

Point-to-Point Synchronisation on Shared Memory Architectures

Carwyn Ball and Mark Bull

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 contained by a thread, a point-to-point synchronisation routine can be used. Such a routine forces a thread only to synchronise with a list of relevant threads. If a barrier involves a lot of unnecessary synchronisation, the overhead of this may be reduced using a point-to-point routine. Such a routine has been created using C and OpenMP, and tested with a number of common dependence structures with cyclic and non-cyclic boundaries on a Sun Fire 6800 with 24 processors. It was found that for any level of dependence involving only nearest neighbours (adjacent and diagonal) in a virtual grid of dimensionality one, two and three, point-to-point synchronisation is more efficient than the standard OpenMP barrier to synchronise threads.

1 Introduction

Race conditions can cause a program to execute in a non-deterministic fashion, producing inconsistent results. Synchronisation routines are used to remove race conditions from a code. An *episode* of a synchronisation routine forces a “fast” thread to wait for other threads to reach the same episode. Thus operations on shared data can be separated into different *epochs*, insuring the correct execution of a code.

Barrier synchronisation is a common technique. A thread executing an episode of a barrier waits for all other threads before proceeding to the next epoch. Therefore, when a barrier is reached, all threads are forced to wait for the last thread to arrive.

Use of barriers is common in shared memory parallel programming. The OpenMP library [1], [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 the 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 the data which represent the area surrounding a point in space, it is the threads which update the data on which a thread’s data depends which are contained on the thread’s *dependency list*. The thread need not synchronise with other threads which have no influence on its data.

It is hoped that point-to-point 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 load imbalance between the threads.

A sets of experiments have been performed to investigate the potential benefits of point-to-point synchronisation. These time 1000000 basic executions of the routine with a set of common dependence patterns, with both periodic and non-periodic boundary conditions.

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 algorithm, running on N threads requires a shared array of length N , all members of which are initialised to a constant value (zero). The m^{th} member of this array is “owned” by thread m . Each thread finds the threads on whose data its data depend on. 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. The list of thread numbers—known as the *dependency list*—is stored in a private array.

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

    /* Increase my counter:
     */
    counter[me*PADDING]++;

    /* Wait until each neighbour has arrived:
     */
    for(i=0;i<numneighbours;i++) {
        while(counter[me*PADDING] > counter[neighbour[i]*PADDING]);
    }
}
```

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 6800, cache-lines are 64 bytes (8 64-bit words) long, so the optimal value for `PADDING` is 8.

3 Experiments

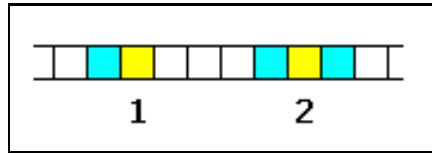
3.1 Common Dependence Patterns

The dependency list of a thread, which is stored in its private memory, 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. A thread’s dependency list should contain the IDs of threads which write to data which it reads.

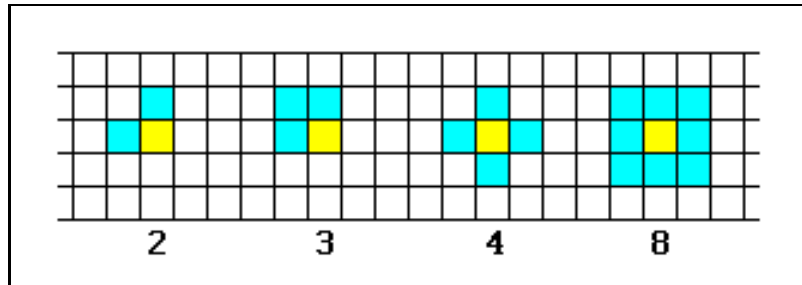
There are several dependence patterns which occur frequently in data-parallel style codes. A selection are illustrated in Figure 2. The threads are arranged into one, two and three dimensional grids, in order to operate on correspondingly ordered data arrays. Cases where dependence flow is restricted 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: $H_{i,j}^{\text{new}} \leftarrow H_{i\pm 1,j}^{\text{old}}$) 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:

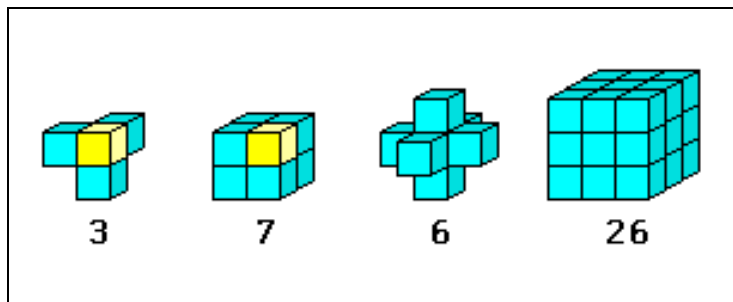
$$H_i^{\text{new}} \leftarrow H_{i\pm 1}^{\text{old}} \quad \text{or} \quad H_i^{\text{new}} \leftarrow H_{i+1}^{\text{old}}, H_{i-1}^{\text{old}}$$



(a) 1 dimension



(b) 2 dimensions



(c) 3 dimensions

Figure 2: Common data flow patterns. Squares or cubes represent threads. The blue threads appear on the dependency list of the yellow thread. The numbers beneath the indicate the number of “blue” threads for the “yellow” thread.

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:

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

Number	2D decomp.	3D decomp.
4	{2, 2}	
6	{2, 3}	
8	{2, 4}	{2, 2, 2}
9	{2, 5}	
10	{2, 5}	
12	{3, 4}	{2, 2, 3}
14	{2, 7}	
15	{3, 5}	
16	{4, 4}	{2, 2, 4}
18	{3, 6}	{2, 3, 3}
20	{4, 5}	{2, 2, 5}
21	{3, 7}	
22	{2, 11}	
24	{4, 6}	{2, 3, 4}

Table 1: Decompositions of non-prime numbers up to 24, for the creation of two and three dimensional grids.

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.

3.2 Benchmarks

The patterns shown in Figure 2 are the patterns which were tested with the routine shown in Figure 1. At the edges of the grid, there is a choice between having cyclic boundaries or not. For each pattern both cases were tested.

For two dimensions and higher, thread numbers which factorised into an appropriate number of dimensions were selected. In cases where there is a choice of factorisation, the “squarest” or “most cubic” selection was chosen—the factorisation for which the factors are as similar as possible. To find this, the number is divided into its prime factors. The smallest two factors are multiplied and the factors are reordered until the list is of appropriate length. For example, the number 24 for two dimensions is factorised into $\{2,2,2,3\}$, which is reduced to $\{4,2,3\}$, which is reordered to $\{2,3,4\}$, which is reduced to $\{6,4\}$. $\{6,4\}$ is a “squarer” factorisation than $\{3,8\}$ or $\{2,12\}$. Table 1 shows the decompositions of the non-prime numbers up to 24.

In order to calculate which threads neighbour each other, each thread calculates its coordinates in the grid, using routine shown in Figure 3(a).

The appropriate neighbour in a particular dimension is the next thread along in that direction. When dimensionality is greater than one, then diagonal neighbours are found by moving along by one in more than one dimension. All the diagonal neighbours which reside on the hyperplane defined by \mathcal{D} —the set of dimensions $\{d_j \mid j \in \mathcal{Z}^+\}$ —are found with the following formula:

$$a_i \longrightarrow \begin{cases} a_i & \text{if } d_i \notin \mathcal{D}, \\ a_i \pm 1 & \text{if } d_i \in \mathcal{D}. \end{cases}$$

For example, if $\mathcal{D} \equiv \{d_2\}$, then:

$$a_i \longrightarrow \begin{cases} a_i & \text{if } i \neq 2, \\ a_i \pm 1 & \text{if } i = 2. \end{cases}$$

```

void id_to_coords(int *factor, int *coord,
                 int id, int ndimen-
                 sions){
    int i,temp=1;

    for (i=0; i<ndimensions; i++) {
        coord[i] = (id/temp) % factor[i];
        temp *= factor[i];
    }
}

```

(a) id \rightarrow coordinates

```

int coords_to_id(int *coord,int *factor,
                int ndimen-
                sions){
    int temp=1,rtn=0,j;

    for(j=0; j<ndimensions; j++) {
        rtn += coord[j]*temp;
        temp *= factor[j];
    }

    return rtn;
}

```

(b) coordinates \rightarrow id

Figure 3: Routines for exchanging a thread’s grid coordinates, and its identification number. The array `factor[]` contains the dimensions of the grid (factors of the number of threads). `id` is the thread’s identification number.

This formula produces the adjacent neighbours along dimension 2. To find the neighbours of a thread, one uses this *nearest neighbour formula* for each D of interest.

For example, all the neighbours of a node in a three dimensional grid are: six nearest neighbours (up, down, back, front, left and right); twelve neighbours which are found on two dimensional planes (the up-down-left-right plane, the up-down-front-back plane and the left-right-front-back plane, each of which contain four neighbours); and the eight corners of the cube, totalling 26. Each of the new sets of coordinates is converted back to a thread number using the function shown in Figure 3(b) and this thread number is added to the dependency list.

4 Results

The results from the simple benchmarks are illustrated in Figures 4–9. For comparison, times obtained from the same test-code using instead the OpenMP barrier are shown in Figure 10.

For most cases, the time is either approximately logarithmic with the number of threads, or approximately constant. A number of the cases with higher dimensionality have a large drop in performance on 24 threads. This may be caused by full-machine-contention—background processes competing for CPU—which such highly synchronised codes are more sensitive to. Furthermore, there are two examples of more unusual behaviour:

1. If we assign each 2d virtual grid to a set, depending on the smaller of the dimensions (the *2-set* contains the

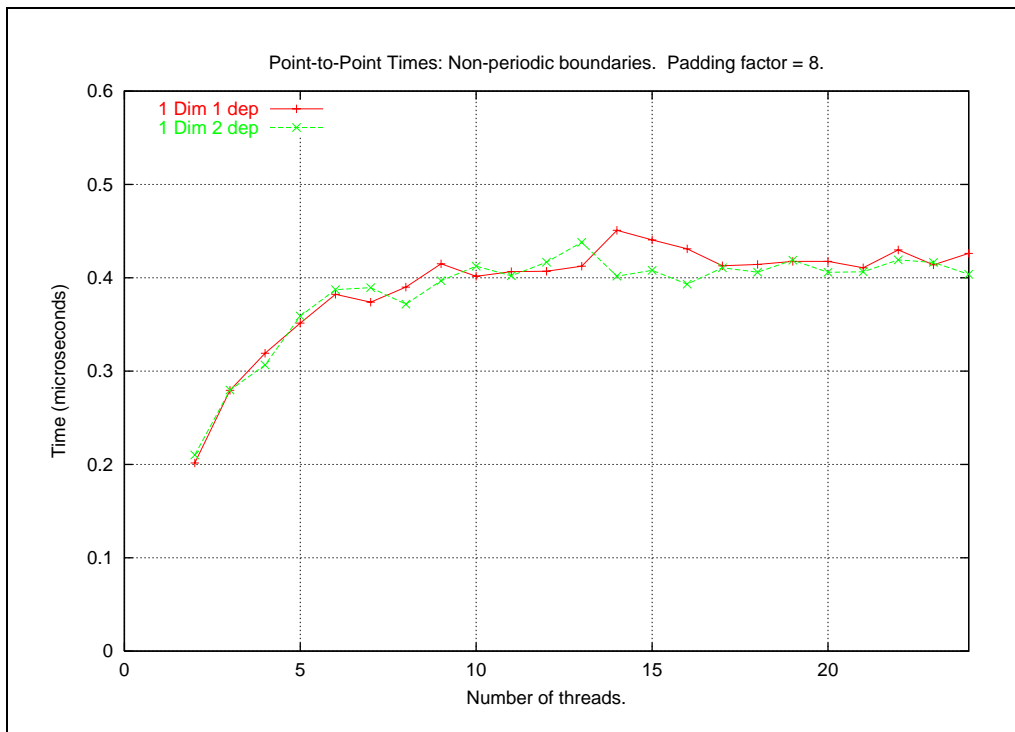


Figure 4:

grids: 2×2 , 2×3 , 2×4 , 2×5 , 2×7 , 2×11 , which correspond to 4, 6, 8, 10, 14, 22 thread parallelisations; similarly, the 3-set contains 9, 12, 15, 18, 21 and the 4-set contains 16, 20 and 24), the time taken by the code is approximately constant for each set. Examples of this behaviour are the the 2Dim 4dep cases and the 2Dim 3dep periodic boundaries case. The most pronounced example is the 2D-4d-periodic case, for which the 2-set times are grouped around 0.7, the 3-set times are grouped around 1.2 and the 4-set times are grouped around 1.1. (The 24 threads (4×6) case breaks this pattern with a large drop in performance.) For each case, the 2-set is most efficient. The 3-set and 4-set are similar valued and neither is consistently more efficient.

Perhaps this behaviour can be explained in terms of degeneracy. For a $2 \times X$ grid with periodic boundaries, the opposite neighbours in one dimension will be the same thread. For non-periodic cases, the `neighbours` list contains a thread's own id if it is located at the edge of the grid. The processor can in both cases avoid a cache miss.

2. The behaviour of the times obtained from the 2D-8d non-periodic boundary case. Timings appear to be vaguely grouped around 1 microsecond, which shows significantly worse performance than cases with fewer dependencies. For thread numbers 4, 10, 21, execution is significantly more efficient than similar values.

The significance of these numbers is not clear: it may be the case that this effect is just experimental scatter. The high interdependence of the threads is preventing threads from settling into exaggeratedly-efficient runtime-synchronisation-patterns.

If these diagnoses are correct, then it is expected that both forms of unusual behaviour will be absent from real codes, where synchronisation is not the only overhead.

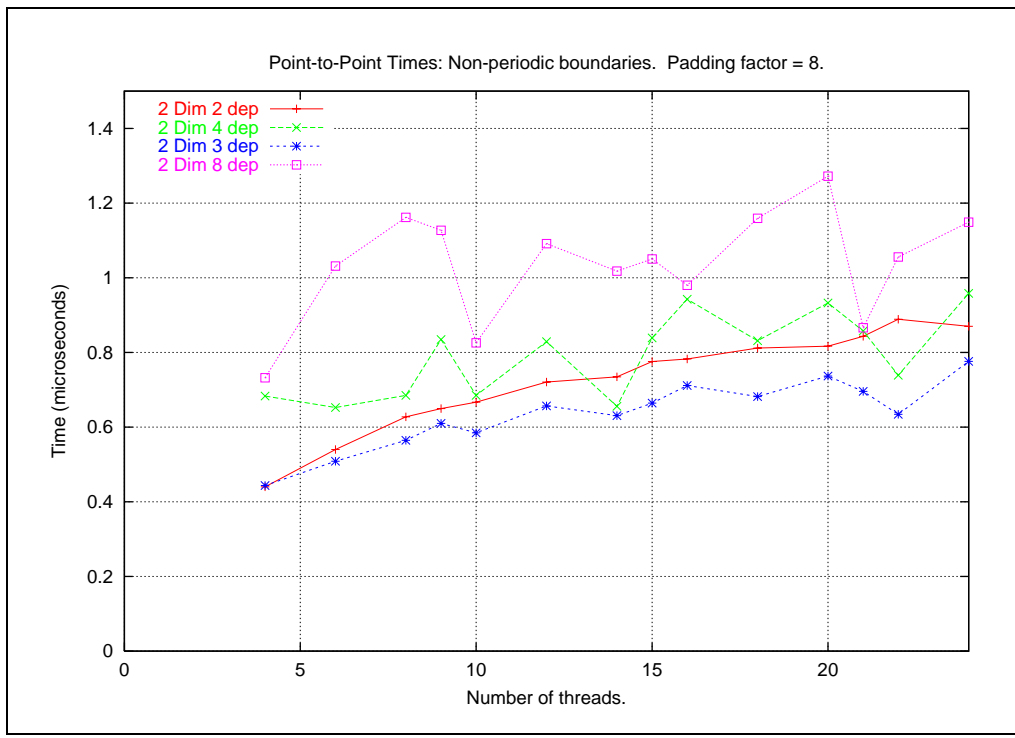


Figure 5:

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 library, 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 library 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 a 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 clause which may after a directive may be `waitforthreads(list, length)`. It would be used to replace the implicit barrier which comes after a directive. After a directive, the 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. It could be used with the `for` directive, which is used to easily parallelise loops.

Another useful place for a point-to-point algorithm is after a `single` section. All the non-executing threads

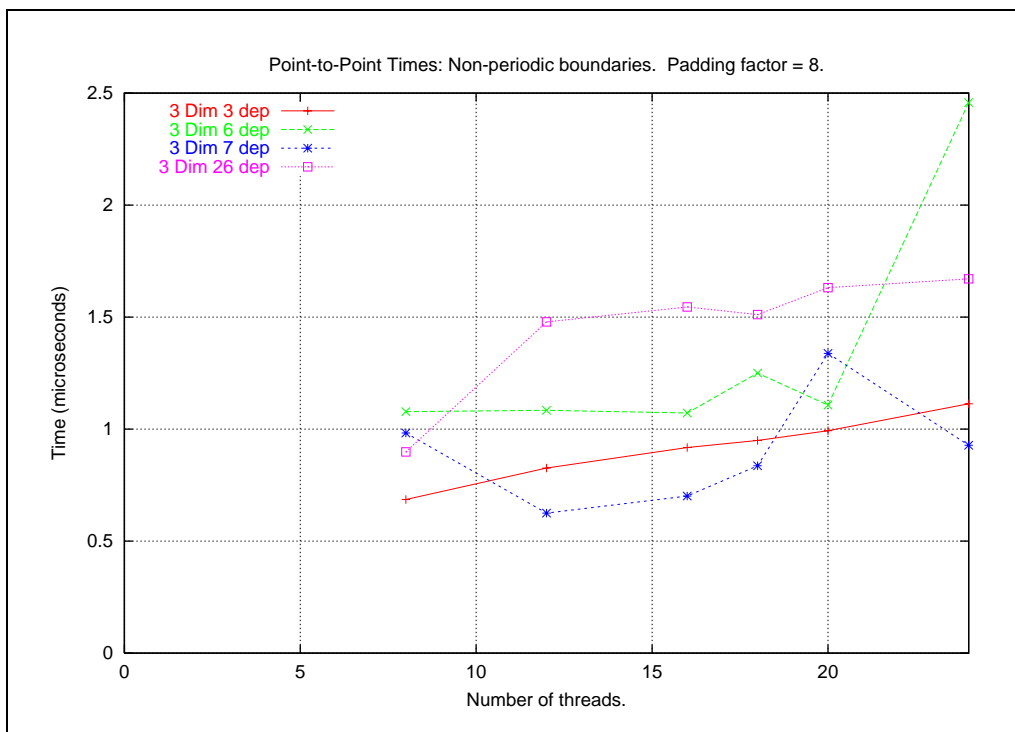


Figure 6:

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` will not be necessary. If 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.

6 Conclusion

Barrier synchronisation require a large amount of synchronisation which is unnecessary for many applications. Largely, a thread is only required to synchronise with the neighbours containing data on which its data depend. 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 a number of common data dependence patterns.

The point-to-point routine was found to be competitive compared to the OpenMP barrier algorithm for all of the dependence patterns examined.

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. It may transpire that point-to-point routines are more beneficial, not because of their overhead, but because of their function—allowing threads to restart earlier.

References

- [1] "The OpenMP Home Page", <http://www.openmp.org/>
- [2] "OpenMP C and C++ Application Program Interface", Version 1.0, October 1998.
- [3] "OpenMP Fortran Application Program Interface", Version 2.0, November 2000.

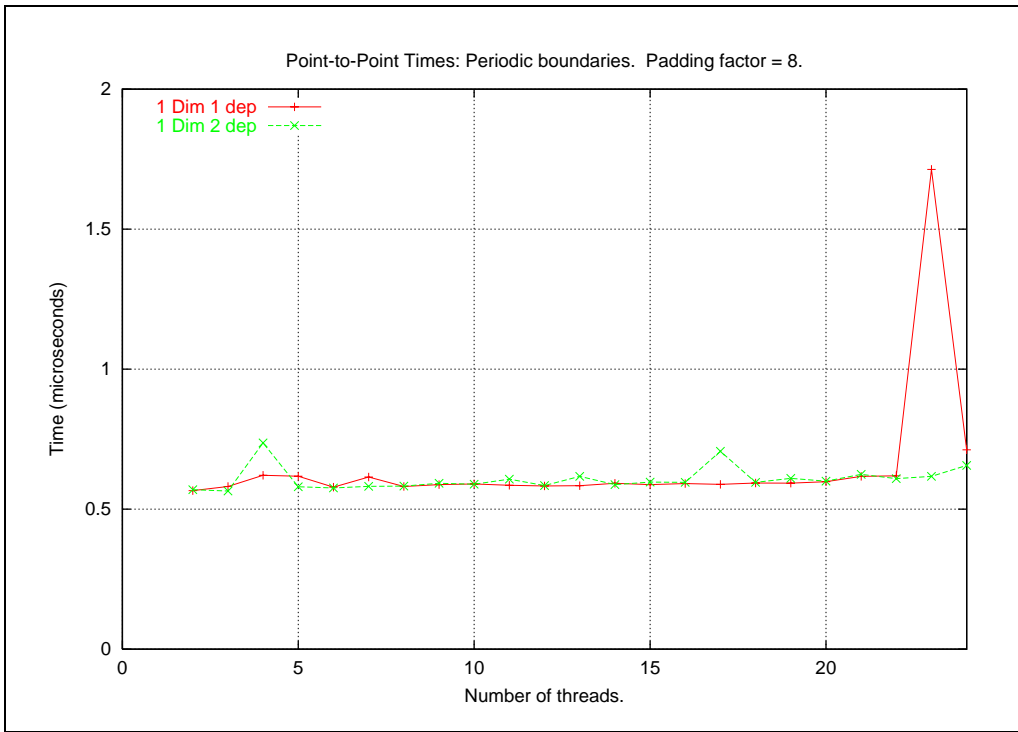


Figure 7:

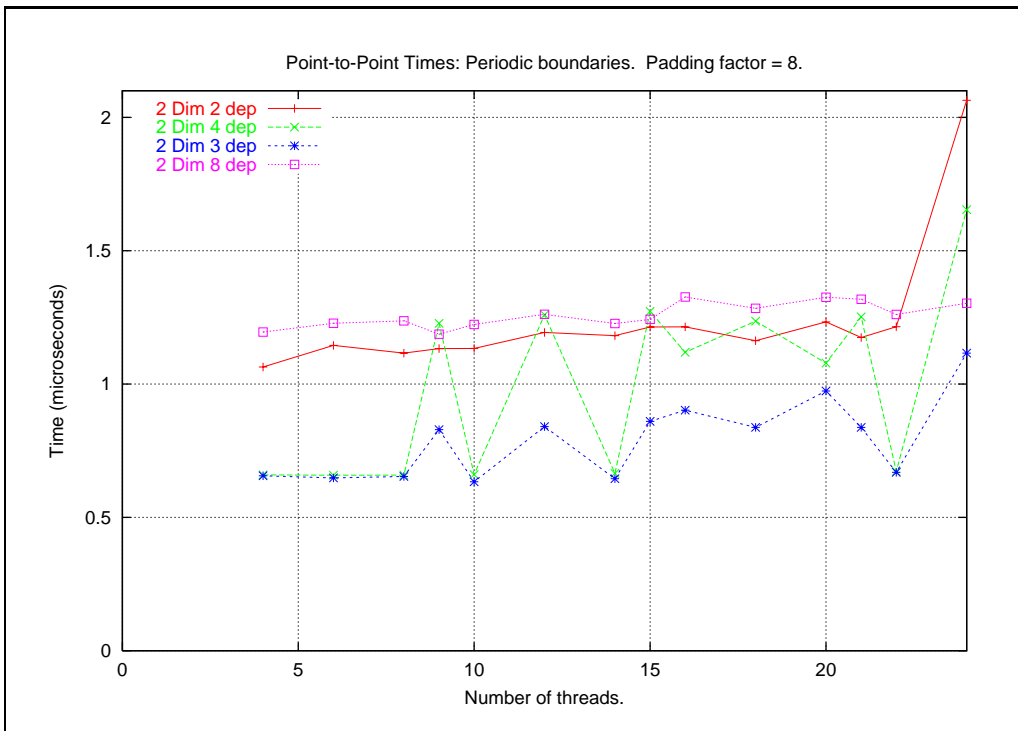


Figure 8:

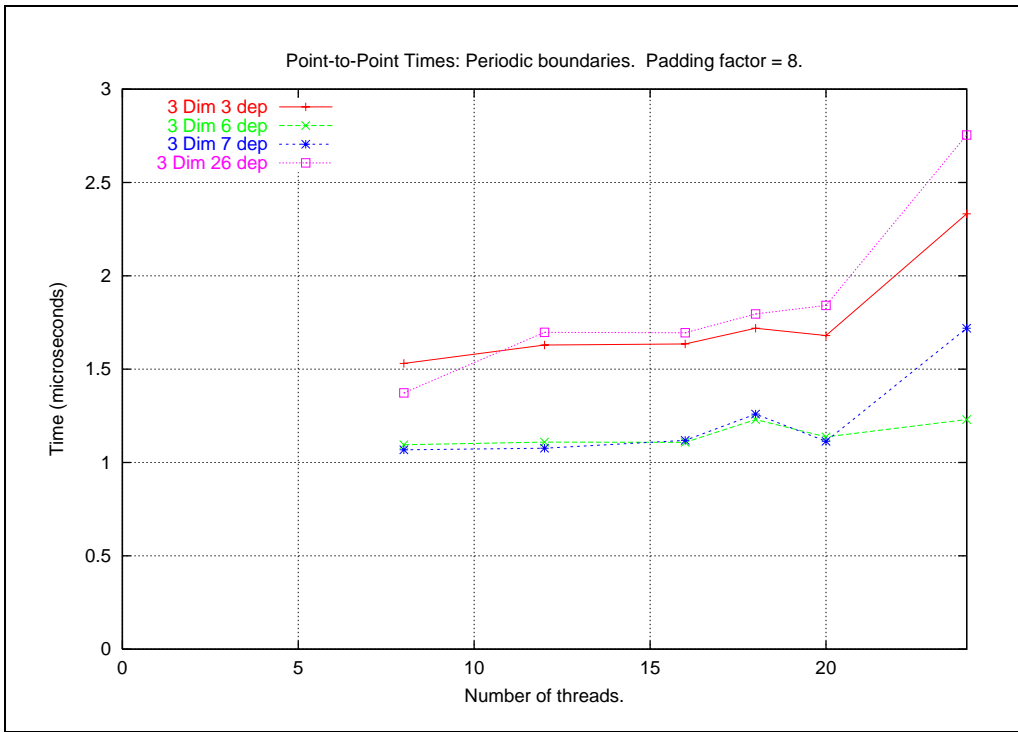


Figure 9:

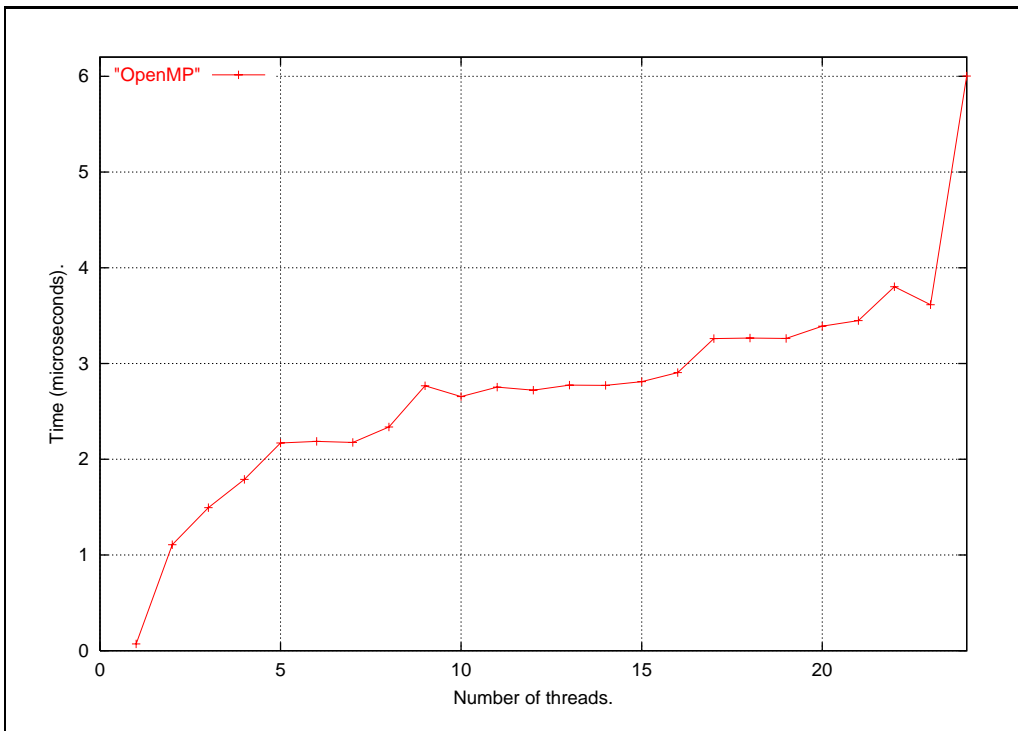


Figure 10: