer very poor communication performance. The root cause of this is currently unknown, but effectively negates the performance benefits of using SHMEM seen at lower core counts when running at very large scale.

6 Discussion

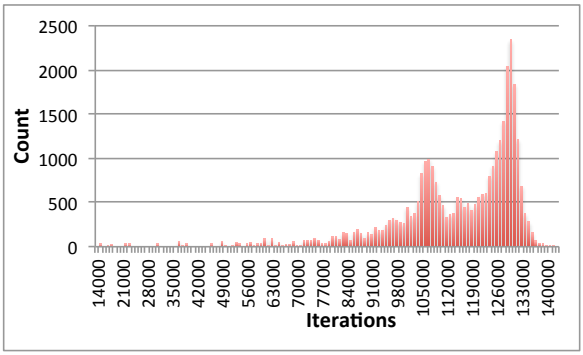
6.1 Performance Defects

One of the reasons asynchronous algorithms in general are of interest for future Exascale HPC applications is that the loose coupling between processes increases the scope for fault-tolerance. While we have not implemented any fault tolerance that would handle hardware faults, or the loss of a node, we have observed two cases of performance faults which our asynchronous algorithms were impervious to.

In the first case, we initially observed the performance of our *async* and *racy* codes to be approximately twice as fast as the *sync* version when running at large scale e.g. on 32768 cores. On closer investigation of profiling data from the *sync* runs, it was determined that there was one single CPU



(a) MPI



(b) SHMEM

Figure 7: Iteration histograms for MPI and SHMEM *async* on 32768 processes

core in the machine which was running much slower (by about a factor of two) than all other cores. The node reported no memory or hardware faults that could be observed by systems administration, and so was almost undetectable except by per-process profiling. Critically, the synchronous Jacobi method forces all processes to run at the same speed as this one slowest process, while the asynchronous variants continued to iterate as normal, with the single slow process lagging behind. This did not however affect the overall convergence by any noticable amount.

Secondly, when HECToR was upgraded from AMD Magny-cours to AMD Interlagos processors (HECToR phase 2b - phase 3), we observed a curious convergence profile which affected only the synchronous Jacobi variants (see Figure 8). The figure shows two clear regimes. In the first, the slower convergence rate is caused by the fact that each iteration rate is much lower than expected (up to approx 3.5 times), before suddenly switching to the normal speed, which carries on until termination. This initial period can slow down the synchronous algorithm by up around 50% (for Synchronous MPI on 32768 cores) but is noticable when using at least 512 cores. The duration of this slow-down is governed by the rate at which non-zero values propagate across the global domain. As long as exact zeros remain, due to the synchronous nature of the algorithm, all processes proceed at a slower rate. Use of hardware performance counters revealed that during this period, processes do more FLOPs per memory load than during the normal regime. We hypothesise that when exact zeros are involved in the calculation, this inhibits the use of vectorization (e.g. AVX). It is not clear whether this is due to the different CPU or the compiler since both

were changed during the upgrade. However, it is important to note that this effect does not affect the asynchronous codes in the same way. Due to the construction of the problem, the processes which contribute most to the residual are close to the source of the non-zero values, and so quickly begin iterating at a higher rate. Processes further away retain exact zeros (and run slower) for longer, but as there is no synchronisation between processes, they do not slow down the others, and make no noticable difference to the global rate of convergence. For the sake of fair comparison we modified the system by adding 1 to the boundary conditions and initialising the grid to 1. Thus the same amount of work has to be done to converge, but the effect of exact zeros is avoided.

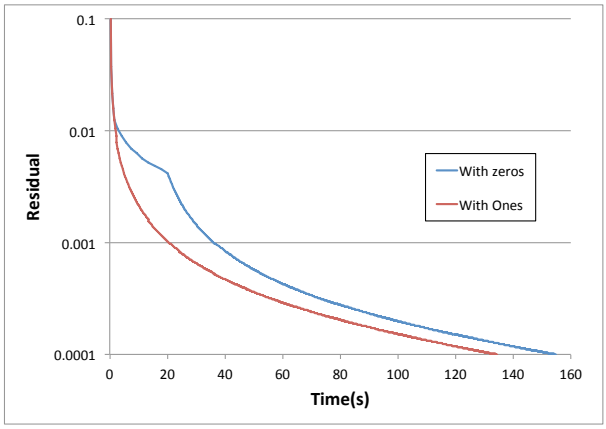


Figure 8: Convergence of Synchronous MPI on 1024 cores showing effect of exact zeros as initial conditions

Both of these performance effects are subtle, unexpected, and may have gone unobserved in a real application code. As larger systems are solved regularly in the future it becomes increasingly likely that software or hardware effects such as the above might cause some processes to run slower,

and global synchronisation needlessly slows down the remaining processes. We argue that this is further justification for use of asychronous algorithms in practice.

6.2 Convergence of asynchronous Jacobi

As discussed in Sections 2 and 3, for our systems with $\rho(|M|) = \rho(M) < 1$, both synchronous and asychronous variants of Jacobi's method are guaranteed to converge, but we are interested in how the use of asynchrony affects the convergence behaviour. From our tests we observe that though all of our implementations converged with broadly the same profile, closer examination revealed that while in the synchronous implementations convergence was monotonic, this is not the case for the asynchronous implementations (Figure 9). While halo data is sent at each iteration (subject to the limit of 100 outstanding messages) it turns out that messages are not received in such a regular manner, but rather tend to arrive in batches. For both SHMEM *async* and *racy*, and MPI *racy* we see small batches of around 5 messages arriving in the same iteration but with MPI *async* much larger batches of up to 60 messages arrive at once. Because data arriving on a process is out-of-date with respect to its current iteration, it increases the local contribution to the residual, which then decreases again as further local iterations are performed until more halo data arrives. This effect is clearly much more visible in the convergence profile of MPI *async*, but can also be seen in the other cases where the batching of messages is a much smaller effect.

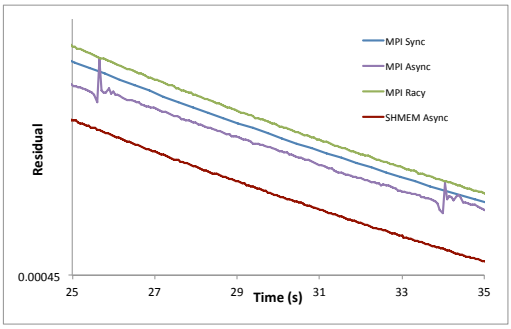


Figure 9: Detail of convergence of 1024 core runs on HECToR

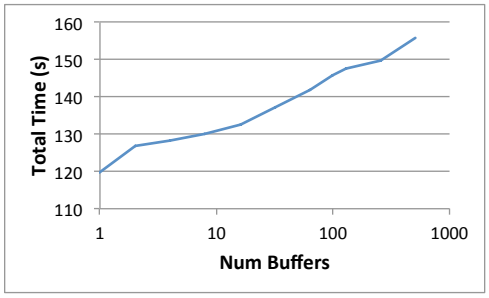
6.3 Choice of parameters

As mentioned in Section 3.1 the choice of $R = 100$ for the maximum number of messages outstanding was somewhat arbitrary - chosen to be large enough that there is enough buffering to allow processes to be able to send messages at each iteration (as per the synchronous case), but without any associated synchronisation. If in the worst case we run out of buffers, computation continues regardless, until the data has arrived and the receiver has freed up buffer space. We can see from the profiling data (Figure 4) that the amount of time taken to complete the grid operations is higher for *async* and *racy* where multiple messages are allowed in-flight. This is due to poorer performance of the memory hierarchy (shown by a 25% decrease in Translation Lookaside Buffer re-use) due to the memory occupied by buffers (either explicitly, or inside MPI), which increases the range of addresses needed to be stored beyond what can fit in the TLB, hence the higher miss rate.

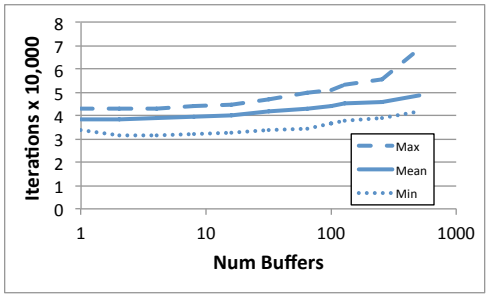
To understand the effect of this parameter on performance, we varied the number of buffers from 1 to 512 for a fixed problem size of 512 cores. The effects on total runtime, time per iteration, and the number of iter-

ations are shown in Figure 10. The first figure clearly shows that using only a single buffer is in fact the optimum choice in this case. Moreover, by varying the number of buffers it is possible to solve the system in only 120s, which is 11% faster than the synchronous implementation, compared with our default choice of 100 buffers, which is 9% slower! While increasing the number of buffers causes the number of iterations to increase slightly, most of the gain from using only a single buffer is due to the dramatic increase in iteration rate in this case. To understand the reason for this we examined profiling data from these runs and discovered that the difference in iteration rate between one and two buffers is due to extra time spent packing and unpacking the halo buffers i.e., when more buffers are available processes are able to send messages more frequently. However, we see that the number of sends per iteration does not freely increase with the number of buffers but rather plateaus at 5.12 (slightly less than the average number of neighbours - 5.25) when there are two or more buffers available - two buffers would appear to be enough that processes are able to achieve the maximum throughput of messages. We might expect this increase in message throughput to accelerate convergence, offsetting the increased cost of packing the halo data into buffers, but this turns out not to be the case, which shows global convergence of our test problem is dependent mostly on local calculation, rather than propagation of the solution by message passing. Of course, this property cannot be assumed for general linear systems, where the ratio of elements communicated at each iteration to values which can be computed based purely on local data might be much higher.

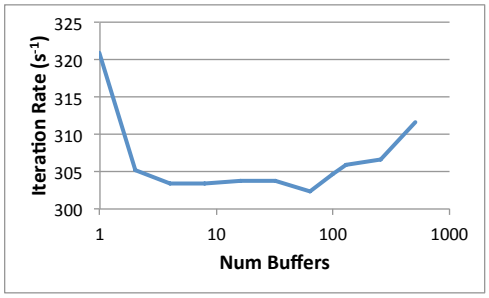
To investigate if this is in fact the case we modified the *sync* code to only



(a) Total time against number of buffers



(b) Iterations against number of buffers



(c) Iteration rate against number of buffers

Figure 10: Effects of varying the number of buffers for MPI *async* on 512 cores

carry out the halo swap once every N iterations. Figure 11 shows that as the period between halo swaps increases, the overall time to solution decreases to a minimum at $N = 32$. This shows the result that by reducing the communication burden (thus increasing the average iteration rate) we are able to progress faster towards convergence. For the $N = 32$ case, the iteration rate is 33% greater than $N = 1$, but only requires 2% extra iterations to converge. Therefore we conclude that when designing an asynchronous algorithm, rather than setting out to allow frequent communication as cheaply as possible (e.g. by buffering large numbers of messages), we should first attempt to identify the optimal rate of communication, and only then consider reducing communication overhead using an asynchronous method.

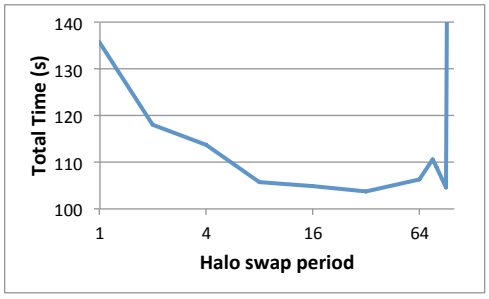


Figure 11: Iteration time against N (period of halo swap) for MPI *sync* on 512 cores

6.4 Generalisation to other iterative methods

We recognise that while the issues we have discussed show that improved scalability can be obtained by the use of asynchronous communication, our findings are specific to Jacobi's algorithm which is of little practical interest as more rapidly convergent solution schemes are usually preferred -

often one of the many Krylov Subspace methods (e.g. [19], Chapters 6 & 7). Introducing asynchrony into these methods is challenging as data from immediately previous iteration(s) is typically required in order to compute the next search vector, and dot products of distributed vectors are often needed, introducing global synchronisation at each step of the algorithm. Some work has been done on reducing the impact of communication latency in algorithms such as GMRES and Conjugate Gradients [15], but even so this does not allow such loose asynchrony as we have found can occur in practice on large-scale machines.

Instead, we propose that our methods can be applied by using the multi-splitting scheme described in [8] and [13], Section 4.1, where we would employ our asynchronous communication between iterations of the block algorithm. The inner 'block' solves should be performed by an existing synchronous parallel solution method such as an optimised implementation of Conjugate Gradients operating in parallel within the regime where it still scales well. We believe this coupling of asynchronous and synchronous solvers offers a the best opportunity for introducing asynchrony to allow problems to run at extreme scale while still allowing us to take advantage of existing highly efficient synchronous solvers. We plan to investigate the implementation and performance of our proposed scheme for a wider range of problems (including linear elasticity in a voxel finite element framework [12]) in future work.

7 Conclusion

We have implemented several variations of asynchronous parallel Jacobi using MPI, SHMEM and OpenMP, evaluated the performance of each implementation and compared the ease with which asynchronous algorithms can be implemented in each programming model. The performance of these algorithms depended on two key factors - the efficiency and scalability of the implementation, and the effect of asynchrony on the number of iterations taken to converge - both of which vary with the number of cores used. Nevertheless, we have shown that (except on 32768 cores) SHMEM can provide a more efficient implementation of asynchronous message-passing than MPI, and that for problems using on the order of thousands of cores, asynchronous algorithms can outperform their synchronous counterparts by around 10%, although in other cases may be slower. OpenMP was found to give good performance for asynchronous algorithms, and was also very easy to program compared to either MPI or SHMEM. Although it has limited scalability due to the number of cores in a shared memory node, we suggest that OpenMP might be applicable in a hybrid model with MPI, for example, particularly since we found asynchronous Jacobi in OpenMP to be 33% faster than the sychronous equivalent even on a relatively modest 32 cores. Our experience implementing these algorithms using MPI and SHMEM allowed us to propose a concise API for asynchronous 'channel' communication - a higher level of functionality than is implemented in current communication libraries.

Our investigations also uncovered some key issues of practical importance when implementing asynchronous algorithms, such as the link between

frequency of message passing and progress to convergence. If optimal parameters were chosen there is hope that we could improve the performance of our asynchronous methods significantly, although we recognise that this is difficult to achieve *a priori*. In addition, we showed that asynchronous algorithms are tolerant to performance defects, which would be an advantage on a large, noisy machine.

While our investigation concerned the use of Jacobi's method to solve Laplace's equation for a particular set of boundary conditions we have argued that our implementations could be generalised to solve arbitrary linear systems and to use other iterative solution methods by using multi-splitting. Extending our experiments to different HPC systems would also be of interest, since many of the performance characteristics we discovered are strongly architecture-dependent. Finally, the convergence behaviour of our implementations is still only understood from an implementation standpoint, and in the future we plan to combine these results with appropriate theoretical analysis which will give us the tools to make predictive, rather than empirical, statements about convergence.

8 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 are supported by EPSRC grant EP/I006702/1 "Novel Asynchronous Algorithms and Software for Large Sparse Systems".

References

[1] OpenSHMEM Application Programming Interface Version 1.0, January 2012.

[2] J. Bahi, S. Contassot-Vivier, and R. Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *17th IEEE and ACM int. conf. on International Parallel and Distributed Processing Symposium, IPDPS 2003*, pages 40a, 9 pages, Nice, France, Apr. 2003. IEEE computer society press.

[3] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing aiac algorithms. *J. Supercomput.*, 35(3):227–244, Mar. 2006.

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

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

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

[7] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[8] J.-C. Charr, R. Couturier, and D. Laiymani. Adaptation and evaluation of the multisplitting-newton and waveform relaxation methods over distributed volatile environments. *International Journal of Parallel Programming*, 40:164–183, 2012.

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

[10] Cray Inc. `intro_shmem` man pages, September 19th 2011.

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

[12] M. J. Fagan, N. Curtis, C. Dobson, J. H. Karunanayake, K. Kupczik, M. Moazen, L. Page, R. Phillips, and P. O'Higgins. Voxel-based finite element analysis: working directly with microCT scan data. *Journal of Morphology*, 268:1071, 2007.

[13] A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123:201–216, 2000.

[14] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lums-

daine. A case for standard non-blocking collective operations. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[15] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, Berkeley, CA, USA, 2010. AAI3413388.

[16] C. Maynard. Comparing One-Sided Communication With MPI, UPC and SHMEM. In *Proceedings of the Cray User Group (CUG) 2012*, 2012.

[17] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009.

[18] OpenMP Architecture Review Board. OpenMP Application Program Interface, July 2011.

[19] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003.

[20] H. Yu, I.-H. Chung, and J. Moreira. Topology mapping for blue gene/l supercomputer. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.