CS 757 - Parallel Computer Architecture

Spring 2014

THE UNIVERSITY of WISCONSIN MADISON

Aman Rakesh Chadha
(906 835 4597)

Ranjini Mysore Nagaraju
(906 924 4441)

1 Overview

Ocean is a simulation of large-scale sea conditions from the SPLASH benchmark suite. It is a scientific workload used for performance evaluation of parallel machines. For this assignment, we have written two scaled-down versions of the variant of the Ocean benchmark.

Our version of Ocean simulates water temperatures using a large grid of integer values over a fixed number of time steps. At each time step, the value of a given grid location will be averaged with the values of its immediate north, south, east, and west neighbors to determine the value of that grid location in the next time step (total of five grid locations averaged to produce the next value for a given location).

2 Problem 1: Sequential Ocean

a Source code

```
#include <stdio.h>
void ocean (int **grid, int xdim, int ydim, int timesteps)
{
    /******* the red-black algorithm (start) ********/
    /*
    In odd timesteps, calculate indices with - and in even
    timesteps, calculate indices with *
    See the example of 6x6 matrix, A represents the corner
    elements.
        A A A A A A
        A - * - * A
        A * - * - A
        A - * - * A
        A * - * - A
        A A A A A A
    */

    int tsteps, i, j;

    for (tsteps = 0; tsteps < timesteps; tsteps++) {
      for (i = 1; i < (xdim-1); i++) {
        //handle the alignment of *'s and -'s as an offset
         int offset = (i+tsteps)%2;

        for(j = (1+offset); j < (ydim-1); j += 2) {
          grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
          + grid[i][j-1] + grid[i][j+1])/5;
        }
      }
    }

    ///////// the red-black algorithm (end) /////////
}
```

b Arguments on the validity of implementation

Our implementation seeks to average values of each grid location over a series of timesteps. If the operation is done over a sufficiently large number of such timesteps, we expect to obtain a grid with its values distributed over a lesser range (due to convergence) as compared to the original grid. This is similar to a Gaussian Blur operation in Image Processing.

Indeed, when we initialized the grid to a 130x130 size, and performed the averaging operation over a large number of timesteps (100, in our case), we obtained the expected results. Here, X Axis represents the X co-ordinate of the grid location and Y Axis represents the Y co-ordinate of the grid location. Figure 1 shows the input grid to our ocean implementation.

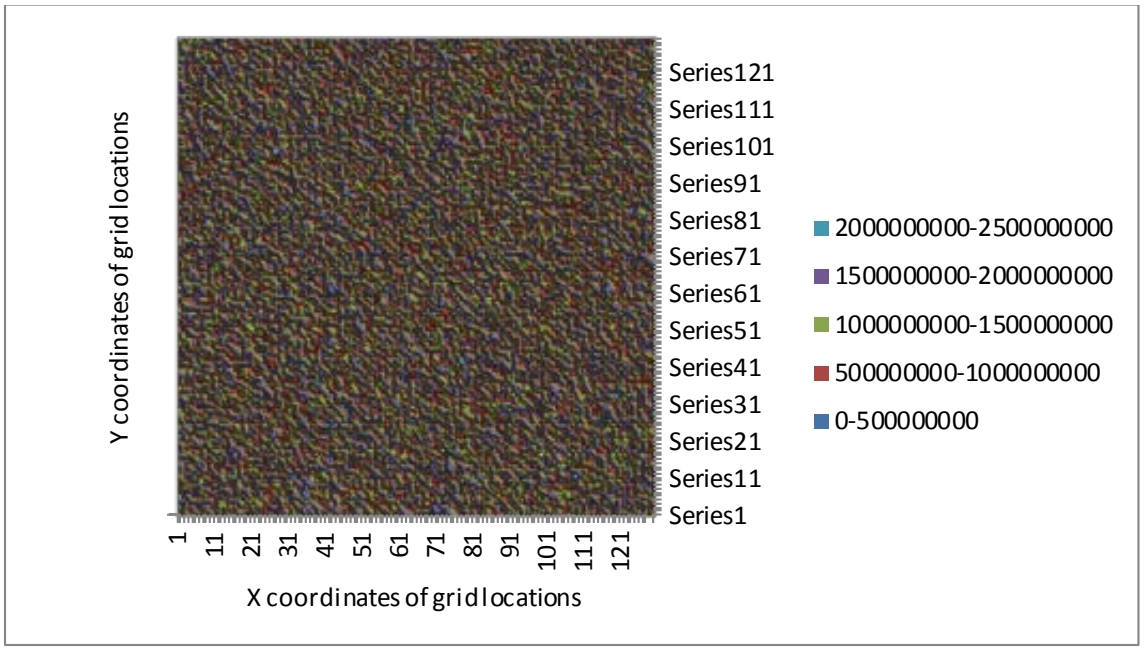

Figure 1: Input grid

After performing the averaging operation over 100 timesteps, we see the values at the grid locations (apart from those at the edges) tend to converge to a single value. This is a direct product of the averaging operation. Figure 2 shows the output grid from our implementation.

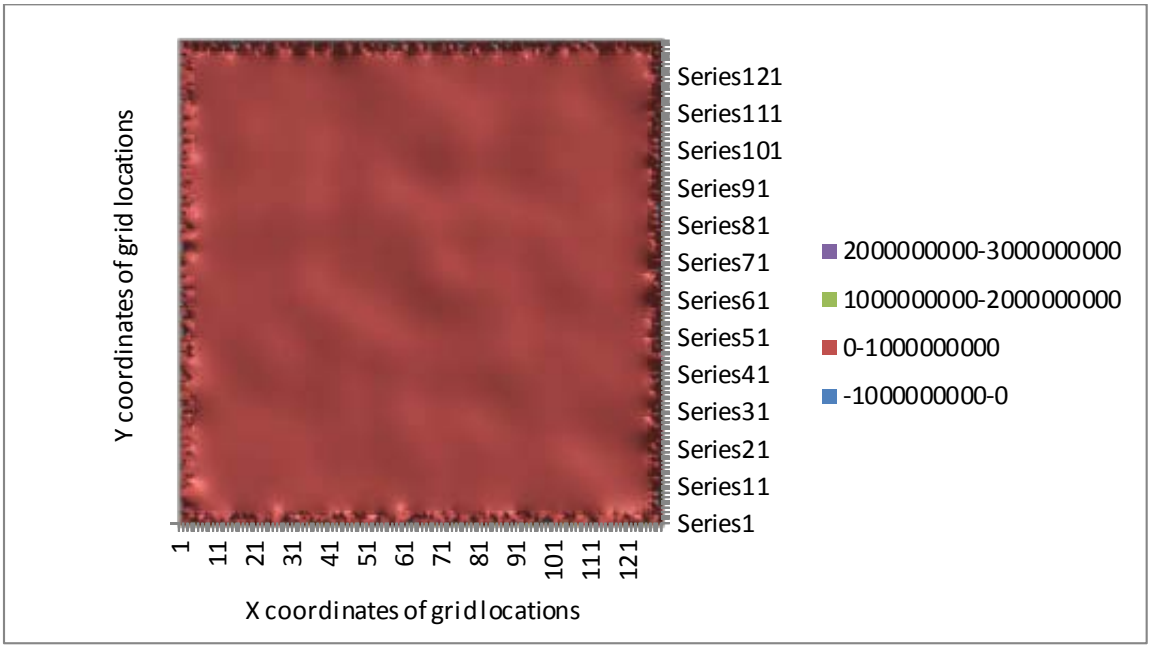

Figure 2: Output grid after 100 time steps

Further, upon performing the averaging operation over 10000 timesteps, the values get even closer to converging to a single value as shown in Figure 3.

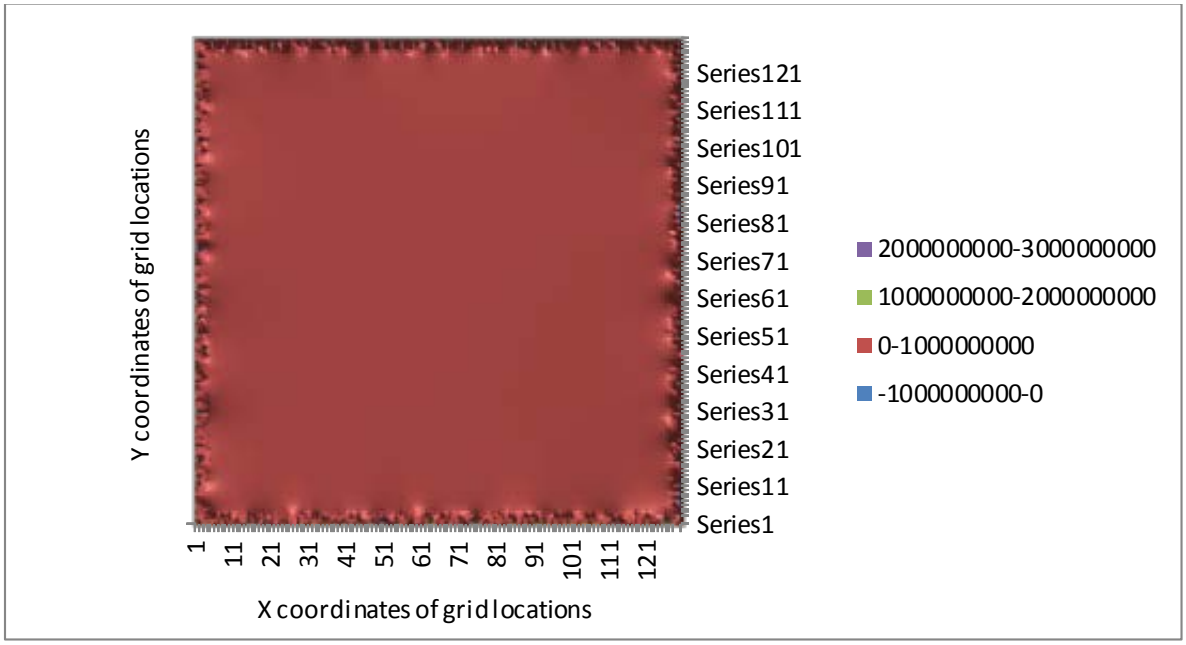

Figure 3: Output grid after 10000 time steps

3 Problem 2: Parallel Ocean using OpenMP

a Source code

```
#include <omp.h>
void ocean (int **grid, int xdim, int ydim, int timesteps)
{
  int tsteps,i,j,offset;

  //indicates that the number of threads can be adjusted during run
  time (for dynamic scheduling)
  omp_set_dynamic(1);

  for(tsteps = 0; tsteps < timesteps; tsteps++) {
    #pragma omp parallel for shared(grid,xdim,ydim) private(i,j)
    schedule(dynamic)
    for(i = 1; I < xdim-1; i++) {
      offset = (i+tsteps)%2;
      for(j = 1+offset; j < ydim-1; j += 2) {
           grid[i][j] = (grid[i][j] + grid[i-1][j] + grid[i+1][j]
          + grid[i][j-1] + grid[i][j+1])/5;
      }
    }
  }
}
```

b Comments on implementation

We exploited loop-level parallelism by starting off from our serial implementation and using an OpenMP compiler directive for parallelizing loops. The syntax for such a directive is:

sentinel	directive-name	clause
#pragma omp	parallel for	shared(var1) private(var2)

We declared loop variables as private to avoid interference between threads. The 'grid', 'xdim', and 'ydim' variables were declared to be shared across all threads since they are not written to, by any thread and are common across all threads.

This also helps in exploiting the benefit of shared memory, since setting the 'grid' variable as private would entail having a private grid for each thread, leading to a significant memory consumption for a large grid such as 4098x4098 across 8 threads.

We parallelized only the middle for loop, hence distributing a bunch of rows between threads as we found this strategy optimal. We also experimented with parallelizing both middle and inner for loops which resulted in finer grained sharing of grid by threads. Such fine grained data set distribution results in increased interference between threads. We understand that the optimal way of dividing the grid between threads is to assign square sized sub grids to each thread as this increases amount of computation per thread and decreases interference between threads. Since such a square sized division of data complicates our algorithm, we settled for parallelizing only middle loop. As expected, we obtained better performance by parallelizing only the middle loop. The simulations support the principle that we should parallelize only when speedup obtained is more than the overheads introduced.

c Arguments on the validity of implementation

The results of the parallel OpenMP implementation were matched against that of the serial implementation. As expected, the resultant grid values with and without the parallel pragma, were equal.

4 Problem 3: Analysis of Ocean

a Comparison of parallel v/s serial Ocean implementations for a 4098x4098 grid

For 100 timesteps, and a grid size of 4098x4098, the speedup plot is as follows:

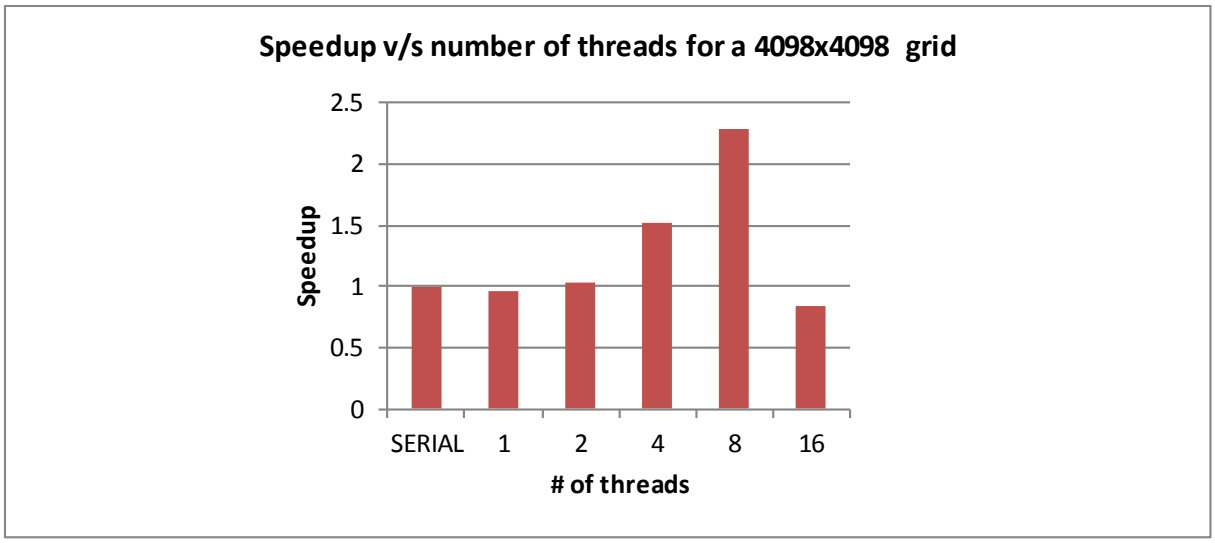

Figure 4: Speedup v/s number of threads for a 4098x4098 grid

Figure 4 shows a plot of speedup obtained by parallelization using OpenMP. As expected, the parallel version with single thread performs worse than serial version. This is due to inherent overheads of parallelization. Further, we can observe performance increasing with increasing number of threads, though this is a sub-linear speedup. Reasons behind such sub-linear speedup can be attributed to:

1. Only the *for* loop is parallelized while rest of the program still executes in serial fashion.

2. There is some amount of inherent communication between threads at the boundaries of data blocks assigned.

3. Inherent overheads of parallelization.

Also, as thread count increases beyond 8 we observed performance degradation. This is due to the fact that the *ale* machine has 8 cores. While using 16 threads, they can be context switched out of a core. In the event that a thread context switched out from one core gets scheduled on another core, its working set, TLB entries etc. need to migrate to the new core. This is not very unlikely to happen. By setting environment variable to set thread affinity that binds threads to processor, we observed better results.

b Comparison of parallel v/s serial Ocean implementations for a 8194x8194 grid

For 100 timesteps, and a grid size of 8194x8194, the speedup plot is shown in Figure 5. We can observe similar trends in performance as in the case of 4098x4098.

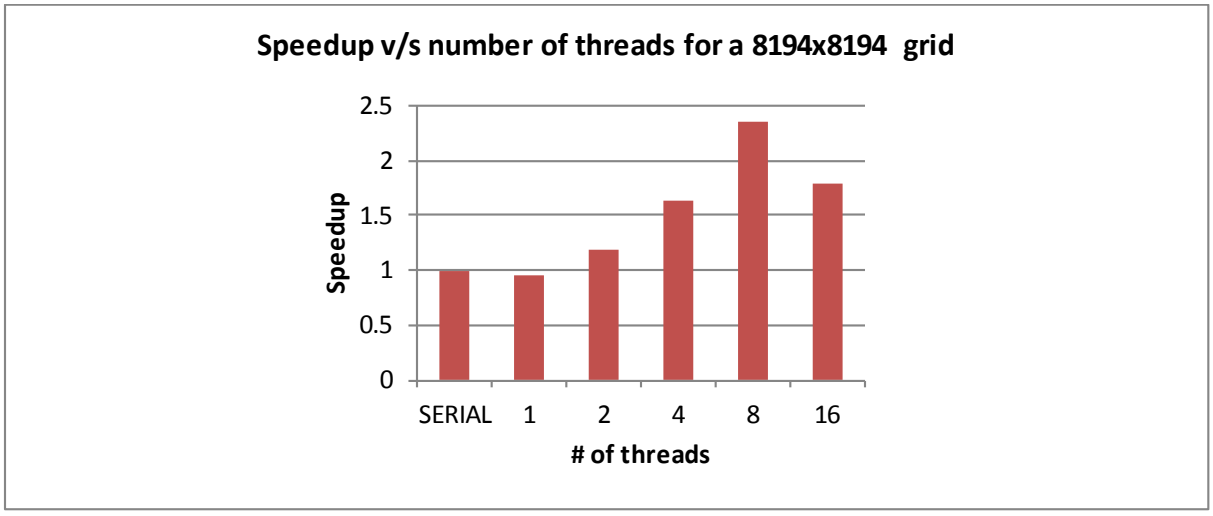

Figure 5: Speedup v/s number of threads for a 8194x8194 grid

5 Problem 4: Parallelization using Static Partitioning

a Dynamic scheduling v/s static scheduling for a 4098x4098 grid

For 100 timesteps, and a grid size of 4098x4098, the normalized speedup plot is as follows:

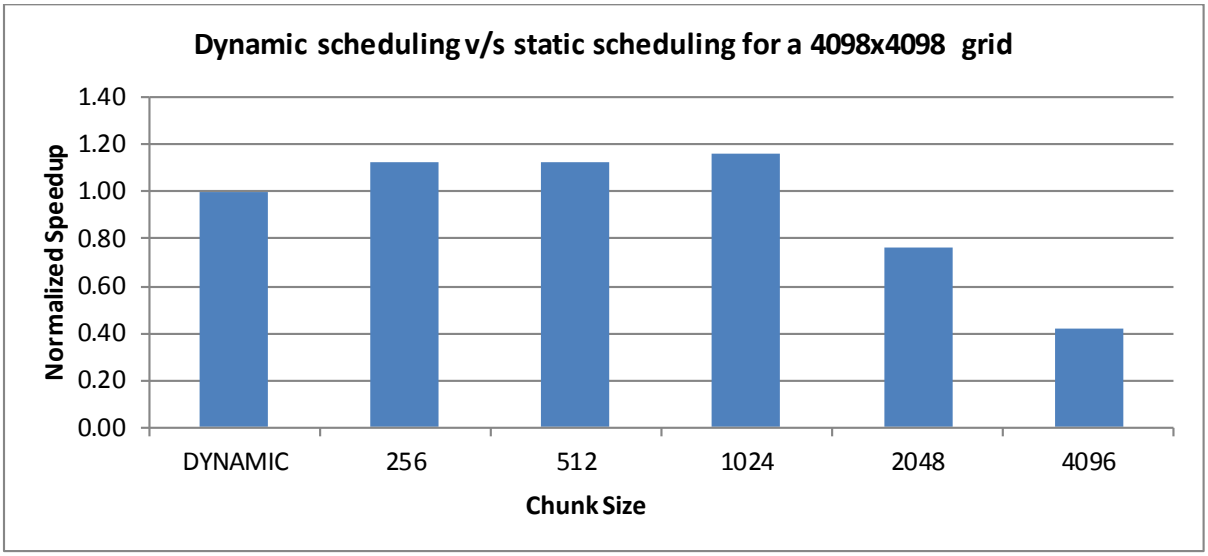

Figure 6: Dynamic scheduling v/s static scheduling for a 4098x4098 grid

Figure 6 shows a plot of performance using static scheduling versus dynamic scheduling for grid size 4098x4098. We can observe that performance improves in increase in chunk size. It reached a maximum at chunk size of 1024 and then started to decline. After the chunk size is large enough to divide data grid between the threads, further increase in chunk size introduces load imbalance thus leading to poor performance.

b Dynamic scheduling v/s static scheduling for a 8194x8194 grid

For 100 timesteps, and a grid size of 8194x8194, the normalized speedup plot is as follows:

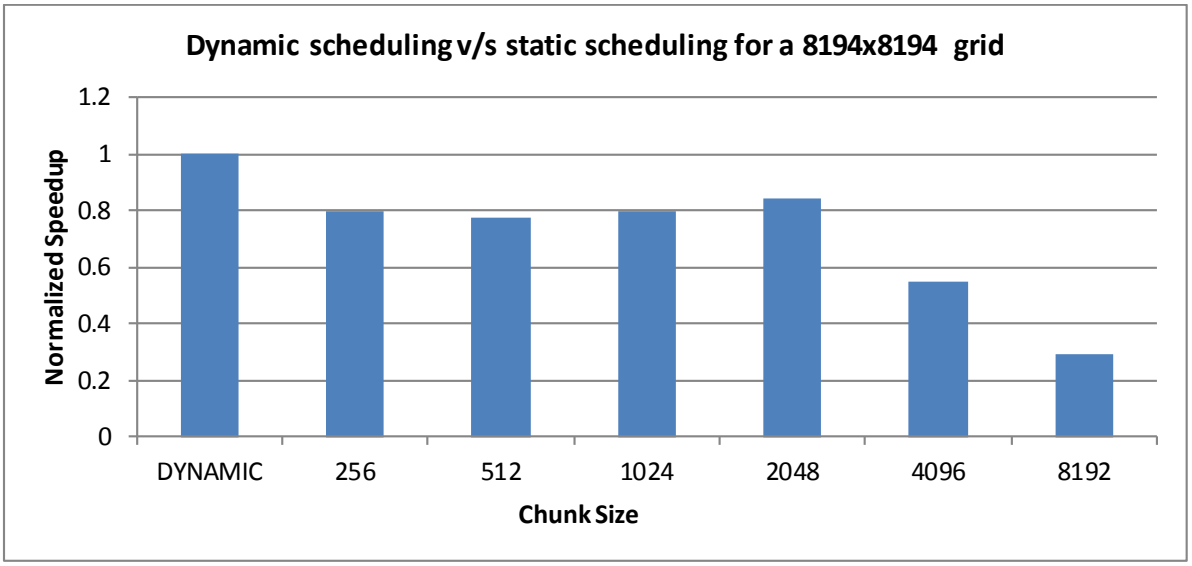

Figure 7: Dynamic scheduling v/s static scheduling for a 8194x8194 grid

Figure 7 shows a plot performance using static versus dynamic scheduling for grid size of 8194x8194. We observe that dynamic scheduling performs better in this case. This grid is four times larger than the previous and takes much longer time to execute. During such long executions, some threads might be context switched out for some time and there are multiple other reasons for different threads to finish at different times even though they are doing same amount of work. Here statically dividing grid into equal sized chunks will create a load imbalance towards end of the execution.

c Identification of grid sizes for which static scheduling performs better than dynamic scheduling

Form our experiments with varying chunk sizes and grid sizes we observed that for small grid sizes, static scheduling gives better performance. Programmer can identify the best way to divide work between threads.

However as grid size increases, we observed decline in performance using static scheduling. It seems best to dynamically divide and hand work out to threads that finish early and are free. We believe there will be some load imbalance towards end of execution for small grid sizes too, but it seems to be negligible compared to large grid sizes.

Please refer the last page for the data. For 100 timesteps and the number of threads as 8, the cells corresponding to the grid sizes for which static scheduling performs better than dynamic scheduling have been highlighted.

6 Analysis/Explanation of Trends

a Better performance with static partitioning than dynamic partitioning

With static scheduling, programmer can take advantage of his knowledge of data size and structure. As programmers we worked to increase computation for each thread while keeping the communication to the least possible. This avoids expensive communication between threads and results in better performance.

The flipside to using static scheduling is that the prior decision of chunk size, is not adjustable at runtime. Towards the end of program it can introduce load imbalance by handing out chunk sized work to one thread while it would have been a better decision to hand out smaller sized workloads to more threads that are free. Dynamic scheduling changes chunk size dynamically depending on runtime conditions and avoids this imbalance. As we can see both have their own tradeoffs. Overdoing any type of scheduling hurts performance.

b Differences between the speedup slopes for the grid sizes 8194x8194 and 4098x4098

We observed better performance for a grid size 8194x8194 than for a grid size of 4098x4098. Increase in grid size provides increased computation to communication ratio for each thread thus increasing achievable speedups.

c Any changes in the slope for each of the grid sizes separately

These are explained along with the plots before.

d Sources of super-linear speedups

We did not observe any super-linear speedups.

e Sources of sub-linear speedup

These are explained along with the plots before.

f Trends observed in the plot of normalized speedup v/s chunk size for both grid sizes

For a grid size of 4098x4098, we see that static scheduling produces a speedup for chunk sizes ranging from 256 to 1024. Beyond this value of chunk size, we see no speedup with static scheduling. Infact post a chunk size of 4096, the amount of computation time needed with static scheduling is seen to remain constant.

Similarly, for a grid size of 8194x8194, we see that static scheduling produces absolutely no speedups for chunk sizes ranging from 256 to 8388608.

We observed that static scheduling performs better for smaller grid sizes while dynamic scheduling performs better for larger grid sizes. Reasons are explained before.

7 Appendix: Data obtained from Experiments

Problem 1				
Serial Grid; 100 Timesteps				
Grid	**Time (ns)**	**Time (ns)**	**Time (ns)**	**Average (ns)**
4098	2738787631	2738879529	2748880898	2742182686
8194	10922197076	10923139348	10941775980	10929037468
Problem 3				
4098 4098 100				
Threads	**Time (ns)**	**Time (ns)**	**Time (ns)**	**Average (ns)**
1	2844308206	2850230507	2846907994	2847148902
2	2689156026	2649644247	2657248108	2665349460
4	1776152920	1816838430	1812124416	1801705255
8	1213083936	1171957168	1215978360	1200339821
16	3405194738	3225023570	3242142269	3290786859
8194 8194 100				
Threads	**Time (ns)**	**Time (ns)**	**Time (ns)**	**Average (ns)**
1	11347133675	11353771499	11359308880	11353404685
2	9304450408	9167376238	9298296676	9256707774
4	6743317453	6768657133	6661745441	6724573342
8	4630509957	4625718722	4630531422	4628920034
16	6167635631	6071444726	6077641386	6105573914

Problem 4																
8 Threads; 100 Timesteps																
STATIC																
Chunk / Grid	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**	**262144**	**524288**	**1048576**	**2097152**	**4194304**	**8388608**
258	11247836	11376467	11260906	11354074	11243855	11361719	11318716	11282210	11385863	11326823	11333732	11299894	11427641	11339236	11295601	11407931
514	29936487	40779332	40744622	40842538	40793044	40787365	41037976	40887048	40838467	40791061	40794452	40793035	40827639	40793188	40835818	40756996
1026	66548881	103598188	169430066	170504478	170453736	170221979	171877445	171765599	169795604	170920566	169317641	170943142	170939653	169858409	171531765	170849307
2050	229260729	250049688	392444792	719921308	704350184	706434903	701556216	703010600	703429887	702982063	702369467	705278108	724282687	702822011	702710428	700755560
4098	1064741837	1065537953	1033280376	157500527 8	284668831 2	283974098 4	284762349 5	285949504 2	283944345 9	285829994 8	283814393 3	283857834 1	283609324 0	283871482 6	283599641 9	284914959 4
8194	4136439516	424865164 8	411814487 3	390084725	599843154 3	113420433 19	113302475 96	113309732 88	113499990 01	113217189 89	113342823 74	113504984 97	113284774 30	113476378 93	113419159 61	113598698 04

Problem 4		
8 Threads; 100 Timesteps		
Grid	**DYNAMIC**	**SERIAL**
258	14840250	9351916
514	42360311	37006913
1026	99799304	165911608
2050	334189420	678148748
4098	1200339821	2743415491
8194	4628920034	10931433380