

# 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 Algorithm and suggested optimization

1. Initiate sends with MPI\_Isend.
2. Initiate receives with MPI\_Irecv.
- 3. Perform Jacobi grid computation on the interior of each block (owned by each process)**
4. Wait for communication to complete with MPI\_Waitall
- 5. Perform Jacobi grid computation on the perimeter of each block (owned by each process)**

The operation marked green is the improvement that we are proposing by overlapping computations with communication by doing some useful computation while messages are exchanged between adjacent processes.

Note: 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)

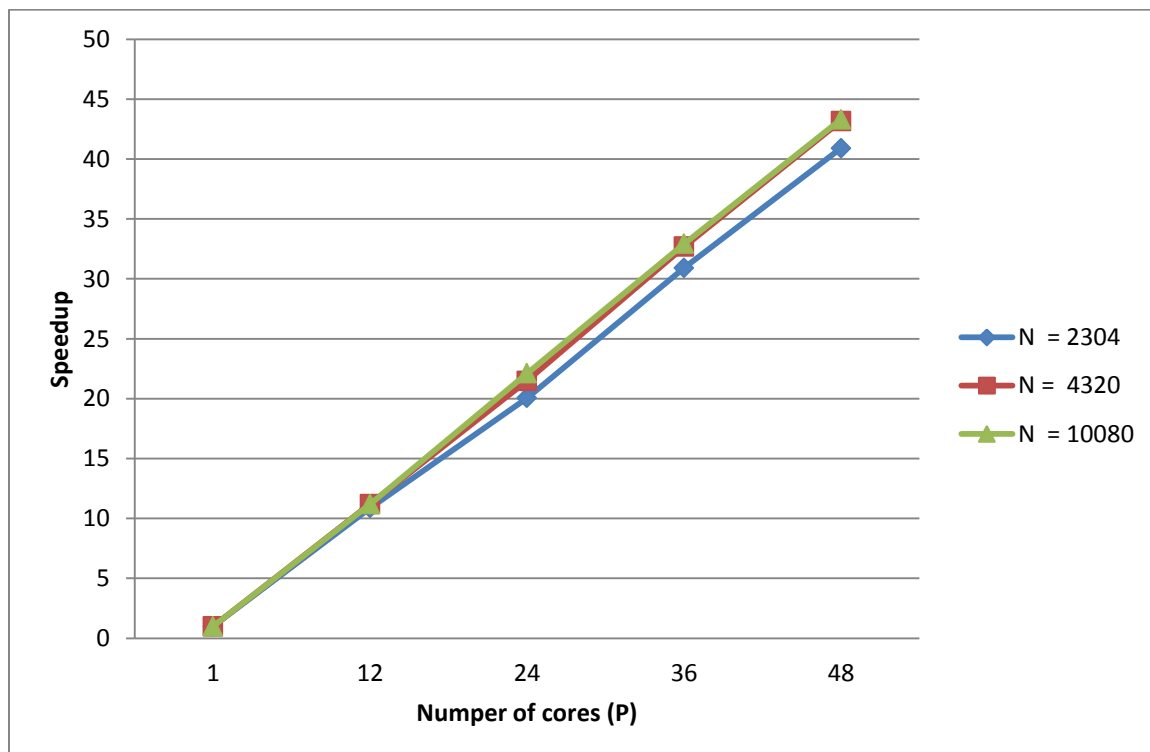
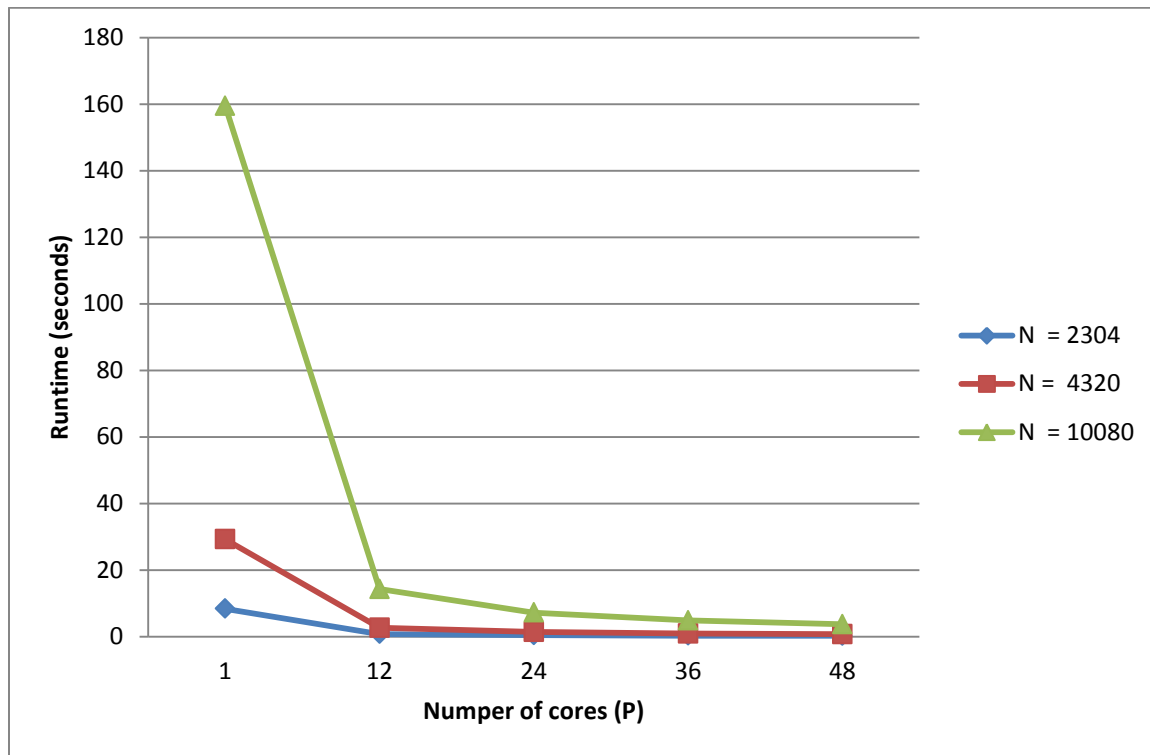
Tolerance = The value of *gmaxdiff* in the code, which gives the maximum change for any point below the threshold value,

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)
<b>Row Decomposition with (N,R,C) - (2304,2302,2300) and <math>I_{\max} = 100</math></b>					
1	8.32334805	1.000	100	0.0012070087	0.18192301
12	0.76641703	10.860	100	0.0012070087	0.18192301
24	0.41500688	20.055	100	0.0012070087	0.18192301
36	0.26932043	30.905	100	0.0012070087	0.18192301
48	0.20348991	40.903	100	0.0012070087	0.18192301
<b>Row Decomposition with (N,R,C) - (4320,4318,4316) and <math>I_{\max} = 100</math></b>					
1	29.22814989	1.000	100	0.0012070087	0.18192301
12	2.60950494	11.200	100	0.0012070087	0.18192301
24	1.36022902	21.487	100	0.0012070087	0.18192301
36	0.89434981	32.680	100	0.0012070087	0.18192301
48	0.67701101	43.172	100	0.0012070087	0.18192301
<b>Row Decomposition with (N,R,C) - (10080,10078,10076) and <math>I_{\max} = 100</math></b>					
1	159.39944911	1.000	100	0.0012070087	0.18192301
12	14.24454880	11.190	100	0.0012070087	0.18192301
24	7.20806217	22.114	100	0.0012070087	0.18192301
36	4.84199810	32.920	100	0.0012070087	0.18192301
48	3.68053699	43.308	100	0.0012070087	0.18192301

## Graphical Representation



### Analysis

**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 a particular value of  $P$ , the speedup values are **nearly** the same for all  $N$ s.
  - We know that for a particular value of  $P$ , the computation cost will increase with increasing values of  $N$ , but the communication cost will remain almost the same ( $\sim P-1$ ). As in our implementation we have overlapped the inner grid computation (which is majority of the computation cost) with communication, so the computation cost becomes roughly equal to the communication cost (which is same for all  $N$ s with a particular value of  $P$ ) and as the result the performance is also nearly the same.
- With very large values of  $P$ , the performance curve breaks the linear behavior.
  - Let take the example of  $N=10080$ . Had the curve been linear, then the expected value of speedup at  $P = 48$  is 43.89, but we got 43.308. Similarly, for  $N = 4320$ , the obtained value of speedup which is 43.172 at  $P = 48$  degrades from the expected value of 43.57 (which should be if the curve is linear). The same is true for  $N = 2304$ .

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 the examples mentioned above, the communication cost involved when  $P = 48$  is much greater than the computation gain by sharing the workload among processors and as a result the performance degrades after  $P = 48$ .

- The speedup increases with value of  $P$ .
  - With more number of  $P$ , the workload per process ( $N/P$ ) decreases leading to better overall performance (or lesser runtime).

### 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.

**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  $< \text{num\_processors}$ ) is the remainder of  $n / \text{num\_processors}$ .

We have to distribute **remaining\_work** rows, one to each of **num\_processors** cores. In this way there will be cores ( $= \text{remaining\_work}$ ) having  $(N / P + 1)$  rows and some ( $= \text{num\_processors} - \text{remaining\_work}$ ) having  $N / P$  rows.

### Code modification:

```
120
121 //Partitioning the rows among the different 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 on computation

This will lead to load imbalance as there will be  $(\text{num\_processors} - \text{remaining\_work})$  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  $\text{num\_processors}$ .

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 of the four cores will do the parallel computation for a block of 4 rows each, whereas in the later case,  $(P_0, P_1, P_2)$  will do the parallel computation for blocks containing (6,5,5) rows. Obviously  $P_0$  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 of the four cores will do the parallel computation for a block of 4 rows each, where as in the later case,  $(P_0, P_1, P_2, P_3, P_4)$  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  $P_0$  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 (for Impact on computation), 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 ( $\sim (n^2/P)*I$ , where  $I$  is the number of iterations) will very high as compared to communication cost ( $\sim n*I$ ).

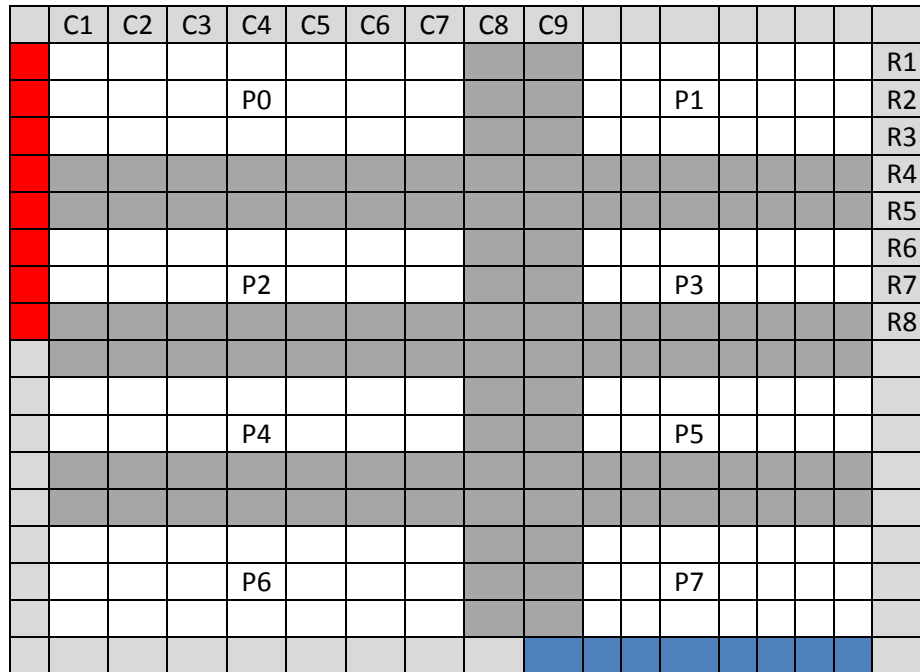
Now we are going to present some experimental values to substantiate the above arguments.

Runtimes (secs) for Row decomposition with $N = 4320$		
$P=47$	$P=48$	$P=49$
.73272445	0.67701101	0.98328710

Runtimes (secs) for Row decomposition with $N = 10080$		
$P=47$	$P=48$	$P=49$
3.79232391	3.68053699	5.29350000

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 in  $P = 49$  because of enhanced communication cost.

**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  $c$ . The obvious 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  $4 \times 8$  (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 locality we would like to have  $n/c$  as long as possible. One of the 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. But with a large computation cost. So the optimal value will be the one with highest value of computation/communication cost.
  - 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 than the communication cost of  $n$  that we get with 2D-row decomposition. Hence with rectangular 2D-decomposition, the computation/communication ratio becomes higher and leads to better performance.
- For fast exchange of ghost layers,
  - Set up communicators so that every processor in the same row is in a given communicator.
  - Set up communicators so that every processor in the same column is in a given communicator.



## 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.**

### Overview of the Algorithm and suggested optimization

1. Initiate sends with `MPI_Isend`.
2. Initiate receives with `MPI_Irecv`.
- 3. Perform Jacobi grid computation on the interior of each block (owned by each process)**
4. Wait for communication to complete with `MPI_Waitall`
- 5. Perform Jacobi grid computation on the perimeter of each block (owned by each process)**

The operation marked green is the improvement that we are proposing by overlapping computations with communication by doing some useful computation while messages are exchanged between adjacent processes.

## Experimentation data for Column Decomposition

### Tabular Representation

$N$  = Grid length

$(R,C)$  = A single floating-point value for the temperature at cell  $(R,C)$

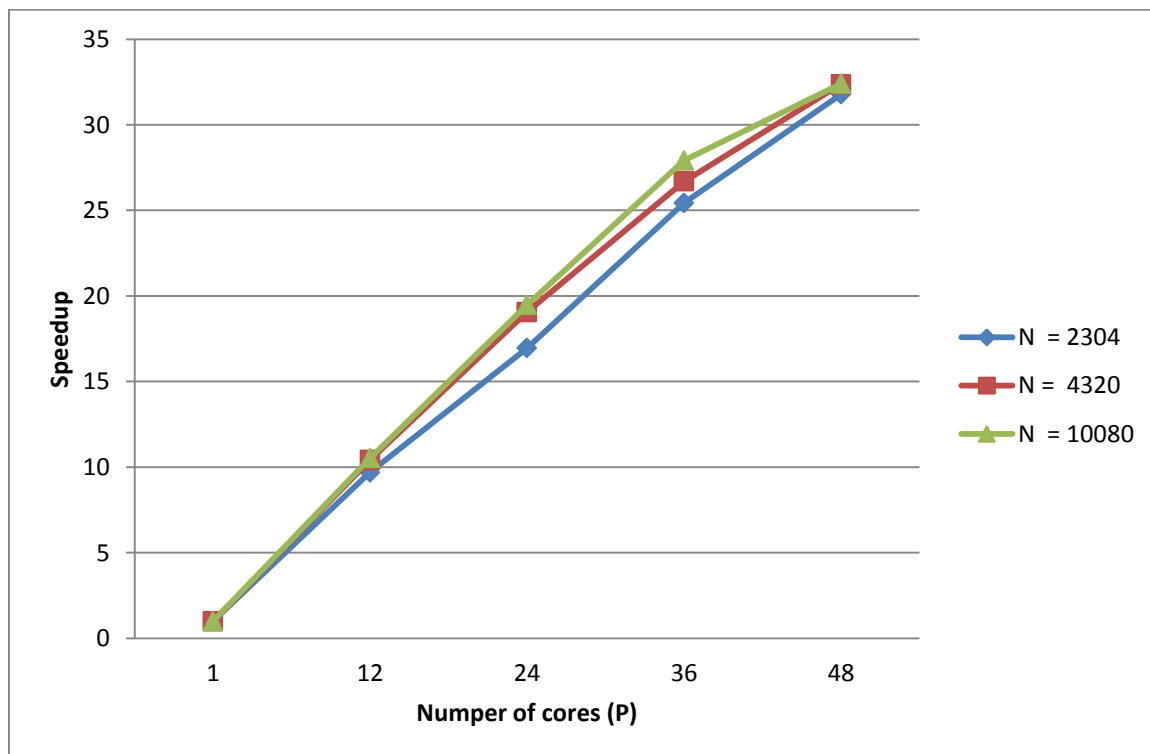
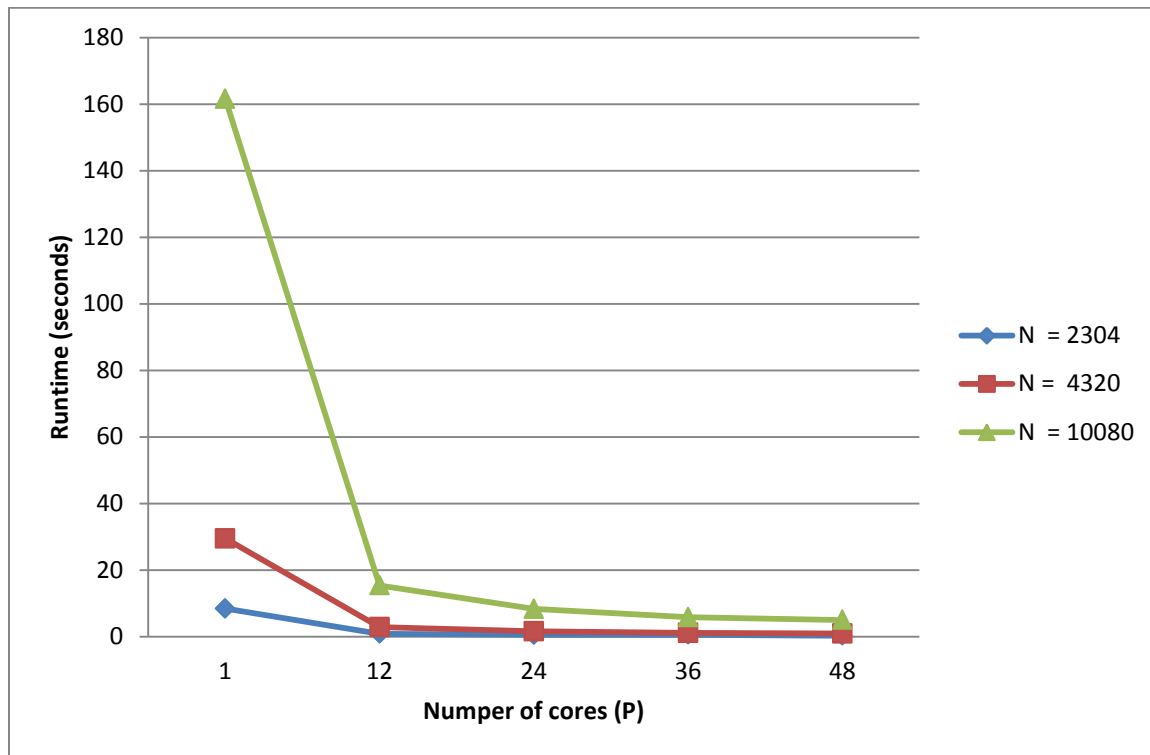
Tolerance = The value of *gmaxdiff* in the code, which gives the maximum change for any point below the threshold value,

Iteration = The maximum number of iterations reached.

**Note: The speedup is calculated using  $(\text{runtime for } P=1 / \text{runtime for } P=p)$  for a given value of  $N$ .**

P	Runtime (seconds)	Speedup	Iteration	Tolerance	(R,C)
Column Decomposition with (N,R,C) - (2304,2302,2300) and $I_{\max} = 100$					
1	8.38472390	1.000	100	0.0012070087	0.18192301
12	0.86562300	9.686	100	0.0012070087	0.18192301
24	0.49459815	16.952	100	0.0012070087	0.18192301
36	0.32974374	25.428	100	0.0012070087	0.18192301
48	0.26381898	31.782	100	0.0012070087	0.18192301
Column Decomposition with (N,R,C) - (4320,4318,4316) and $I_{\max} = 100$					
1	29.44589710	1.000	100	0.0012070087	0.18192301
12	2.82891083	10.408	100	0.0012070087	0.18192301
24	1.54481506	19.061	100	0.0012070087	0.18192301
36	1.10320687	26.691	100	0.0012070087	0.18192301
48	0.90961006	32.372	100	0.0012070087	0.18192301
Column Decomposition with (N,R,C) - (10080,10078,10076) and $I_{\max} = 100$					
1	161.55549502	1.000	100	0.0012070087	0.18192301
12	15.34859300	10.525	100	0.0012070087	0.18192301
24	8.29507518	19.476	100	0.0012070087	0.18192301
36	5.78556991	27.923	100	0.0012070087	0.18192301
48	4.98713112	32.394	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, which is also confirmed by the lesser speedup gains than the 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 adds to a slight complexity to the code.

#### Analysis

**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 a particular value of P, the speedup values are **nearly** the same for all Ns.
  - Same reason as for row-decomposition.
- With very large values of P, the performance curve breaks the linear behavior.
  - Same reason as for row-decomposition.
- The speedup increases with value of P.
  - Same reason as for row-decomposition.

#### 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.

**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  $\text{num\_processors}$  as the number of cores and  $\text{remaining\_work}$  ( $> 0$  and  $< \text{num\_processors}$ ) is the remainder of  $n / \text{num\_processors}$ .

We have to distribute  $\text{remaining\_work}$  columns, one to each of  $\text{num\_processors}$  cores. In this way there will be cores ( $= \text{remaining\_work}$ ) having  $(N / P + 1)$  columns and some ( $= \text{num\_processors} - \text{remaining\_work}$ ) having  $N / P$  columns.

**Code modification:**

```
125 //Partitioning the columns among the different 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.