

# Point-to-Point Synchronisation on Shared Memory Architectures

J. Mark Bull and Carwyn Ball  
EPCC, The King's Buildings, The University of Edinburgh,  
Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.  
email: m.bull@epcc.ed.ac.uk

## Abstract

The extensive use of barriers in OpenMP often causes synchronisation of threads which do not share data, and therefore have no data dependences to preserve. In order to make synchronisation more closely resemble the dependence structure of the data used by a thread, point-to-point synchronisation can be used. We report the results of implementing such a point-to-point synchronisation routine on a Sun Fire 15K with 32 processors and an IBM p690 with 8 processors. We show that for some common dependency patterns, it is significantly more efficient than the standard OpenMP barrier.

## 1 Introduction

Use of barriers is common in shared memory parallel programming. The OpenMP API [2], [3], which facilitates the transformation of sequential codes for shared memory multiprocessors, not only contains explicitly called barriers, but many of the directives also contain hidden or *implicit* barriers.

To avoid race conditions, the order of memory operations to individual shared data must be guaranteed. To guarantee the order of operations on a single shared datum, the threads which operate on the datum are the only ones which require consideration. There is no need to synchronise threads which do not operate on the same data. Often, much of the synchronisation performed by a barrier is unnecessary.

A *point-to-point* synchronisation routine can be used to take advantage of limited thread interdependence. A point-to-point routine forces each thread to synchronise with an individual list of threads. For example, in the case where the value of a datum depends on the values of neighbouring data in an array, only the threads which update neighbouring data need to be considered. The thread need not synchronise with other threads which have no influence on its data.

It is hoped that point-to-point routine will have two performance benefits over barriers. First, the time taken to communicate with a small number of threads is much less than the time taken for all threads to communicate with each other, no matter how intelligent the communication algorithm. Second, looser synchronisation can reduce the impact of non-systematic load imbalance between the threads.

Excessive barrier synchronisation is often the reason why regular domain decomposition OpenMP codes do not scale as well as MPI versions of the same code on very large shared memory systems. Another situation where barrier overheads can dominate is in legacy vector codes which contain large numbers of fine-grained parallel loops. Such codes frequently perform poorly on superscalar SMP systems.

A set of experiments has been performed to investigate the potential benefits of point-to-point synchronisation. We also consider how point-to-point synchronisation might usefully be incorporated into the OpenMP API.

## 2 Point-to-Point Routine

The point-to-point routine, running on  $N$  threads uses a shared array, all members of which are initialised to a constant value (zero). Each thread uses a different element of the array. Each thread passes to the routine a list of threads on which it has a dependency. This process is not automatic, but the programmer may choose from a number of pre-designed patterns (see the next section) or create his own. It should be noted, however, that where dynamic loop scheduling is used, it is not possible to determine which thread accesses which data. The list of thread

numbers—known as the *dependency list*—is stored in a private array. Each thread waits until all its dependent threads have entered the routine the same number of times as itself.

```

void sync(int me, int *neighbour, int numneighbours) {
    int i;

    /*   Increase my counter:   */

    counter[me*PADDING]++;
    #pragma omp flush (counter)

    /*   Wait until each neighbour has arrived:   */

    for(i=0;i<numneighbours;i++) {
        while(counter[me*PADDING] > counter[neighbour[i]*PADDING]){
            #pragma omp flush (counter)
        }
    }
}

```

Figure 1: Point-to-Point synchronisation code.

When a thread arrives at the routine, it increments its member of the shared array and busy-waits until the members of the array belonging to each of the threads on its dependency list has been incremented also. In doing so it is ensuring that each of the relevant threads has at least arrived at the same episode. The relevant code fragment is shown in Figure 1.

In this routine, `counter` is an array of 64-bit-integers (`long long`), to reduce the possibility of overflow. It is unlikely that a number of synchronisations larger than the maximum capacity of a 64-bit `long long` ( $2^{63} - 1 \approx 10^{19}$ ) will be required in any application. If this is necessary, however, the initialisation routine can be run again to reset the array to zero. `PADDING` is the *array padding factor*, which prevents false sharing by forcing each `counter` to be stored on a different cache-line. On a Sun Fire 15K, for example, cache-lines are 64 bytes (8 64-bit words) long, so the optimal value for `PADDING` is 8.

Note the use of the OpenMP `flush` directive to ensure that the values in the `counter` array are written to and read from the memory system, and not stored in registers. We have also implemented system specific versions for both the IBM and Sun systems which do not use the `flush` directive. On the Sun system, the above code functions correctly without the `flush` directives provided it is compiled without optimisation and `counter` is declared `volatile`. On the IBM, in addition to the `volatile` declaration, synchronisation instructions (see [1]) have to be added, as the Power4 processor has a weaker consistency model and allows out-of-order execution. A `lwsync` instruction is needed in place of the first `flush`, and an `isync` instruction after the while loop.

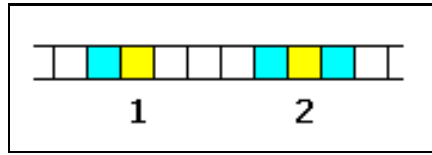
### 3 Common Dependence Patterns

The dependency list of a thread, contains the IDs of threads which it synchronises with. With the synchronisation routine shown in Figure 1, it is left to the programmer to build this list.

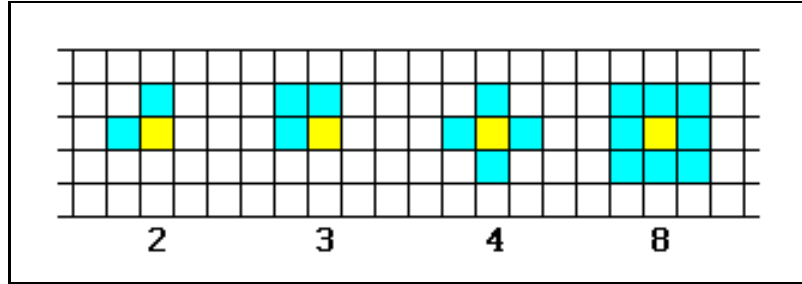
There are several dependence patterns which occur frequently in data-parallel style codes. A selection is illustrated in Figure 2. The threads are arranged into one, two and three dimensional grids, in order to operate on correspondingly ordered data arrays. Cases where dependence flow is restricted to a number of dimensions lower than the dimensionality of the grid (for example, where each new value depends on some function of the neighbouring values in one dimension but not the other:  $H_{i,j}^{new} \leftarrow H_{i\pm 1,j}^{old}$ ) are referred to as *dimensionally degenerate* cases and are not considered.

One dimensional arrays have two common patterns: the data can depend on one or both neighbouring data points:

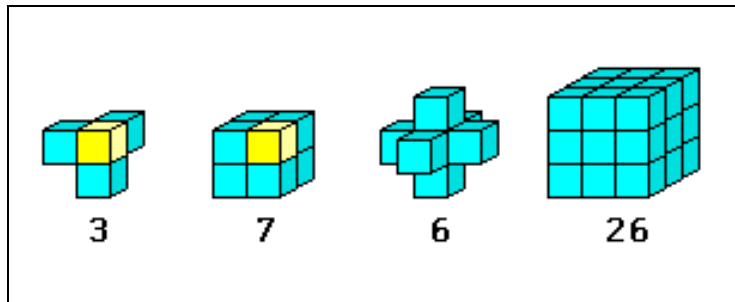
$$H_i^{new} \leftarrow H_{i\pm 1}^{old} \quad \text{or} \quad H_i^{new} \leftarrow H_{i+1}^{old}, H_{i-1}^{old}$$



(a) 1 dimension



(b) 2 dimensions



(c) 3 dimensions

Figure 2: Common data flow patterns. The numbers indicate the number of threads in the dependency list.

Four common non-degenerate two-dimensional patterns are shown in Figure 2(b). The leftmost illustrates the case where a data point depends on two adjacent neighbours:

$$H_{i,j}^{new} \Leftarrow H_{i-1,j}^{old}, H_{i,j-1}^{old}$$

Next, the so called *wave-fronted loop* is the case where a data point depends on the value of one of the diagonally adjacent points, for example:

$$H_{i,j} \Leftarrow H_{i-1,j-1}$$

It is called “wave-fronted” because after the top-right corner point has executed, the threads can execute in a diagonal wave through the grid.

Next is the *5-pointed stencil*, which is where a data point is dependent on its four nearest adjacent neighbours in some way. This is shown in the central diagram in Figure 2(b). The rightmost diagram shows a variation called the *9-pointed stencil*, which can be used to obtain more accuracy. For example, where the 5-pointed stencil would correspond to a diffusion term such as:

$$H_{i,j}^{new} \Leftarrow \frac{1}{4}(H_{i-1,j}^{old} + H_{i+1,j}^{old} + H_{i,j-1}^{old} + H_{i,j+1}^{old})$$

a more accurate result can be gained by instead using the term:

$$H_{i,j}^{new} \leftarrow \frac{1}{5}(H_{i-1,j}^{old} + H_{i+1,j}^{old} + H_{i,j-1}^{old} + H_{i,j+1}^{old}) + \frac{1}{20}(H_{i-1,j-1}^{old} + H_{i+1,j-1}^{old} + H_{i-1,j+1}^{old} + H_{i+1,j+1}^{old})$$

which requires a 9-pointed stencil pattern. The three-dimensional non-degenerate patterns shown in Figure 2(c) are natural extensions into three dimensions of the two dimensional patterns.

The patterns listed above are not the only non-degenerate patterns. Involving all nearest neighbours in  $m$  dimensions, there are  $(3^m - 1)! - 3 \times (3^{m-1} - 1)!$  potential non-degenerate patterns including all laterally and rotationally symmetric repetitions. However most of these represent dependence patterns which occur rarely in practice.

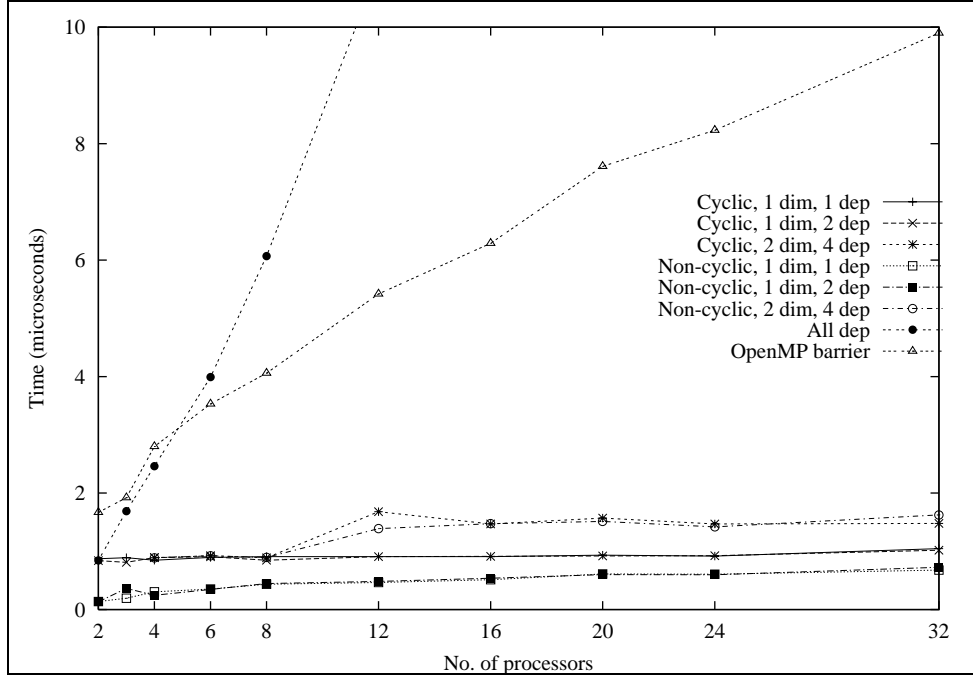


Figure 3: Overhead for synchronisation routines on Sun Fire 15K

## 4 Results

In this section we report the results of experiments which assess the performance of the point-to-point routine. We measured the overhead of the routine by measuring the time taken to execute 1,000,000 calls, and computing the mean overhead per call. We ran each experiment five times and took the best result in each case.

We used two systems to test the routine: a Sun Fire 15K system where we had access to 32 out of a total of 52 processors, and a 32 processor IBM p690 where we had access to an eight processor logical partition. The Sun Fire 15K system contains 900 MHz UltraSparc-III processors, each with a 64 Kbyte level 1 cache and an 8 Mbyte level 2 cache. The system has 1 Gbyte of memory per processor. We used the Forte Developer 7 C compiler.

The IBM p690 system contains 1.3 GHz Power4 processors, each with a 32 Kbyte level 1 cache. Each pair of processors shares a 1.44 Mbyte level 2 cache and all eight processors in the logical partition share 128 Mbytes of level 3 cache. We used the XLC 6.0 C compiler.

Figure 3 shows the overhead of the point-to-point routine on the Sun Fire 15K for a number of synchronisation patterns: 1 and 2 neighbours in one dimension and 4 neighbours in 2 dimensions, with cyclic and non-cyclic boundary conditions in each case. We also show the overhead of the OpenMP barrier directive, and the result of implementing a barrier (i.e. synchronising all threads) using the point-to-point routine. We see immediately

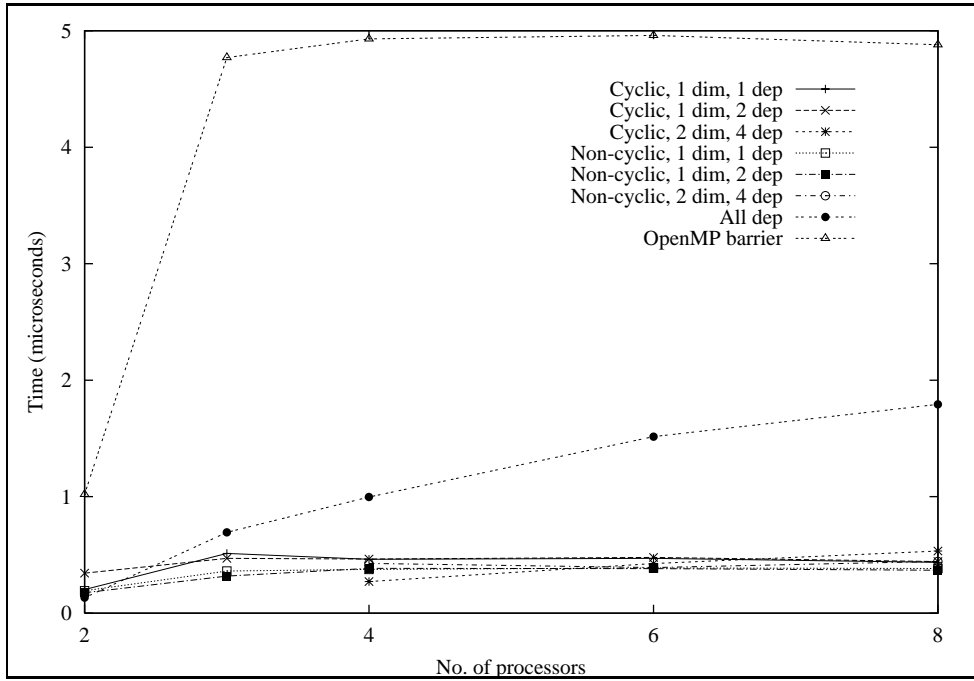


Figure 4: Overhead for synchronisation routines on IBM p690

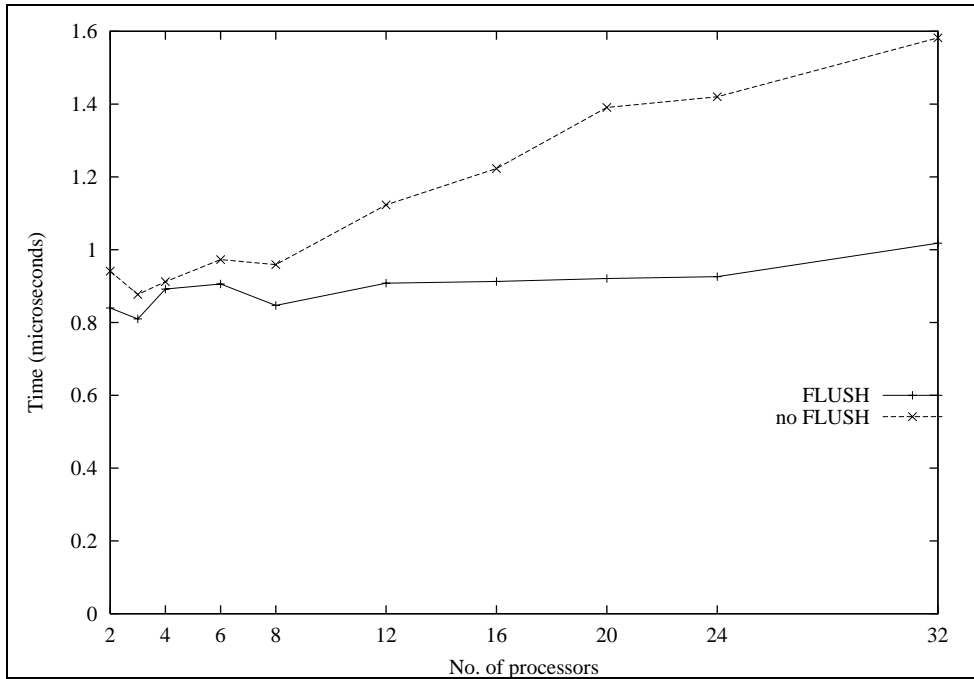


Figure 5: Comparison of point-to-point implementations on Sun Fire 15K

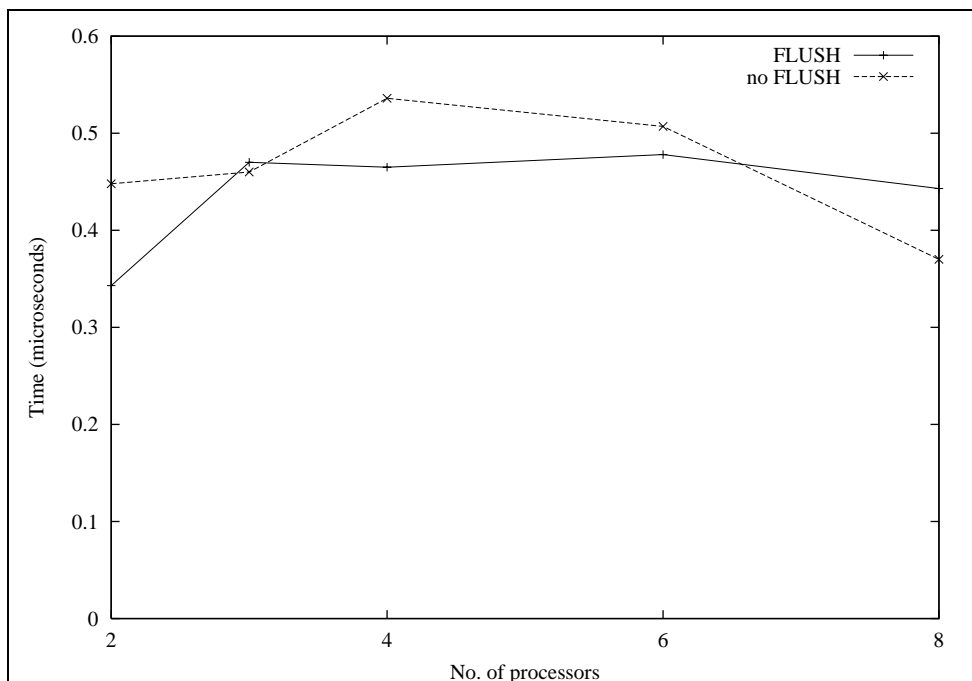


Figure 6: Comparison of point-to-point implementations on IBM p690

that the point-to-point routine for small numbers of dependencies has significantly less overhead than an OpenMP barrier. On 32 processors, there is roughly an order of magnitude difference. The overhead increases with the number of dependencies, and is lower for non-cyclic boundary conditions than for cyclic ones. Synchronising all threads with the point-to-point routine is, predictably, very expensive for more than a small number of threads.

Figure 4 shows the results of the same experiments run on the IBM p690. On three or more processors there is about an order of magnitude difference between the point-to-point routines and the OpenMP barrier. The number of neighbours affects the overhead here less than on the Sun system: this might be an effect of the shared cache structure. Indeed, synchronising all threads with the point-to-point routine is significantly cheaper than the OpenMP barrier, suggesting some room for improvement in the implementation of the latter.

Figures 5 and 6 show the overhead of the point-to-point routine for one of the above cases (1 dimension, 2 dependencies, cyclic boundaries) on the Sun and IBM system respectively. In each case we compare the performance of the OpenMP version shown in Figure 1 with the system-specific versions (without the `flush` directive) described in Section 2. On the Sun Fire 15K the OpenMP version is significantly faster: this can be attributed to the need to compile the system-specific version with a lower optimisation level. On the IBM there is not much to choose between the versions. We can conclude that on both systems the `flush` directive is efficiently implemented and there is no reason to prefer a system-specific solution.

In order to assess the impact of using the point-to-point routine, we devised a small benchmark to simulate its use in a code with fine-grained loop parallelism. The kernel of this code is shown in Figure 7.

A standard OpenMP version of this code would require two barriers per outer iteration. For the point-to-point routine, each thread only has a dependency on its two neighbouring threads in a 1-dimensional grid, with non-cyclic boundaries.

Figure 8 shows the speedup obtained on 8 processors of the IBM system with values of  $N$  varying from 1000 to  $10^6$ . We observe that for large values of  $N$ , both versions give good speedups (note that there are some superlinear effects due to caching). For smaller values of  $N$ , however the difference in speedup becomes quite marked. For  $N = 1000$ , no speedup can be obtained by using barriers. By using the point-to-point routine a modest but useful speedup of just under 4 is obtained.

## 5 Making Point-To-Point Synchronisation Useful

The experiments revealed that for the dependence patterns studied, point-to-point synchronisation is worth using instead of barriers. For a point-to-point routine to be useful, it must be straightforward to use. In this section we discuss how a point-to-point routine might fit into the OpenMP API.

In these circumstances, we ask: how much detail could be hidden by the API, while not restricting functionality? The aim of this section is to find a good compromise between simplicity and flexibility, i.e. keeping it simple, but still useful.

As discussed before, the OpenMP API provides directives and clauses which allow the programmer to inform the compiler about how variables should be treated. For example, a piece of code which should be executed on only one thread should be labelled with a `single` directive. This normally contains an implicit barrier, but this can be suppressed with the use of the `nowait` clause.

There are two places where a point-to-point routine could be useful: as a directive and as a clause. The directive form, `pointtopoint(list, length)` could replace a `barrier` for some situations where barriers are currently used. It could be particularly useful in situations where the amount of work required for each datum at each time-step is irregular. A barrier would force all threads to wait for the last one at each time-step. A point to point routine would allow more flexibility, and reduce the overhead of the load imbalance.

To make the routine easier to use, it would be advantageous to provide a number of functions which generate dependency lists for all the commonly used patterns. A programmer should be able to generate his own pattern for cases where his pattern is not provided for.

A `waitforthreads(list, length)` clause could appear wherever a `nowait` clause is permitted. It would be used to replace the implicit barrier which comes at the end of a construct. Each thread waits for all the relevant neighbours to be in place before proceeding. This requires a list which depends on the situation. We must allow for the possibility that two different sections of code may require two different lists. This would be most easily done by requiring that the programmer pass in a list.

Another useful place for a point-to-point algorithm is after a `single` section. All the non-executing threads only need to wait for the thread which executes the code. As the executing thread is the first thread to arrive at that section, the other threads' lists would need to contain its ID. The communication required of the first thread to pass its own ID would not be significantly more than that required to inform all the others that it is the executing thread. The clause `waitforsingle` is not necessary: it is more efficient to use this than an implicit barrier, then it should be used, but the programmer doesn't need to know about it.

We have noted that system-specific implementations of point-to-point synchronisation appear unnecessary. As a result, the portable synchronisation routine of Figure 1 may be used, and there may not be much merit in extending the OpenMP API for this purpose.

Finally we note that a compiler with sufficiently good dependence analysis could automatically replace implicit or explicit barriers with the required point-to-point synchronisation.

```
for (iter=0; iter<NITERS; iter++) {  
  
    for (i=1; i<=N; i++)  
        b[i] = 0.5*(a[i-1]+a[i+1]);  
  
    for (i=1; i<=N; i++)  
        a[i] = 0.5*(b[i-1]+b[i+1]);  
  
}
```

Figure 7: Fine grained loop kernel

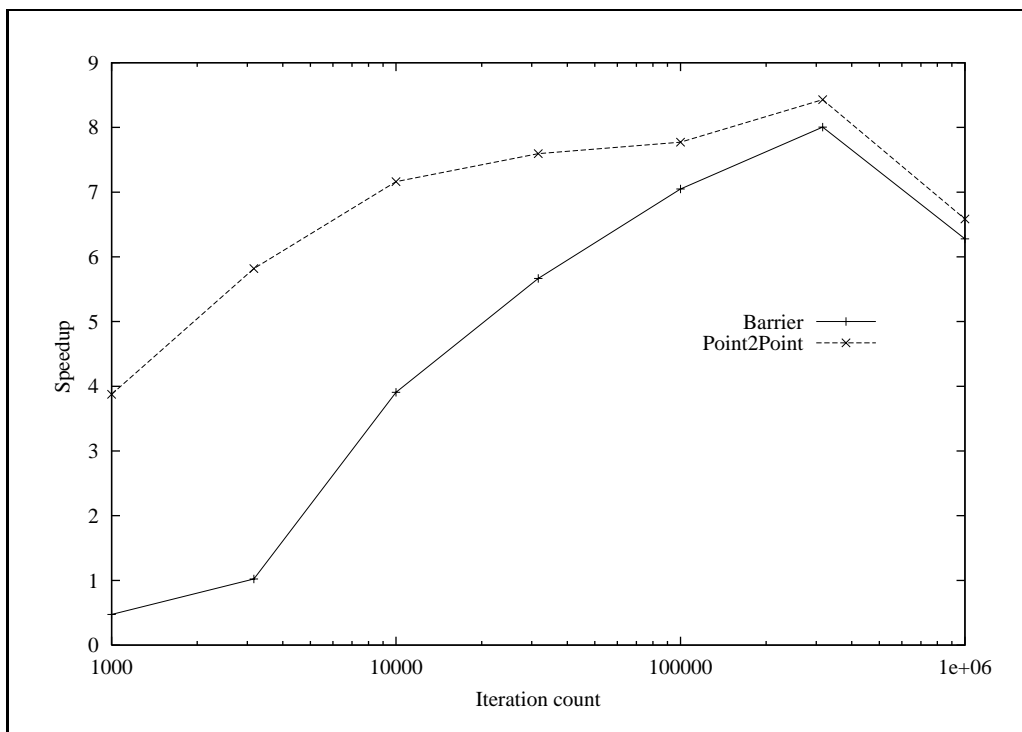


Figure 8: Speedup of fine grained loop kernel on eight processors of the IBM p690

## 6 Conclusion

Barriers can be a significant overhead in OpenMP codes, but they may enforce more synchronisation between threads than is required by the data dependencies. A point-to-point synchronisation routine allows a programmer to mould the synchronisation to fit the dependency structure. To measure the benefit, a point-to-point routine was made using C and OpenMP and timed for 1,000,000 iterations for some common data dependence patterns.

The point-to-point routine was found to be significantly faster than the OpenMP barrier for the dependence patterns examined, but there was no advantage from system-specific implementations which avoid the use of the `flush` directive.

We have considered how such a routine might be included in the OpenMP API, but the benefits of doing so are unclear.

It would be interesting to test our point-to-point routine in a selection of real-world applications to discover whether there is any advantage, as has been suggested by these results.

## References

- [1] Hay, R.W. and G.R. Hook, "POWER4 and shared memory synchronisation", April 2002, available at [http://www-1.ibm.com/servers/esdd/articles/power4\\_mem.html](http://www-1.ibm.com/servers/esdd/articles/power4_mem.html)
- [2] OpenMP ARB, "OpenMP C and C++ Application Program Interface", Version 2.0, March 2002.
- [3] OpenMP ARB, "OpenMP Fortran Application Program Interface", Version 2.0, November 2000.