Optimizing Performance for SCore Cluster Enabled OpenMP Applications: Solving the Laplace Equation and Performing Fourier Transforms

H'sien J. Wong* and Alistair P. Rendell
Department of Computer Science, Australian National University
Canberra ACT 0200, Australia
jin.wong@anu.edu.au
alistair.rendell@anu.edu.au

Abstract

In this paper the performance of two application kernels (the 2D Laplace equation and the 1D fast Fourier transform) parallelised using the SCore cluster enabled OpenMP compiler is studied. Timing results for a variety of implementations run on a cluster with dual 550 MHz Pentium III processor nodes linked via a 100 MBit Ethernet interconnect are given. For the 2D Laplace equation the effect of page placement was found to be critical in order to obtain any speedup. Parallelisation of the 1D fast Fourier transform reveals a lack of appropriate OpenMP directives to deal with divide-and-conquer styled work sharing schemes. Possible extensions to the OpenMP standard to address this deficiency are presented.

1 Introduction

Recently there has been much interest in developing OpenMP programming environments for cluster computer systems [4, 6, 7, 8]. Most of these attempts have been based on mapping OpenMP to some existing Software Distributed Shared Memory (SDSM) environment, although we note interesting recent work by Huang *et al* to implement OpenMP over Global Arrays [5]. In either case, to obtain reasonable performance from an OpenMP code on a cluster it is likely that the application programmer will require some knowledge of the implementation.

SCore is a cluster system software that provides a parallel programming environment. Originally developed by the Real World Computing Partnership (RWCP), the project has now been passed on to the PC Cluster Consortium [2]. SCASH [3] is a page based SDSM system based on the release consistency model. The SDSM also supports multiple writers through a twinning-and-diffing mechanism. OpenMP is supported on SCore through the Omni OpenMP compiler that maps OpenMP code to SCASH.

In previous work [9], the details of the SCASH page fault handling mechanism was outlined and the various types of page faults identified. These were based upon permutations of three conditions: access type (read/write), current permission for the page (unmapped/read only/write only), and page ownership – whether the page is owned by itself (home), by a thread residing within the same node on the cluster (local), or by some thread on another node of the cluster (remote). On a cluster consisting of 550 MHz Pentium III dual processor nodes linked via a 100 MBit Ethernet interconnect, it was found that page faults requiring page fetches took about 590 microseconds to resolve. In contrast, those that don't involve page fetches require only about 25 microseconds. The performance of the OpenMP implementation was also examined using the EPCC OpenMP synchronisation microbenchmarks [1]. The results showed that much of the overheads for the "parallel", "for", and "parallel for" OpenMP directives can be attributed to the implicit barriers associated with these directives. In other words, in this environment barriers are relatively expensive and their use should be minimised.

This paper extends the previous study to consider the performances of two application kernels; one that solves the 2D Laplace equation and another that performs a 1D fast Fourier transform (FFT). Of particular interest are the use of Omni OpenMP extensions to specify page placement, use of the special Omni OpenMP "mapping" directive and "affinity" scheduling clause OpenMP, and the challenge of implementing work sharing for a divide-and-conquer algorithm (FFT).

2 The 2D Laplace Equation

In this application kernel the temperature of a conducting plate is held constant at the edges and the aim is to determine the temperature of the interior of the plate. The prob-

lem is described by the 2D Laplace equation, and as such is similar to a number of other related problems. The equations are solved iteratively using a finite difference approach with a regular rectangular grid. In each iteration a new value of the temperature at a given grid point is computed based on the average of the temperatures of the four surrounding grid points, with iterations continuing until some agreed convergence is reached. Ignoring convergence testing, the basic sequential code is as follows:

```
/*1*/ for (i=0; i<no_of_iterations; i++) {
/*2*/   for (y=1; y<no_of_rows-1; y++)
/*3*/     for (x=1; x<no_of_cols-1; x++)
/*4*/       new[x,y] = (old[x+1,y]+
                        old[x-1,y]+
                        old[x,y+1]+
                        old[x,y-1])/4.0
/*5*/   tmp=new; new=old; old=tmp;
/*6*/ } /* next iteration */
```

The simplest way to parallelise the sequential code is to insert a "`#pragma omp parallel for`" directive before line 2 (Naïve approach). This provides for thread creation and work sharing based on a striped partitioning of the grid.

The implied barriers at the start and end of the parallel region means that the Naïve approach incurs two barriers per iteration of the i-for-loop. In earlier work [9], it was observed that barriers in Omni cluster-enabled OpenMP are expensive and that their cost increases greatly with the number of threads involved. Thus to reduce the number of barriers the "`#pragma omp parallel for`" is split into two separate directives. In this "barrier optimised" approach, a "`#pragma omp parallel`" directive is placed before line 1, and a "`#pragma omp for`" before line 2.

A second area for optimisation is data locality. The aim is to match data allocation and work sharing so that communication is only required at the boundaries of the striped partitions. Figure 1 illustrates a striped work sharing scheme where each thread works on three rows of data. The differently shaded boxes indicate page allocation. The default behaviour of `malloc()` in Omni is to allocate pages in a round-robin manner as shown in Figure 1 (a). It is evident that this does not match page placement to work sharing. To match allocation, the `ompsm_galloc()` function was used to achieve a block-wise page allocation (Figure 1 (b)). However, as the y-for-loop does not cover the first and last rows of the grid, work-sharing is actually distributed as follows: Thread 0 works on rows 1-3, thread 1 on rows 4-6, thread 2 on rows 7-9, and thread 3 on row 10. This does not match the block page placement. To fix this the y-for-loop was adjusted as follows:

```
#pragma omp parallel for schedule(static)
for (y=0; y<no_of_rows; y++) {
  if (y==0||y==no_of_rows-1) continue;
  ...
```

An alternative to manual page placement is to use the page placement extensions provided by Omni OpenMP. These are the "#pragma omni mapping" directive and the "schedule(affinity)" clause. The former allows the user to specify a block-wise page allocation to statically allocated arrays. The following code fragment is an example:

```
double mem[no_of_rows][no_of_cols];
#pragma omni mapping(mem[block][*])
```

The mapping can later be applied using the "schedule(affinity)" clause as follows:

```
#pragma omp parallel for \\
        schedule(affinity,mem[y][*])
for (y=1; y<no_of_rows-1; y++) {}
```

Note that for the compiler to recognise such mappings the argument to the "affinity" clause must have been an argument to a "#pragma omni mapping" directive. Thus, affinity scheduling is available only to statically allocated arrays.

From the above discussion, one sequential and six parallel implementations were considered:

0. Sequential
1. Naïve
2. Barrier minimised
3. Manual locality optimisation
4. Manual locality opt. with barrier minimisation
5. Affinity clause optimisation
6. Affinity clause opt. with barrier minimisation

2.1 Comparison of Implementations

Timing results for the sequential and parallel implementations run on 1, 2, and 4 nodes of the Pentium III cluster for a grid size of 1024x1024 and 100 iterations are given in Table 1. Each data point is an 8 byte double and the page size is 4096 bytes; so a grid size of 1024x1024 has 1024 rows of two pages each.

Even with the OpenMP overheads, the single thread performance of all parallel implementations are comparable to the sequential implementation. This is because in the underlying SCASH SDSM, heuristics are employed to check for the special case where there is only one thread.

Implementations 1 and 2 suffer a large increase in total time when the number of threads is increased from one

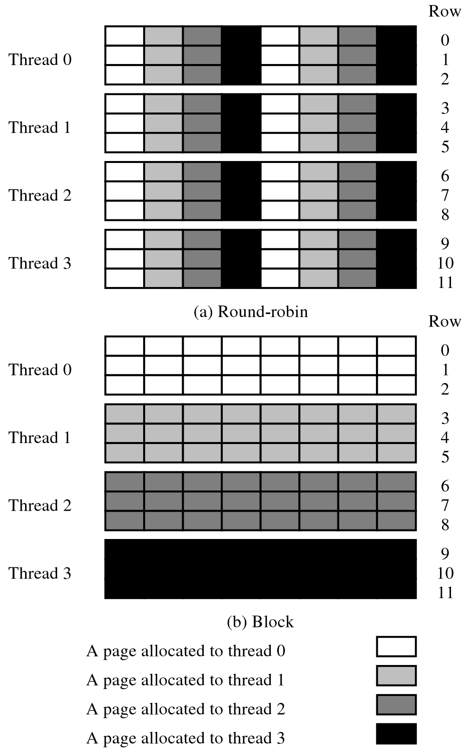

Figure 1. (a) Round-robin and (b) block page placements of grid data for the 2D Laplace equation.

	Time (seconds)			Speedup	
Impl	1	2	4	2	4
0	7.5	-	-	-	-
1	7.5	63.7	78.7	0.12	0.10
2	7.5	63.5	70.8	0.12	0.11
3	7.5	6.9	5.2	1.10	1.44
4	7.5	6.7	4.2	1.12	1.79
5	7.5	6.9	4.9	1.10	1.53
6	7.5	6.8	4.6	1.11	1.62

Table 1. Performance comparison of sequential heat code with six alternative parallel implementations run using SCore on the Pentium III cluster for a grid size of 1024x1024 over 100 iterations with 1, 2, and 4 threads. (To approximate a dedicated system, each test is rerun several time and its minimum used.)

to two. In contrast, implementations 3 and 4, which employ data locality optimisation, experience a slight decrease in total time. This is because implementations using the round-robin page placement incur page fault costs throughout the grid; whereas the implementations using the block page placement only incur page faults at the borders of the work-sharing domains.

Increasing the number of threads from two to four further increases the total time for implementations 1 and 2. This is due to two factors; first, the cost of barriers have now increased; and second, three pages out of every four are now remote (with two threads, every other page is a remote page). Even so, implementations 3 and 4 still experience a decrease in the total time; although increasing the number of threads does increase the number of domain borders over which communication is required, these can potentially take place in parallel and therefore not greatly increase the communication time.

Aside from very poor absolute performance, implementations 1 and 2 also exhibit large variability in the measured times. For instance average times for implementation 1 when using 1, 2 and 4 threads were 7.5, 92.8, and 127.0 seconds respectively with standard deviations 0.01, 16.96, and 28.43. In contrast, the averages for implementation 3, a solution with optimised locality, are 7.5, 6.9, and 5.3 seconds with standard deviations 0.01, 0.02, and 0.06; a markedly more stable result. This effect is due to the non-deterministic ordering of page requests and shows that data locality is important for both better and more predictable performance.

The effect of barrier minimisation is shown by the difference in the performances of implementations 3 and 4. (These implementations are used in favour over implementations 1 and 2 because the latter exhibit high timing variability.) The results show that barrier minimisation does indeed improve performance, especially as the number of threads increases.

The timings for implementations 3, 4, 5, and 6 shows that implementations using affinity scheduling have performances that are comparable to manual page placement and scheduling. Interestingly, the performance of implementation 5 is better than 3 on four threads, but implementation 6 is not as good as 4. The reason for this is not clear. In other cases however, the results show that affinity scheduling does incur a slight overhead penalty. An advantage of using affinity scheduling is that it offers a more elegant solution as the developer only needs to insert directives (not modify the sequential code). A distinct disadvantage, however, is that it is only available for statically allocated arrays.

2.2 Different Grid Sizes

To assess performance as a function of grid size implementations 0, 4, and 6 were run for grid dimensions: 256x256, 512x512, 1024x1024, and 2048x2048 (parallel

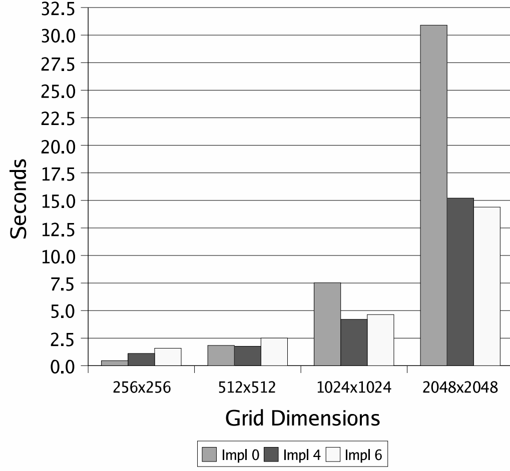

Figure 2. 2D Laplace performance of implementations 0, 4, and 6 over increasing grid sizes (100 iterations) on the Pentium III cluster. The parallel implementations are run using 4 threads.

implementations were run using four threads). The timings are displayed in Figure 2. As the edge of the square grid doubles, the problem size increases by four times giving $O(n^2)$ scaling. The bar plots for implementation 0 depicts this nicely, e.g. changing from ~7.5 to $\sim 4 \times 7.5$ seconds as the grid size increases from 1024 and 2048.

In Amdahl's law, the time taken to solve a problem is divided into a serial (S) and parallel (P) component such that with n_t threads the time to solution is $S + P/n_t$. Aside from the swapping of pointers (line 5), the rest of the 2D Laplace iterations are perfectly parallel, so S can effectively be ignored and the parallel run time approximated as P/n_t.

The performance for the parallel implementations in Figure 2 does not, however, exhibit P/n_t scaling. This is because Amdahl's law ignores the overheads associated with parallelism and in particular communication costs on a cluster are non-trivial. Implementations 4 and 6 only incur communication along the borders of the work division strips. Thus, while each thread's workload grows as $O(n^2)$, communication costs only grow as $O(n)$. Taking communication costs (C) into consideration, then the total time, T, can be expressed as:

$$T = \frac{P}{n_t} + C$$

As P has $O(n^2)$ runtime complexity while C grows as $O(n)$ gives:

$$T = \frac{t_p n^2}{n_t} + t_c n$$

where t_p is some constant processing cost and t_c is some constant communication cost. The timings for implementation 4 is 4.2 seconds on a 1024x1024 grid and 15.2 seconds on a 2048x2048 grid. Using these timings, we obtain $t_p = 0.13$ microseconds and $t_c = 7.81$ microseconds (results are divided by 100 iterations). The much larger t_c compare to t_p means that for small n, communication costs will dominate, but eventually as n is increased computation costs will dominate. In otherwords speedup will only be observed for large grid sizes.

3 The 1D Fast Fourier Transform

The 2D Laplace case study features an $O(n^2)$ iterative solution that is straight forward to parallelise. In contrast, the 1D Fourier transform involves a $O(n \log_2 n)$ recursive divide-and-conquer algorithm that is significantly more challenging. In this work we develop 1 sequential and 4 parallel implementations:-

Implementation 0: Two sequential implementations were considered initially. The first used a recursive approach with repeated calls to the same subroutine, while the second implemented the same algorithm but within one subroutine using an iterative approach. For small problem sizes the performance of the iterative scheme was found to be superior to the recursive approach. However, when the data size can no longer reside in the cache, the recursive approach is fastest. As a consequence, a hybrid scheme was implemented that uses the recursive algorithm until the data becomes cache resident and then switches to the iterative approach. The hybrid scheme is as follows:

Algorithm *Hybrid*(S, T, n, ω_n^1)
Input: S – the input data
Input: T – the temporary buffer
Input: n – the number of data points in S
Input: ω_n^1 – the second n^{th} roots of unity
Output: The result stored in S
1. **if** $n = 1$
2. **then return**
3. **if** ($n \times$ *bytes per datapoint*) $<$ (*cache size*)
4. **then** *Iterative*(S, n)
5. **return**
6. **for** $j \leftarrow 0$ **to** $n/2 - 1$
7. **do** $T[j] \leftarrow S[2j]$
8. $T[j + n/2] \leftarrow S[2j + 1]$
9. $\omega_n^2 \leftarrow \omega_n^1 \omega_n^1$
10. *Hybrid*$(T, S, n/2, \omega_n^2)$
11. *Hybrid*$(\&T[n/2], \&S[n/2], n/2, \omega_n^2)$

12. $\omega \leftarrow \omega_n^0$
13. **for** $j \leftarrow 0$ **to** $n-1$
14. **do** $k \leftarrow j$ **mod** $n/2$
15. $S[j] \leftarrow T[k] + \omega T[k+n/2]$
16. $\omega \leftarrow \omega\omega_n^1$

Both the splitting stage at line 6 and the combining stage at line 13 have $O(n)$ runtime complexity. Since there will be at most $\log_2 n$ recursions the total runtime complexity is $O(n \log_2 n)$.

Implementation 1: The initial parallel scheme is based on the recursive divide-and-conquer approach. Divide-and-conquer is an ideal work-sharing mechanism because the work is logically divided and undertaken separately. The scheme begins with g number of threads that work together at the splitting-odd-even stage; the for-loop at line 6 of Impl01. Because the iterations of this loop have to be scheduled for the team, then sub-team, and then sub-sub-team, as the recursion unfolds, a "#pragma omp for" directive cannot be used. Thus the iterations of this for-loop are scheduled manually using r – the rank of a thread within a sub-team, c – the number of iterations a thread is assigned, and b – the starting index of the loop for the thread. Therefore, the runtime complexity of this stage is $O(n/g)$.

At the recursion, the threads are divided into two sub-teams of $g/2$ threads each based on r. Each sub-team then recurses into either the odd or even branch. The base case for work-sharing is when g is one; in which case the lone thread will recurse into both odd and even branches. Barrier points are inserted before and after the work-sharing recursions as shown in algorithm Impl01. The first occurs before a work-sharing recursion and the second after. The first barrier ensure that the shared splitting-odd-even task has been completed; while the second ensures that the both subproblems have been completed by both sub-teams. Normal recursions where $g = 1$ do not need to be guarded by barriers. As in the sequential implementation, there will be at most $\log_2 n$ recursions.

Similar to the splitting-odd-even stage, the work at the combining-results stage is also shared by the threads in the sub-team. As before, c and b are calculated for scheduling the subsequent for-loop. The runtime complexity of this stage is also $O(n/g)$. Assigning a value to the initial ω in the parallel implementation (line 20) is slightly more costly than the sequential implementation, because the value of the initial ω now depends on the value of b; which differs among threads. This, however, doesn't impact much on performance as it is only done once per thread per work-sharing recursion.

The overall runtime complexity of the implementation is therefore $O((n \log_2 n)/n_t)$ where n_t is the total number of threads.

Algorithm *Impl01*(S, T, n, ω_n^1, g)
Input: S, T, n, ω_n^1 – as in *Hybrid*
Input: g – the number of threads in the sub-team
Output: The result stored in S
1. **if** $n = 1$
2. **then return**
3. $r \leftarrow$ *omp_get_thread_num()* **mod** g
4. $c \leftarrow (n/2)/g$
5. $b \leftarrow rc$
6. **for** $j \leftarrow b$ **to** $b+c-1$
7. **do** $T[j] \leftarrow S[2j]$
8. $T[j+n/2] \leftarrow S[2j+1]$
9. $\omega_n^2 \leftarrow \omega_n^1\omega_n^1$
10. **if** $g > 1$
11. **then** #pragma omp barrier
12. **if** $r < g/2$
13. **then** *Impl01*$(T, S, n/2, \omega_n^2, g/2)$
14. **else** *Impl01*$(\&T[n/2], \&S[n/2], n/2, \omega_n^2, g/2)$
15. #pragma omp barrier
16. **else** *Impl01*$(T, S, n/2, \omega_n^2, g)$
17. *Impl01*$(\&T[n/2], \&S[n/2], n/2, \omega_n^2, g)$
18. $c \leftarrow n/g$
19. $b \leftarrow rc$
20. $\omega \leftarrow \omega_n^b$ $\triangleright \cos(2\pi b/n) + i\sin(2\pi b/n)$
21. **for** $j \leftarrow b$ **to** $b+c-1$
22. **do** $k \leftarrow j$ **mod** $n/2$
23. $S[j] \leftarrow T[k] + \omega T[k+n/2]$
24. $\omega \leftarrow \omega\omega_n^1$

Figure 3 illustrates implementation 1 for the case with four threads. When $g = 4$, all four threads cooperate to split the data array into two halves. The four threads are then split into two sub-teams of two threads each. Each odd/even half is then visited by one sub-team. When the recursion reaches $g = 1$, each thread solves the rest of the recursion individually. Combining the separate results takes the route up the recursion tree. Similar to splitting, threads within each subteam work together to combine their data.

Implementation 2: If a chunk of work has been logically divided, then that chunk should be worked on in the fastest way possible. In this implementation, an additional check is placed after the base case (before line 3 of Impl01) to switch to the hybrid sequential implementation if $g = 1$. Also, there is no need to check for $g > 1$ at line 10 and the solo recursions at lines 16 and 17 can be removed.

Implementation 3: Implementation 1 focused only on work-sharing and made use of the default round-robin page placement for its data allocation. However, it is apparent from Figure 3 that the final subproblem (when $g = 1$) is a contiguous chunk of the original data array that would be best undertaken by a thread with local access to the data. Thus, this implementation improves upon implementation 0 by using a block page distribution for its data allocation.

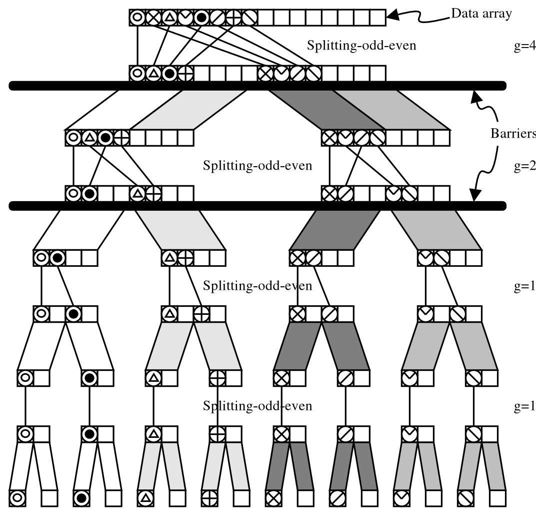

Figure 3. The path taken by each thread (shown with different shades of grey) through the recursion. After the stage where gang-size is one, each thread has to traverse down both branches.

Implementation 4: Optimisations introduced in implementations 2 and 3 are combined.

Table 2 lists the times obtained by the parallel implementations when the problem size is 524288 (2^{19}). Note that results are given using both 1 and 2 threads on each box. The table shows that all implementations perform poorer than the sequential implementation for the case of one thread (1x1). Implementations 1 and 3 are the slowest because they are essentially recursive sequential implementations which are slower than the hybrid sequential implementation. Implementations 2 and 4 are also slower because the implementation first enters the work sharing version of the recursive FFT only to discover that there is just one thread, and that it must now branch into the sequential hybrid implementation.

Implementation 2 performs better than implementation 1 just as implementation 4 performs better than implementation 3. This improvement is attributed to switching to the sequential hybrid scheme when the number of threads in a sub-team is one ($g = 1$).

The effect of the block page distribution can be observed by comparing the performances of implementations 3 to 1, and 4 to 2. Improvements are observed when threads are running on separate boxes of the cluster (2x1 and 4x1). For the case where all threads are on one box (1x1 and 1x2), performances for implementations 3 and 4 are similar to performances for implementations 1 and 2 respectively. This is because the effect of data locality is not applicable in these scenarios.

All implementations perform best with 1x2 threads, as they benefit from having more processors without incurring added network communication costs. No speedup is achieved when threads run on separate boxes (2x1 and 4x1 threads). This is true even for implementations 3 and 4 where a block distribution is used. For comparison Table 2 also includes performance data obtained from a 12 900MHz CPU Sun V1280 system that has hardware shared memory. This also shows limited scalability. It is also interesting to note that the OpenMP code run with one thread on the Sun is significantly slower than the same code compiled without activating the OpenMP directives.

The limited effect of using page placement can be explained by the data access patterns of the splitting-odd-even and combining-odd-even stages. The data access patterns involved at the splitting-odd-even stage are shown in Figure 4. The figure shows the case with four threads using a block distribution of data. The task of splitting involves reading from input array S and writing into output array T. When reading, each thread always reads data it owns. However, when writing, thread 0 writes to the first halves of sections 0 and 2 of T. Thread 1 fares worst as it has to write to the second half of the same two sections; both of which it doesn't own. The data access for threads 2 and 3 mirror those of threads 1 and 0 respectively.

Figure 5 shows the data access patterns involved during the combining-odd-even step. Combining data involves reading two sections from T and writing into one section of S. With the block distribution, the section in S to which each thread writes to is its own section. Although there are no remote write page faults, there are remote read page faults. One trait of the FFT is that each result data point is some composition of all input data points. Thus, there is no data distribution possible that would allow communication to scale at a slower rate than $O((n \log_2 n)/n_t)$.

Given this, the chief benefit of the block distribution is that no page faults occur when the number of threads in the sub-team is one ($g = 1$). Even this is only mildly effective as there are no barriers when $g = 1$; which means that the only increase in cost (if not using a block page distribution) are initial page faults for the first access and updating of changes when the threads meet again at the barrier point before the combine-odd-even stage of $g = 2$.

In order to assess further how costly communication is, we instrumented the underlying SDSM to collect counts for the various communication events such as, the number and type of page faults, and the number of messages and amount of data sent across the network. In previous work we measured the minimum cost of these events [9]. Thus by combining this previous timing data with the event counts obtained here it is possible to give an estimate of the communication time. Table 3 lists the communication events,

Cluster	Boxes x Threads Per Box 1x1	1x2	2x1	4x1
Impl 0	1.5			
Impl 1	3.4	2.3	5.9	5.5
Impl 2	1.6	1.4	5.0	5.1
Impl 3	3.3	2.0	3.7	3.3
Impl 4	1.6	1.4	2.8	2.9

Sun V1280	Threads 1	2	4
Impl 0	0.5		
Impl 2	0.7	0.4	0.3

Table 2. Times (seconds) obtained on the Pentium III cluster and a 12 900MHz CPU Sun V1280 system for the various FFT implementations when the problem size is 524288.

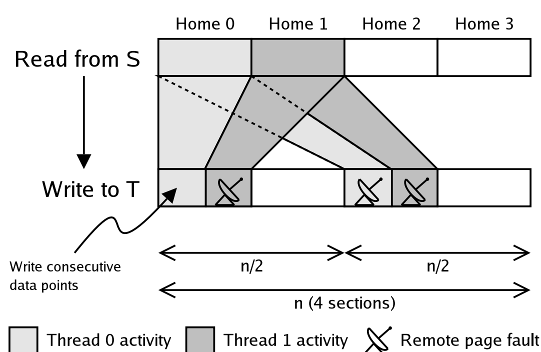

Figure 4. Splitting data of S into odd and even halves of T.

their counts, and the corresponding cost per count for implementation 4 and the case of 524288 data points and 4x1 threads. The table shows that at least 1.8 seconds of the total 2.9 seconds can be attributed to the cost of communication. Deducting this from the original execution time leaves 1.1 seconds, suggesting a potential speedup of 1.4 times if communication costs were zero. This is, however, much lower than the theoretical speedup of four that is given by the $O((n \log_2 n)/n_t)$ scaling. It must be remembered, however, that the "cost per count" timings in Table 3 are minimum timings and under normal conditions the real costs are likely to be higher. If instead we divide the sequential time of 2.9 seconds by four, we obtain 0.7 seconds, suggesting that communication costs are actually $2.9 - 0.7$ or about 2.2 seconds. In practice the real value will be between 2.2 and 1.8 seconds.

The parallel implementations described above only make

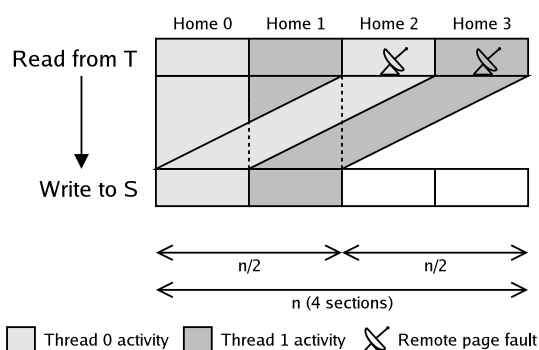

Figure 5. Combining the results from each half of T into S.

use of the "#pragma omp parallel" and "#pragma omp barrier" directives. The "#pragma omp for" directive could not be used for loop scheduling (for splitting-odd-even and combining-odd-even) because that would enforce a scheduling that rostered *all* threads when only the threads within each sub-team should be scheduled for that subproblem. This is also a problem for the case of "#pragma omp barrier". In Figure 3, the second barrier is enforced across all threads when it really should be two barriers, one for each logical portion of the problem. Hence, the OpenMP standard is lacking the directives necessary to express work scheduling in a divide-and-conquer environment.

A possible extension to the OpenMP standard that might alleviate this deficiency would be the following:

```
#pragma omp divide \\
  partition(block,nGangs,resolution){}
#pragma omp divide \\
  partition(roundrobin,nGangs,chunkSize){}
```

The "divide" directive would divide the threads into *nGangs* gangs. The directive would be followed by a structured block like the "parallel" directive. The end of the structured block indicates the end of the divide. In order for the "divide" directive to be useful, it would need to be able to be nested. Otherwise, it wouldn't work with recursive algorithms. The first variant divides the threads into approximately equal sized gangs; differing only by at most *resolution* threads. The second divides the original number of threads into *chunkSize* chunks and distributes the chunks to *nGangs* gangs in a round-robin fashion. Other possible variants might divide the threads into gangs of size related to some power (e.g. a power of 2). However, that would require further investigation to consider the extent of its applicability.

Within the structured block, the function omp_get_num_threads() should return the number of threads

Details	Count	Cost per count (microseconds)	Cost (microseconds)
Remote write page faults	768	599	460032
Remote read page faults	1026	587	602262
Barriers	6	7305	43830
Send2	4170	61.66	257122
Send2 Bytes	2384168	0.181919	433725
Total communication cost			1796971

Table 3. Communication counts and corresponding costs for FFT implementation 4 with 524288 data points and run over 4 boxes with 1 thread per box (4x1)

in the gang. Similarly, the function omp_get_thread_num() should return the rank of the thread within its gang. In addition to these functions, some algorithms may need to know which gang a thread belongs to. To fill this need, the omp_get_num_gangs() and omp_get_gang_num() functions would be included as part of the extension. If there are no gangs, the former function should return 1, and the latter 0. Functions aside, directives that relate to the number of threads should also adapt to the new environment. Specifically, the “for” directive should now schedule based on the number of threads in the gang. Also, the “barrier” directive should now apply only to gang members. The question of whether “master” should refer to thread 0 of each gang would need to be addressed.

4 Conclusions

A recurring theme throughout this paper is the importance and impact of data locality on the performance of cluster OpenMP applications. The results of the two case studies showed that this is especially important for iterative algorithms as the penalty associated with poor data locality is repeated at every iteration. For divide-and-conquer schemes, data locality appears to have less of an impact on performance.

In the 2D Laplace case, the “mapping” and “affinity” extensions were explored and found to be effective with relatively low overheads. The solutions using these extensions were also found to be more elegant as they did not require modifications other than insertion of OpenMP directives (and extensions). One limitation of these extensions, though, is that it can only be used for statically allocated arrays.

Finally, parallelising the 1D fast Fourier transform highlighted a lack of OpenMP directives and clauses for dealing with divide-and-conquer work sharing algorithms. Having such directives is desirable as divide-and-conquer schemes are relatively common.

5 Acknowledgements

The authors gratefully acknowledge discussions with J. Antony and A. Over. This work was supported in part by the Australian Research Council through Linkage Grant LP0347178.

References

[1] Edinburgh parallel computing centre. www.epcc.ed.ac.uk.

[2] PC Cluster Consortium. www.pccluster.org.

[3] H. Harada, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, and I. Ishikawa. SCASH: Software DSM using high performance network on commodity hardware and software. In *Eighth Workshop on Scalable Shared-memory Multiprocessors*, pages 26–27. ACM, May 1999.

[4] Y. C. Hu, H. Lu, A. L. Cox, and Z. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, Dec. 2000.

[5] L. Huang, B. Chapman, Z. Lui, and R. Kendall. Efficient translation of openmp to distributed memory. In *Proceedings of the 4th International Conference on Computational Science (ICCS 2004)*. Springer-Verlag, 2004.

[6] D. Margery, G. Vallée, R. Lottiaux, C. Morin, and J.-Y. Berthou. Kerrighed: a SSI cluster OS running OpenMP. In *Proc. 5th European Workshop on OpenMP (EWOMP '03)*, September 2003.

[7] S.-J. Min, A. Basumallik, and E. Eigenmann. Optimizing OpenMP programs on software distributed shared memory systems. *International Journal of Parallel Programming*, 31(3):225–249, June 2003.

[8] Y. Ojima, M. Sato, H. Harada, and Y. Ishikawa. Performance of cluster-enabled OpenMP for the SCASH software distributed shared memory system. In *Proc. of the 3rd IEEE/ACM Int'l Symp. on Cluster Computing and the Grid (CCGrid'03)*, pages 450–456, May 2003.

[9] H. J. Wong and A. P. Rendell. The SCore cluster enabled OpenMP environment: Performance prospects for computational science. *International Conference on Computational Science*, 2005 (submitted).