Parallelize Jacobi Relaxation Using MPI

Experimentation data for Serial computation

Serial Computation results with P^* = 1 and I_{max} = 100				
(N,R,C)	Runtime (sec)	Iteration	Tolerance	(R,C)
(2304,2302,2300)	8.34179211	100	0.0012070087	0.18192301
(4320,4318,4316)	29.25405598	100	0.0012070087	0.18192301
(10080,10078,10076)	159.50311589	100	0.0012070087	0.18192301

*: P denotes the number of processes.

Row Decomposition

Overview of the Implementation

While implementing, we assumed that the grid length N is evenly divisible by the number of processors P.

Each process is responsible for the boundary value updation and grid computation of N/ P number of contiguous rows. During the communication phase, processes exchange their boundary rows to the neighboring processes (for example, a process with rank r will send its bottom-most row and topmost row to process with rank r+1 and r-1 respectively and receive the topmost row from process r+1 and bottom-most row from rank r-1. Obviously for process with rank 0 and P-1, there will be only one send and receive because they have only one neighbor instead of two. Moreover, for each iteration of grid computation, each process will be sending/ receiving the entire row as one message.

Experimentation data for Row Decomposition

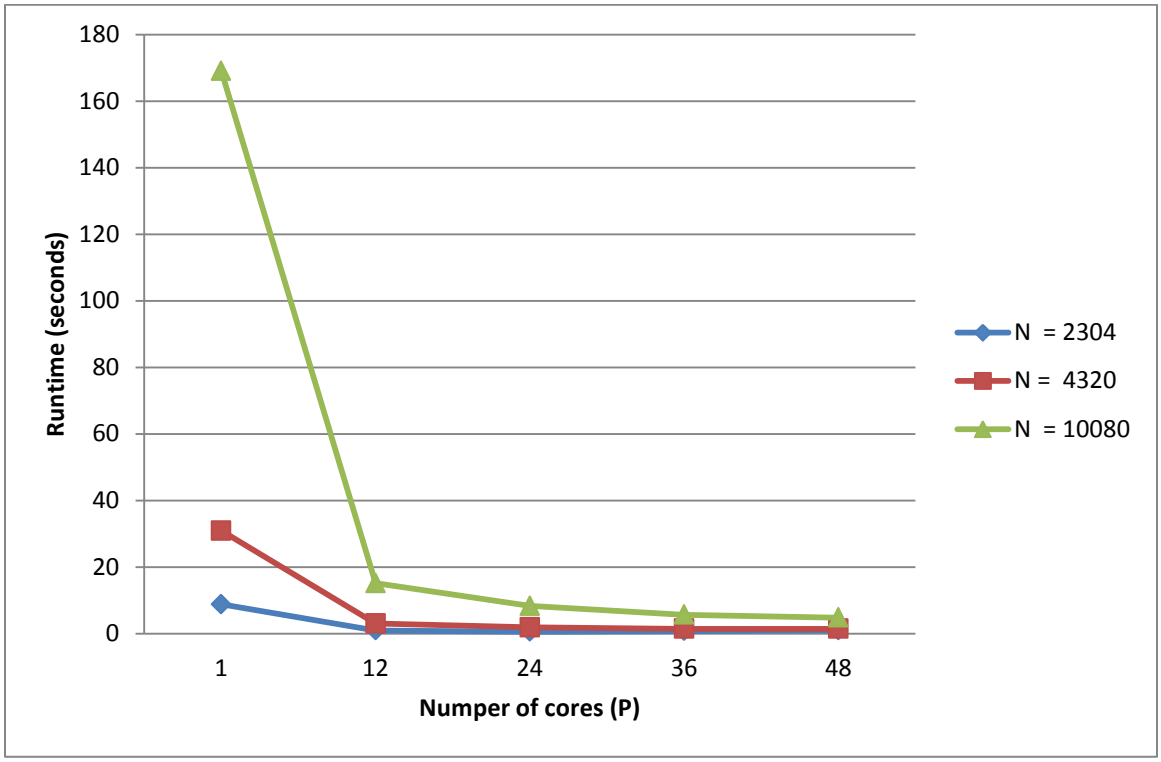
Tabular Representation

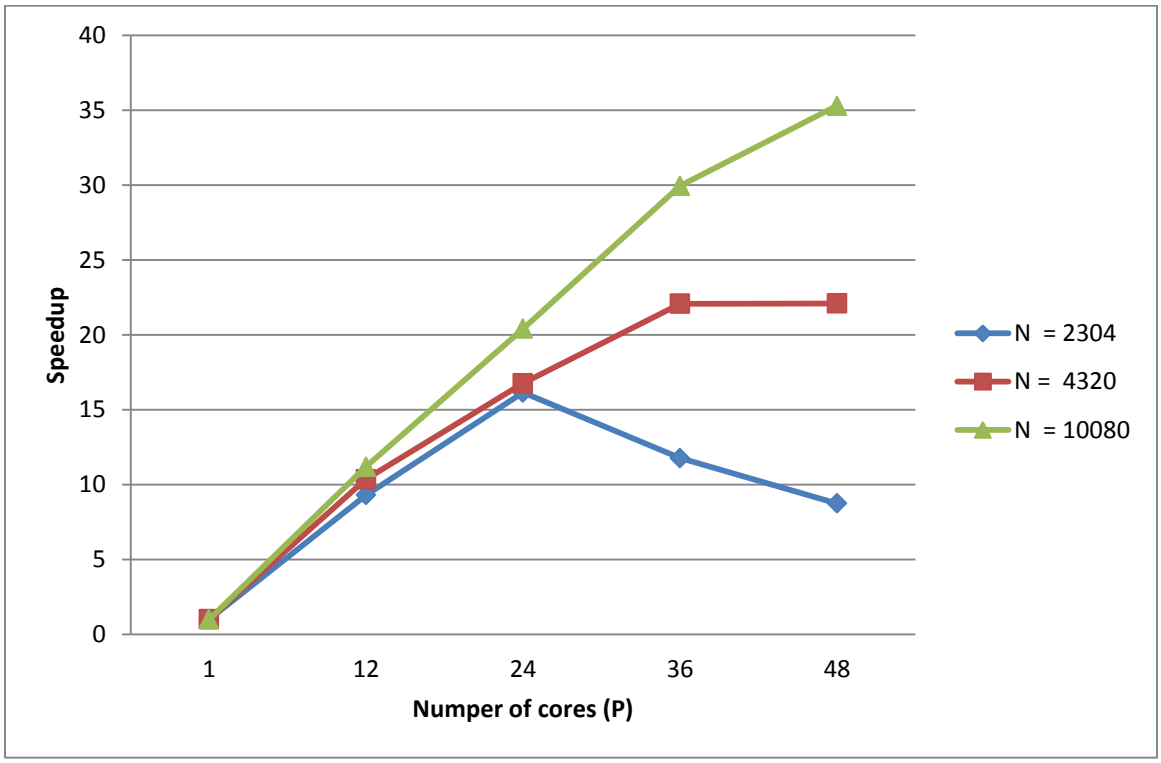
N = Grid length
(R,C) = A single floating-point value for the temperature at cell (R,C)
Tolerance = The value of *gmaxdiff* in the code, which gives the maximum change for any point below the threshold value,
Iteration = The maximum number of iterations reached.

Note: The speedup is calculated using (runtime for P=1 / runtime for P=p) for a given value of N.

P	Runtime (seconds)	Speedup	Iteration	Tolerance	(R,C)
Row Decomposition with (N,R,C) = (2304,2302,2300) and I_{max} = 100					
1	8.79747081	1	100	0.0012070087	0.18192301
12	0.94678211	9.2919	100	0.0012070087	0.18192301
24	0.54512596	16.138	100	0.0012070087	0.18192301
36	0.74793601	11.762	100	0.0012070087	0.18192301
48	1.00662494	8.739	100	0.0012070087	0.18192301
Row Decomposition with (N,R,C) = (4320,4318,4316) and I_{max} = 100					
1	30.88196492	1	100	0.0012070087	0.18192301
12	2.98439908	10.347	100	0.0012070087	0.18192301
24	1.84293103	16.756	100	0.0012070087	0.18192301
36	1.39908290	22.073	100	0.0012070087	0.18192301
48	1.39832211	22.0850	100	0.0012070087	0.18192301
Row Decomposition with (N,R,C) = (10080,10078,10076) and I_{max} = 100					
1	169.08435583	1	100	0.0012070087	0.18192301
12	15.11947107	11.183	100	0.0012070087	0.18192301
24	8.28606009	20.405	100	0.0012070087	0.18192301
36	5.64644909	29.945	100	0.0012070087	0.18192301
48	4.79273486	35.2793	100	0.0012070087	0.18192301

Graphical Representation





Analysis

Q. If the grid size were not divisible by the number of cores, how would you need to modify your program? How would this impact performance?

Answer.

Let suppose that we have **n** as grid size (say number of rows) and **num_processors** as the number of cores and **remaining_work** (> 0 and < num_processors) is the remainder of n/ num_processors.

We have to distribute remaining_work rows, one to each of num_processors cores. In this way there will be cores (= remaining_work) having (N/ P + 1) rows and some (= num_processors – remaining_work) having N/ P rows.

Code modification:

```
120
121                         //Partitioning the rows among the diffrent processes
122
123     //proc_boundary will save the row boundaries for each processor.
124     process_boundary *proc_boundary = (process_boundary *) malloc(numprocessors*sizeof(process_boundary));
125
126     int row_partition = n/numprocessors;
127     int remaining_work = n%numprocessors;
128
129     int actual_work;
130     for(i=0; i<numprocessors; i++) {
131         actual_work = row_partition;
132         if(0 != remaining_work) {
133             actual_work +=  1;
134             remaining_work --;
135         }
136         proc_boundary[i].i_first = actual_work*i + 1 ;
137         proc_boundary[i].i_last  = proc_boundary[i].i_first + actual_work;
138     }
```

In this way, the work will be distributed 'nearly equal' among the cores.

Impact of computation

This will lead to load imbalance as there will be (num_processors – remaining_work) cores having lesser work than the others. So during the grid computation phases some process will remain idle, leading to resource under-utilization and poor runtime than the case where n is exactly divisible by num_processors.

1. For example: let n = 16 and consider the two cases where number of cores, P is 4 vs 3.

In the former case each four of the cores will do the parallel computation for a block of 4 rows each, where as in he later case, (P0,P1,P2) will do the parallel computation for blocks containing (6,5,5) rows. Obviously P0 will become the bottleneck for the later case and decides the running time which will be greater

than the former case (as the computation time on 4 rows is lesser than that of 6 rows)

Impact on communication (when the cost of computation is almost the same)

2. Consider n = 16 and the two cases where number of cores, P is 4 vs 5.

In the former case each four of the cores will do the parallel computation for a block of 4 rows each, where as in the later case, (P0,P1,P2,P3,P4) will do the parallel computation for blocks containing (4,3,3,3,3) rows. In terms of computation, both the cases will take the same time (as P0 will become the bottleneck here and all other cores need to wait for it before MPI_Allreduce). But in the later case number of boundary communication per iteration is 4 as compared to 3 in the former case.

Revisiting Example 1, we can find that even though the computation cost increases for P = 3, but the communication cost decreases. But with very large values of n, the computation cost due to load imbalance (~ (n^2/P)*I, where I is the number of iterations) will very high as compared to communication cost (~n*I).

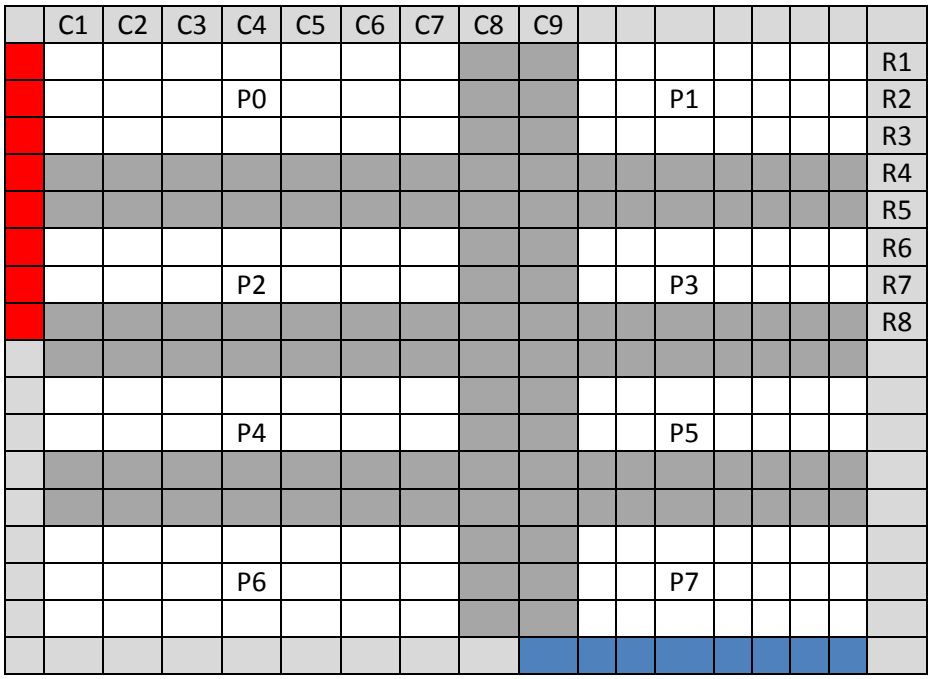
Now we are going to present some experimental values to substantiate the above arguments.

Runtimes (secs) for Row decomposition with N = 4320		
P=47	P=48	P=49
1.67518210	1.54780793	2.24802780

Runtimes (secs) for Row decomposition with N = 10080		
P=47	P=48	P=49
5.13776898	4.98632216	7.17153406

As we can see that for large values of N (4320 and 10080), the performance degrades if N is not divisible by P. Note that for both P=48 and P=49, the computation cost is nearly the same. The performance degrades because of enhanced communication cost.

Q. In this assignment, each process was responsible for computing the updates for a contiguous block of rows in the 2-D space. Describe a 2-D decomposition, where each processor takes a rectangular tile rather than a set of rows. What changes would be required in your code?
Answer.



The idea is to divide the rows into r equal chunks of size n/r and within each chunk divide the number of columns in c equal chunks of size n/c. Obviously the number of process involved will be P = r*c. For brevity of explanation, let us assume that n is both divisible by r and c. Obviously the problem is given an arbitrary number of processors and a grid size, to find an optimal value of r and c. We will be discussing this later.

The figure above demonstrates this with r = 4 and c = 2. The process with rank 0 will do the computation of the rectangular tile of size 4*8 (Rows 1-4 and columns 1-8) and process 1 will own the adjacent rectangular tile and so-on.

The extreme boundaries are marked with light grey where the red and blue layers are the hot and cold layers respectively. The dark grey layers are the ghost ”halo” layers which the processes need to communicate between the neighboring processes. For example process P0 will send the row R4C1-R4C8 to process P2 and receive row R5C1-R5C8 from process P2. Also it has to send column R1C8-R4C8 to P1 and receive R1C9-R4C9 from P1.

Code modifications required

- Given an arbitrary number of processes P and a grid size N, to find a good value of r and c.
    - We know that communication cost for exchanging the columns is higher than that of rows. This is because before sending the column layer to neighboring process it is copied in a send buffer and those accesses are n-strided. Also after receiving a column in a local receive buffer, a process has to be copied that to its ghost column and those accesses are

also n-strided. So we need to make sure that the width n/r must be short in order to avoid strided access. Also to get the benefit of cache spatial location we would like to have n/c as long as possible. One of the optimal options will be r = n/2 and t=2. With this the communication cost will be the minimum and still we have the rectangular decomposition.

- The tile size assigned to each processor is (n/r)*(n/c). Irrespective of the value we choose for r and c, the cost of computation will be the n^2/P, but the communication cost becomes max (n/r, n/c) (= n/c, as we are using rectangular tiles). This is lesser that the communication cost of n that we get with 2D-row decomposition. Hence the computation/communication ration becomes higher and leads to better performance.

- Set up communicators so that every processor in the same row is in a given communicator. This will allow fast exchange of ghost layers.
- Set up communicators so that every processor in the same column is in a given communicator. This will allow fast exchange of ghost layers.

Q. Discuss the results you obtained. Were you able to gain a linear or near-linear speedup? Why or why not?

Answer.

Discussion on results

- For N = 2304, the peak speedup is at P = 24 and the performance degrades after that. But with other greater values of N, the performance is nearly linear.
  - As we know the cost of computation is (n^2/p) and communication cost is n. Also the computation/communication ratio is n/p, which is a crucial governing factor of the performance. In this particular case of N=2304, the communication cost involved when P > 24, is much greater than the computation gain by sharing the workload among processors and as a result the performance degrades after P = 24.
    But for larger values of n (keeping the p constant to say 48), the computation/communication ratio (which is n/p) is much higher and as a result the performance increases.

- For large values of N (4320, 10080), the speedup increases with value of P.
  - With more number of P, the workload per process (N/P) decreases leading to lesser time for parallel computation.
- With very large values of "number of cores", P; the performance curve breaks the linear behavior.
  - With large number of P, the MPI communication cost becomes more which supersedes the performance gain on computation.

Linear or near linear speedup

- The speedup that we obtain **is near linear.**
  - With initial incremental values of P, the speedup is increasing proportionately as the workload is distributed proportionately among the processes. But with larger values of P, the communication cost outweighs the gain with workload

distribution and as a result the speedup curve started degrading at values where the computation/ communication ratios are low.

Column Decomposition

Overview of the Implementation

While implementing, we assumed that the grid length N is evenly divisible by the number of processors P.

Each process is responsible for the boundary value updation and grid computation of N/ P number of adjacent columns. During the communication phase, processes exchange their boundary columns to the neighboring processes (for example, a process with rank r will send its rightmost-most column and leftmost column to process with rank r+1 and r-1 respectively and receive the leftmost column from process r+1 and right-most column from rank r-1. Obviously for process with rank 0 and P-1, there will be only one send and receive because they have only one neighbor instead of two. Moreover, during each send the process will first copy the entire column in a local send buffer and then send that buffer, and for receives, the processes will receive the entire column in a local receive buffer and then copy that to the corresponding column.

Experimentation data for Column Decomposition

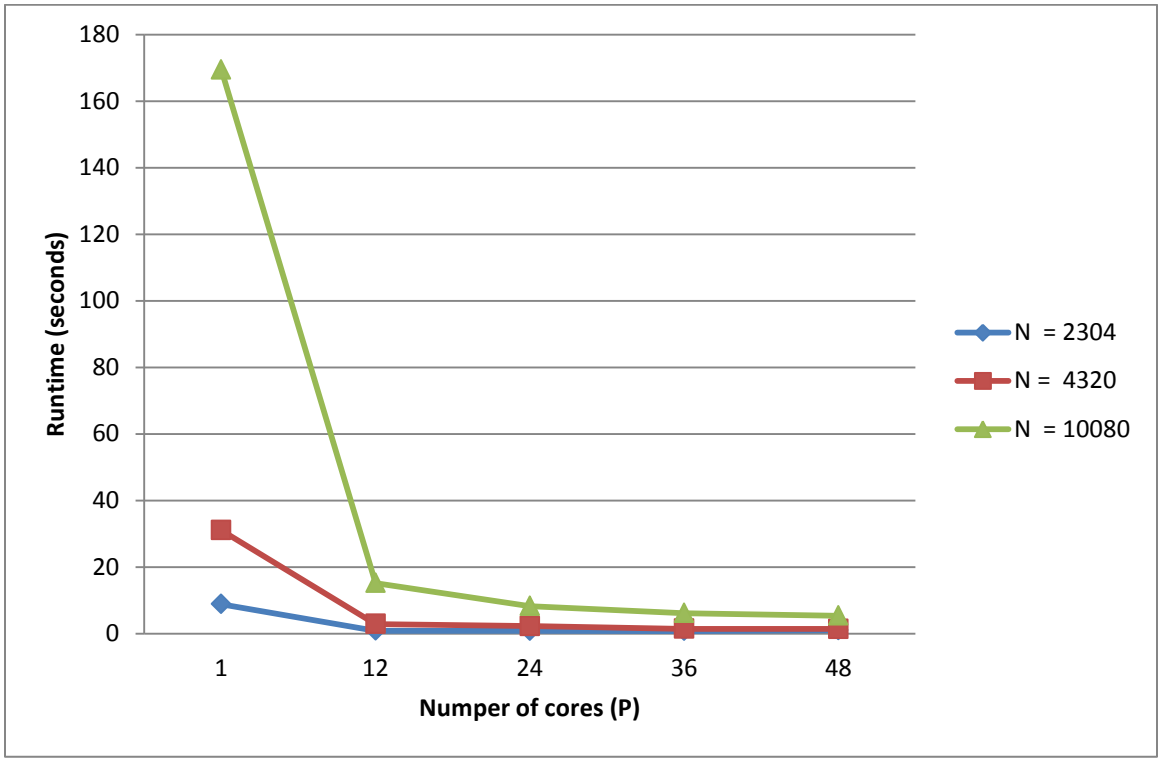
Tabular Representation

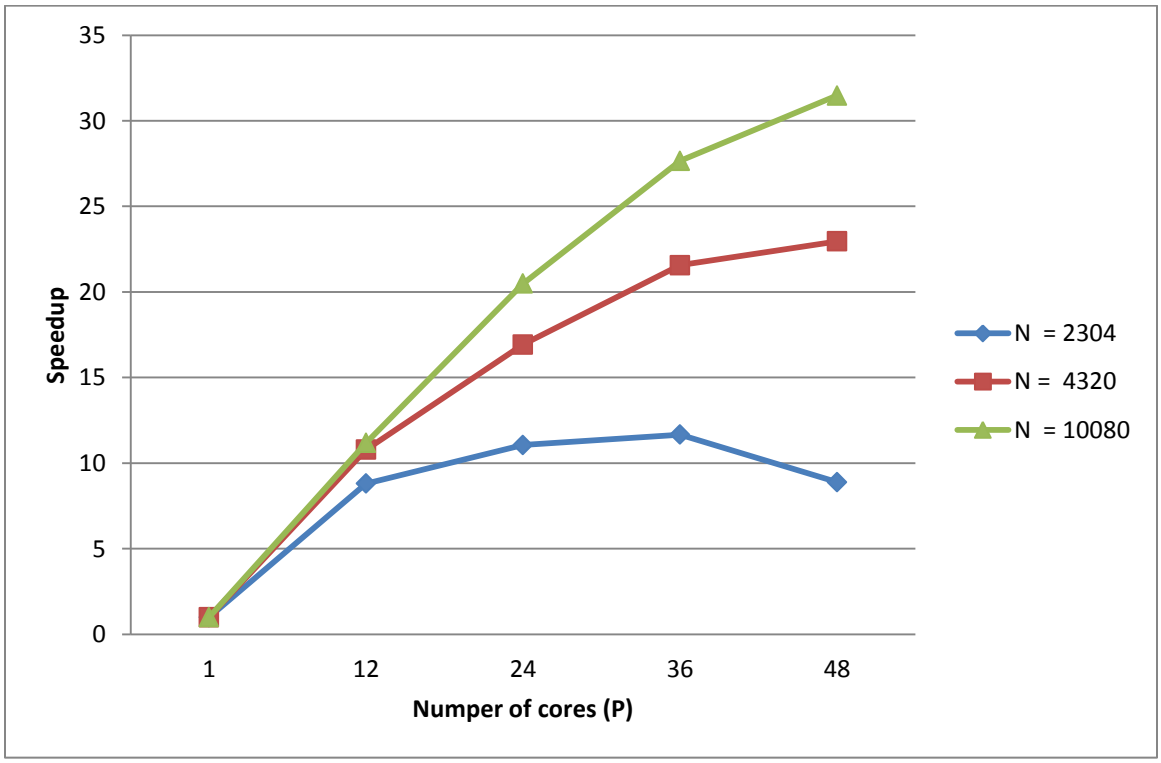
N = Grid length
(R,C) = A single floating-point value for the temperature at cell (R,C)
Tolerance = The value of *gmaxdiff* in the code, which gives the maximum change for any point below the threshold value,
Iteration = The maximum number of iterations reached.

Note: The speedup is calculated using (runtime for P=1 / runtime for P=p) for a given value of N.

P	Runtime (seconds)	Speedup	Iteration	Tolerance	(R,C)
Row Decomposition with (N,R,C) = (2304,2302,2300) and I_{max} = 100					
1	8.80117202	1	100	0.0012070087	0.18192301
12	0.84043694	8.8011	100	0.0012070087	0.18192301
24	0.79606009	11.0559	100	0.0012070087	0.18192301
36	0.75457191	11.663	100	0.0012070087	0.18192301
48	0.99012899	8.8889	100	0.0012070087	0.18192301
Row Decomposition with (N,R,C) = (4320,4318,4316) and I_{max} = 100					
1	31.04929781	1	100	0.0012070087	0.18192301
12	2.88022304	10.780	100	0.0012070087	0.18192301
24	1.83419764	16.928	100	0.0012070087	0.18192301
36	1.44023800	21.558	100	0.0012070087	0.18192301
48	1.35232806	22.959	100	0.0012070087	0.18192301
Row Decomposition with (N,R,C) = (10080,10078,10076) and I_{max} = 100					
1	169.47294307	1	100	0.0012070087	0.18192301
12	15.16023088	11.178	100	0.0012070087	0.18192301
24	8.27069592	20.490	100	0.0012070087	0.18192301
36	6.12829590	27.6541	100	0.0012070087	0.18192301
48	5.38506889	31.470	100	0.0012070087	0.18192301

Graphical Representation





Comment on the differences in efficiency and code complexity between the two implementations.

Efficiency

This implementation is less efficient than that of row decomposition. This is because before sending the column layers to neighboring process it is copied in a send buffer and those accesses are n-strided cache accesses. Also after receiving a column in a local receive buffer, a process has to be copied that to its ghost column and those accesses are also n-strided.

Code Complexity

The column-decomposition code is mostly similar to that of row-decomposition except during ghost layer exchanging (exchanging boundary columns between adjacent processes), where the ghost layer data need to be copied to a local send/ receive buffer to facilitate sending a single message rather than sending/ receiving one message per column element. This add to a slight complexity to the code.

Analysis

Q. If the grid size were not divisible by the number of cores, how would you need to modify your program? How would this impact performance?

Answer.

Let suppose that we have n as grid size (say number of columns) and num_processors as the number of cores and remaining_work (> 0 and < num_processors) is the remainder of n/ num_processors.

We have to distribute remaining_work columns, one to each of num_processors cores. In this way there will be cores (= remaining_work) having (N/ P + 1) columns and some (= num_processors – remaining_work) having N/ P columns.

Code modification:

```
125     //Partitioning the columns among the diffrent processes
126     process_boundary *proc_boundary = (process_boundary *) malloc(numprocessors*sizeof(process_boundary));
127
128     int clmn_partition = n/numprocessors;
129     int remaining_work = n%numprocessors;
130
131     int actual_work;
132     for(j=0; j<numprocessors; j++) {
133         actual_work = clmn_partition;
134         if(0 != remaining_work) {
135             actual_work += 1;
136             remaining_work --;
137         }
138         proc_boundary[j].j_first = actual_work*j + 1 ;
139         proc_boundary[j].j_last  = proc_boundary[j].j_first + actual_work;
140     }
```

In this way, the work will be distributed 'nearly equal' among the cores.

Impact of computation

Same as described before.

Q. Discuss the results you obtained. Were you able to gain a linear or near-linear speedup? Why or why not?
Answer.

Discussion on results

- For N = 2304, the peak speedup is at P = 36 and the performance degrades after that. But with other greater values of N, the performance is nearly linear.
    - As we know the cost of computation is (n^2/p) and communication cost is n. Also the computation/communication ratio is n/p, which is a crucial governing factor of the performance. In this particular case of N=2304, the communication cost involved when P > 36, is much greater than the computation gain by sharing the workload among processors and as a result the performance degrades after P = 36.
      But for larger values of n (keeping the p constant to say 48), the computation/communication ratio (which is n/p) is much higher and as a result the performance increases.

- For large values of N (4320, 10080), the speedup increases with value of P.
    - With more number of P, the workload per process (N/P) decreases leading to lesser time for parallel computation.
- With very large values of "number of cores", P; the performance curve breaks the linear behavior.
    - With large number of P, the MPI communication cost becomes more which supersedes the performance gain on computation.

Linear or near linear speedup

- The speedup that we obtain is near linear.
    - With initial incremental values of P, the speedup is increasing proportionately as the workload is distributed proportionately among the processes. But with larger values of P, the communication cost outweighs the gain with workload distribution and as a result the speedup curve started degrading at values where the computation/communication ratios are low.