TÉCNICO LISBOA

Parallel and Distributed Computing

1º Semester – Academic Year 2013/2014

Stefan Schneider, 79791 - stefan.schneider@ist.utl.pt
Kuganesan Srijeyanthan, 79531 - kuganesan.srijeyanthan@ist.utl.pt
Gayana Ranganatha Chandrasekara Pilana Withanage, 79529 - gayana.withanage@ist.utl.pt
Group 40 - Alameda

Introduction

This report is presented in order to elaborate and describe the serial and parallel programing aspects followed during the implementation in order to optimize the flow of the program. The quantitative results and comparisons are also included. The objective of this project is to implement the given problem with a program which executes serially and convert that program into a parallel one with OpenMP such that we get the same final results with optimized time efficiency. When you follow the report you will encounter section 1 with serial programing methodology followed by parallel programing methodology in section 2. Section 3 will give the results and comparisons. Finally the section 4, contains the conclusion with result justification. In this updated report we included Section 5 to discuss about the OpenMPI implementation.

Section 1: Serial Programing Methodology

There are two possible ways of implementation identified, and below Figure 1 and Figure 2 describes both of them. For our implementation we have selected the method in Figure 2 and the reason for the selection has been described in the next Section.

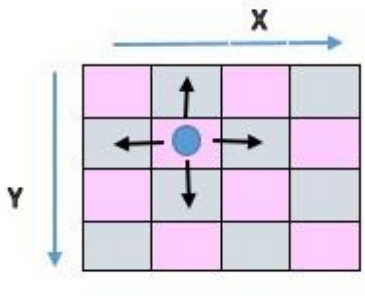


Figure 1: The processing cell (Red) checks adjacent cells and decide where it can moves to any of it. (Assume red cell contains a Wolf or a Squirrel) Processing occur to the X direction.

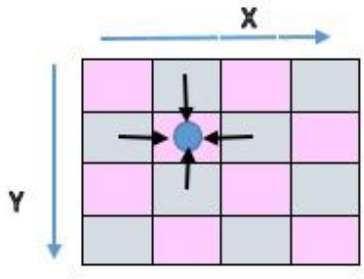


Figure 2: The processing cell (Red) checks adjacent cells and decide which of adjacent cells can come on to it. (Assume red cell is an empty cell) Processing occur to the X direction.

Section 2: Parallel Programing Methodology

In this case we are keeping two maps in the memory, where first map is the source map which will be initialized with the values which are given by input file. Meantime the second map is a temporary copy of the same data structure but it will be initially empty.

When we process the source map according to the way shown in Figure 2, the updates are made in the temporary copy in a certain sub generation (i.e.: Red sub generation). Once a sub generation is completed the changes made into the temporary map will be populated to the source map and temporary map will be cleaned again. Likewise the program executes for all the generations and gives the final result.

Methodology Selection Description

1. Parallel execution in method described in Figure 1.
   In the parallel execution, a race condition will be occurred when T1 (Thread 1) and T2 (Thread 2) both sees a movable position in the source map and tries to update the corresponding temporary map's memory location. The whole sub generation is parallelized so that there is a high probability of getting many thread conflicts which will lead to a poor performance of the parallelized version.

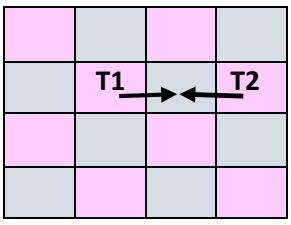


Figure 3

2. Parallel execution in method described in Figure 2.
   In this method, the processing cell and adjacent cells will be read from the source map and relevant updates will be applied to the corresponding cells in the temporary map. Due to the deterministic behavior of the active elements (wolfs and squirrels) this method of updating the temporary map do not encounter any race conditions. Therefore each FOR loops those are responsible for a certain sub generation can be parallelized without any dependencies and this method makes the parallel execution much efficient.

Load balancing

In order to analyze how load distribution among threads could affect overall execution time, we have profiled our parallel code by giving different chunk sizes to various scheduling mechanisms including row and column wise processing. The test has been conducted by choosing row wise processing of map because column wise processing did not contribute more influence to the parallel execution. Ex: High rate of cache misses, this rate will be increased when map size grows. For the experiment we used a map with size 30 and first half of the map was filled with elements (i.e. Wolves, Squirrels) while the latter half of the map is empty. Below table provides the total execution time of each thread within a sub generation (execT) and the time spent by each thread for its assigned load by parallel FOR (bodyT) in different scheduling configurations obtained from OpenMP `profiler`. According to the following statistics we could observe that the most effective load balanced configuration was Dynamic scheduling with chunk (size=1).

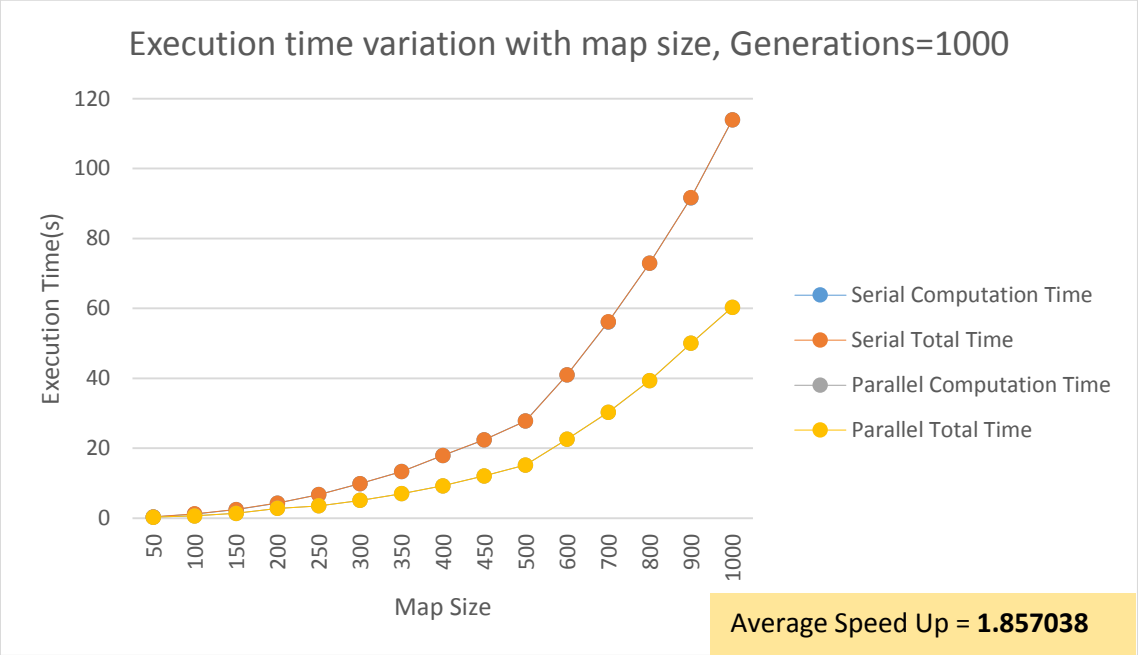
Scheduling Configuration	Sub Generation	T0 (bodyT)	T1 (bodyT)	T2 (bodyT)	T3 (bodyT)	T0,T1,T2,T4 execT
No configuration	Red	3.59	4.44	2.83	1.69	5.47
No configuration	Black	3.37	3.7	2.71	1.7	4.77
Static chunk (size=20)	Red	9.64	2.67	0.04	0.04	11.57
Static chunk (size=20)	Black	8.71	2.69	0.05	0.04	10.63
Static chunk (size=13)	Red	3.5	4.36	2.82	1.69	5.41
Static chunk (size=13)	Black	3.28	3.66	2.71	1.69	4.69
Static chunk (size=10)	Red	4.96	4.62	2.74	0.04	6.34
Static chunk (size=10)	Black	4.63	3.9	2.74	0.04	5.96
Static chunk (size=5)	Red	3.64	4.01	2.62	2.29	5.22
Static chunk (size=5)	Black	3.55	3.73	2.05	2.05	5.06
Static chunk (size=1)	Red	3.34	3.53	2.73	3.23	4.43
Static chunk (size=1)	Black	3.05	3.27	2.52	2.79	4.13
Dynamic chunk (size=20)	Red	2.9	3.67	1.7	3.2	9.29
Dynamic chunk (size=20)	Black	2.84	4.13	1.82	3.68	10.25
Dynamic chunk (size=13)	Red	4.01	2.2	2.9	2.6	5.47
Dynamic chunk (size=13)	Black	3.54	3	3.34	3.2	5.79
Dynamic chunk (size=10)	Red	2.72	3.13	2.97	3.05	4.38
Dynamic chunk (size=10)	Black	4.51	2.27	3.22	2.87	5.88
Dynamic chunk (size=5)	Red	2.97	3.15	3.39	3.41	4.64
Dynamic chunk (size=5)	Black	2.72	3.13	2.97	3.05	4.38
Dynamic chunk (size=1)	Red	3.39	3.4	3.4	3.39	4.19
Dynamic chunk (size=1)	Black	3.12	3.12	3.12	3.1	3.89

Flow of the parallel execution

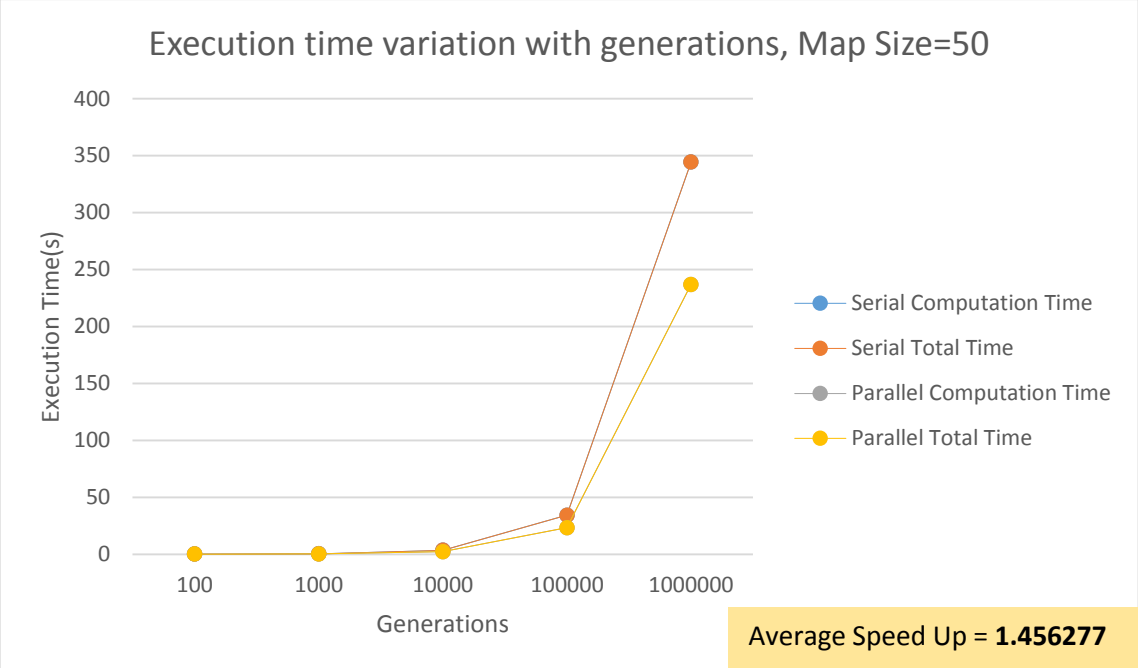
Section 3: Results

Below graphs show the execution time of each serial and parallel programs spent during the processing of the map. One graph was populated by keeping the generation count a constant and changing the map size, while the other graph was populated by keeping the map size constant and changing the number of generations.

- Test machine configuration is, 2 CPUs (4 Thread), 4GB Memory, x86_64 architecture.



Average Speed Up = **1.857038**



Average Speed Up = **1.456277**

Section 4: Conclusion

According the above results we can conclude that the parallel execution time has become almost half of the serial execution time when the number of iterations increase, which we were expecting to see in a 2 core CPU with OpenMP implementation. Moreover the load balancing with dynamic scheduling configuration with chunk size of 1 has provided the most efficient load distribution which was also an expected result, because of following two reasons

1. The parallel execution was free from race conditions which includes map creation and contiguous memory allocation.
2. Since generations are growing, the uncertainty (evolution of the agents across the map) of agents' movements.

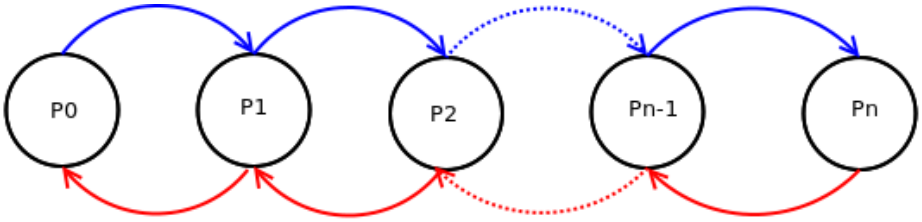
We have observed, the lower the chunk size, has produced significantly lower execution time over other chunk sizes which we have chosen and also the load was balanced uniformly among threads.

Note: In Section 3 Results, the average speed up is lower with fixed map size of 50, compared to the varied map size with fixed generations. The reason for this is, when the map size is lower, the number of blocks assigned for a thread at a time is smaller, and therefore more time is taken for thread management rather than the other scenario where each thread gets a larger block at once to process with higher map sizes.

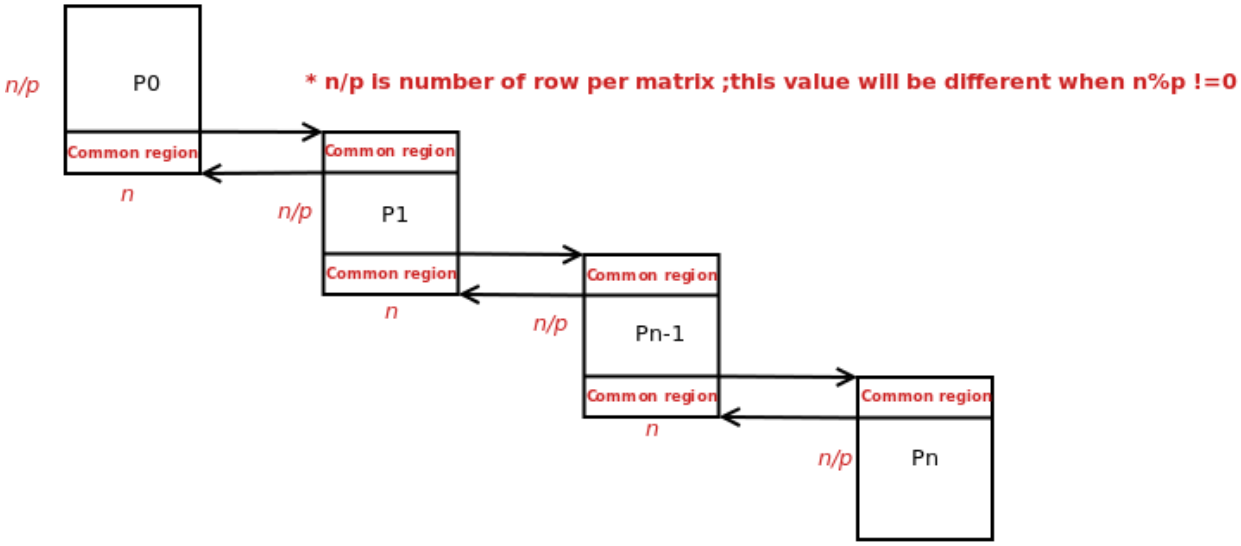
Section 5: OpenMPI Solution

Decomposition

Row-wise decomposition has been used in MPI parallel implementation even though there are several decomposition methods out there such as column-wise and checkerboard. Obviously, row-wise and column-wise methods are having less scalability than checkerboard method but, communication among neighbors will be significantly reduced in row-wise decomposition since middle process will be communicating its adjacent maximum two matrices.



Each processor will be interchanging information with neighbors only

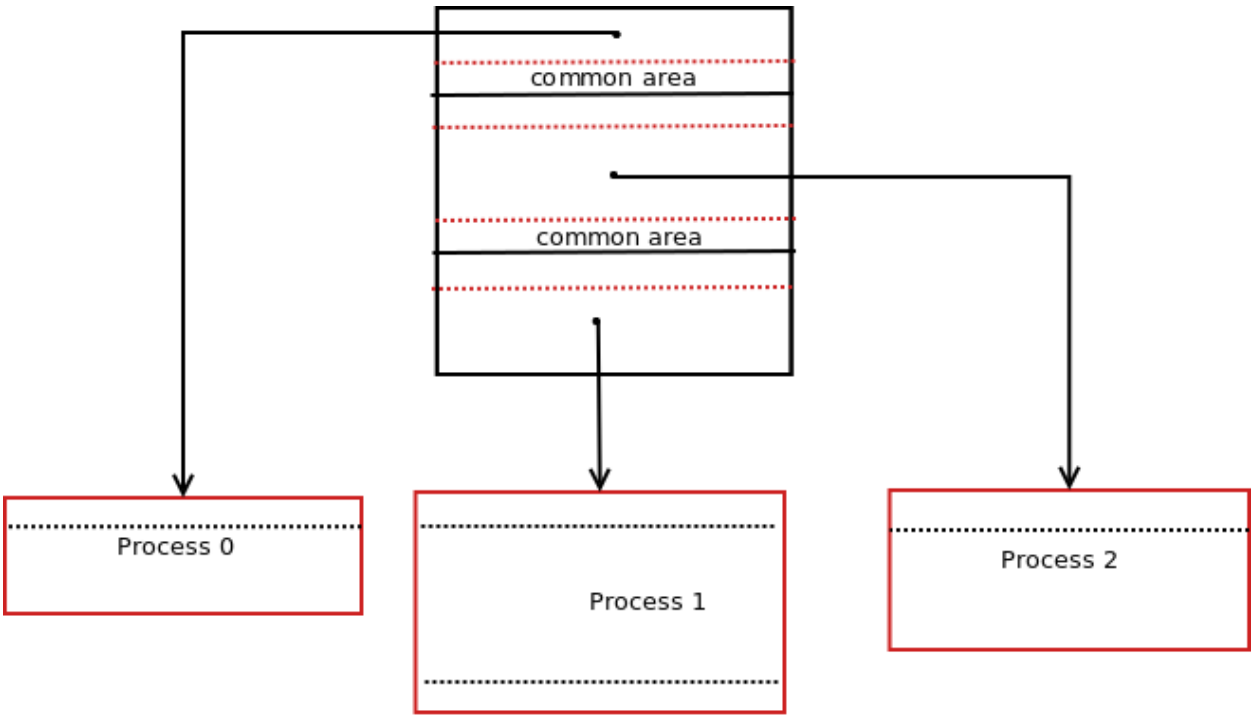


Matrix representation of interprocess communication

The above diagram shows the state of each process and which part of the matrix have to be interchanged with adjacent regions.
Every sub generation, individual process will exchange only updated cell with neighbors which will lead fast communication and allow other process to start the next sub generation processing.

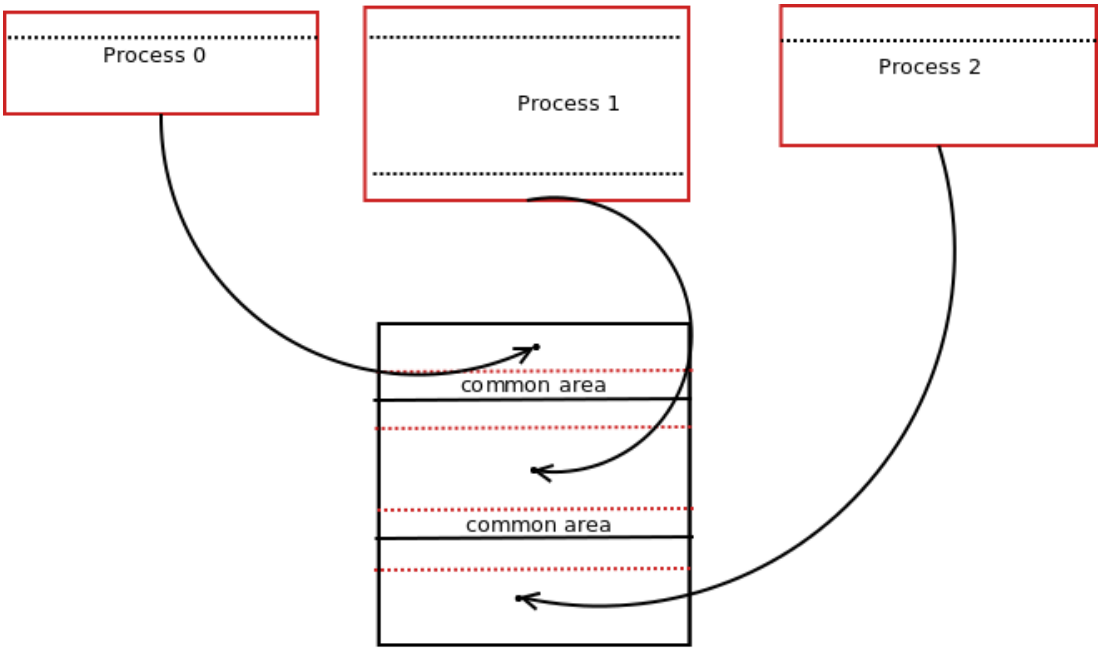
Load balancing

Basically, the load balancing is mainly depends on the number of rows per process, because an evaluation of the population is nondeterministic along with the generation. So, we have divided the total number of rows by process and reminder rows had equally added to the process. The following diagram shows how initial world is scattering and assigning to slaves nodes.



Distributing the work to slaves

Once the job has done by each process, every slaves will sent their part to master process for build up the final processed world. The below diagram shows the flow of information gathering from slaves.



Collecting work from slaves

Results and Conclusion

Execution environment - Rnl cluster

submitted condor job file

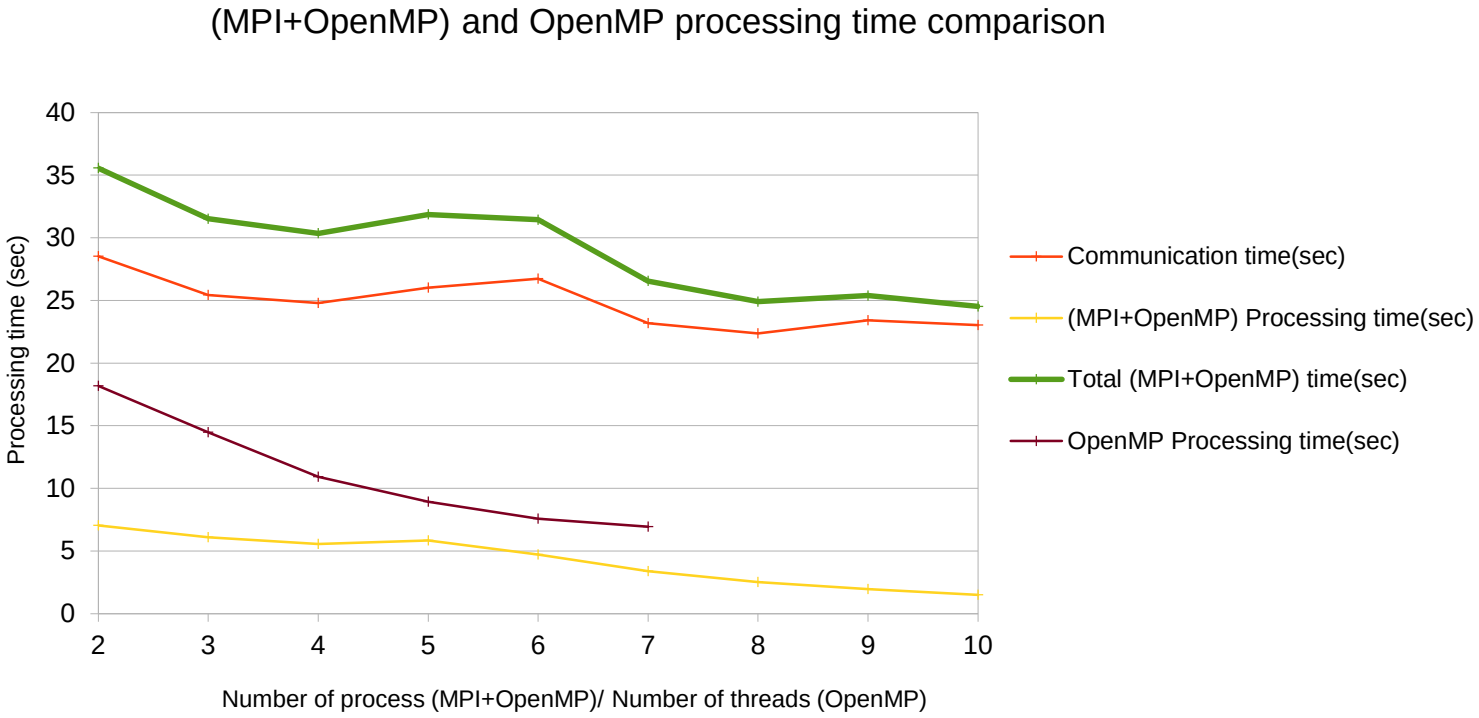
universe = parallel
executable = /usr/sbin/ompiwrapper.sh
arguments = wolves-squirrels-mpi world_1000.in 3 4 4 5000
log = /mnt/nimbus/users/3/1/ist179531/output/log
output =/mnt/nimbus/users/3/1/ist179531/output/out
error = /mnt/nimbus/users/3/1/ist179531/output/err
machine_count = 8
queue

In order to compare the performance of MPI implementation, we have tested along with OpenMP in some extent. OpenMP version has been executed only Rnl cluster's host machine which has maximum 8 cores, so, we have utilized up to 7 cores to get an execution time.

MPI version have submitted through condor job and results were obtained in two different ways such as communication time and processing time. Communication time shows how much average time have taken to interchange the information among neighbors for whole generation. Processing time shows how much time has taken to process the dedicated matrix in each processor.

Our implementation has a capability of spawning threads in host where MPI program instance is running. We have observed that most of the host machine are having four core processor and spawning more than two threads will be decreased the overall performance in terms of communication and processing. So, whenever condor submit the job to the host machine first our program check whether any other same MPI instances have been assigned by condor, if there is no same MPI instance is running , two more threads will be created to process the matrix ,otherwise it will be executing with one thread.

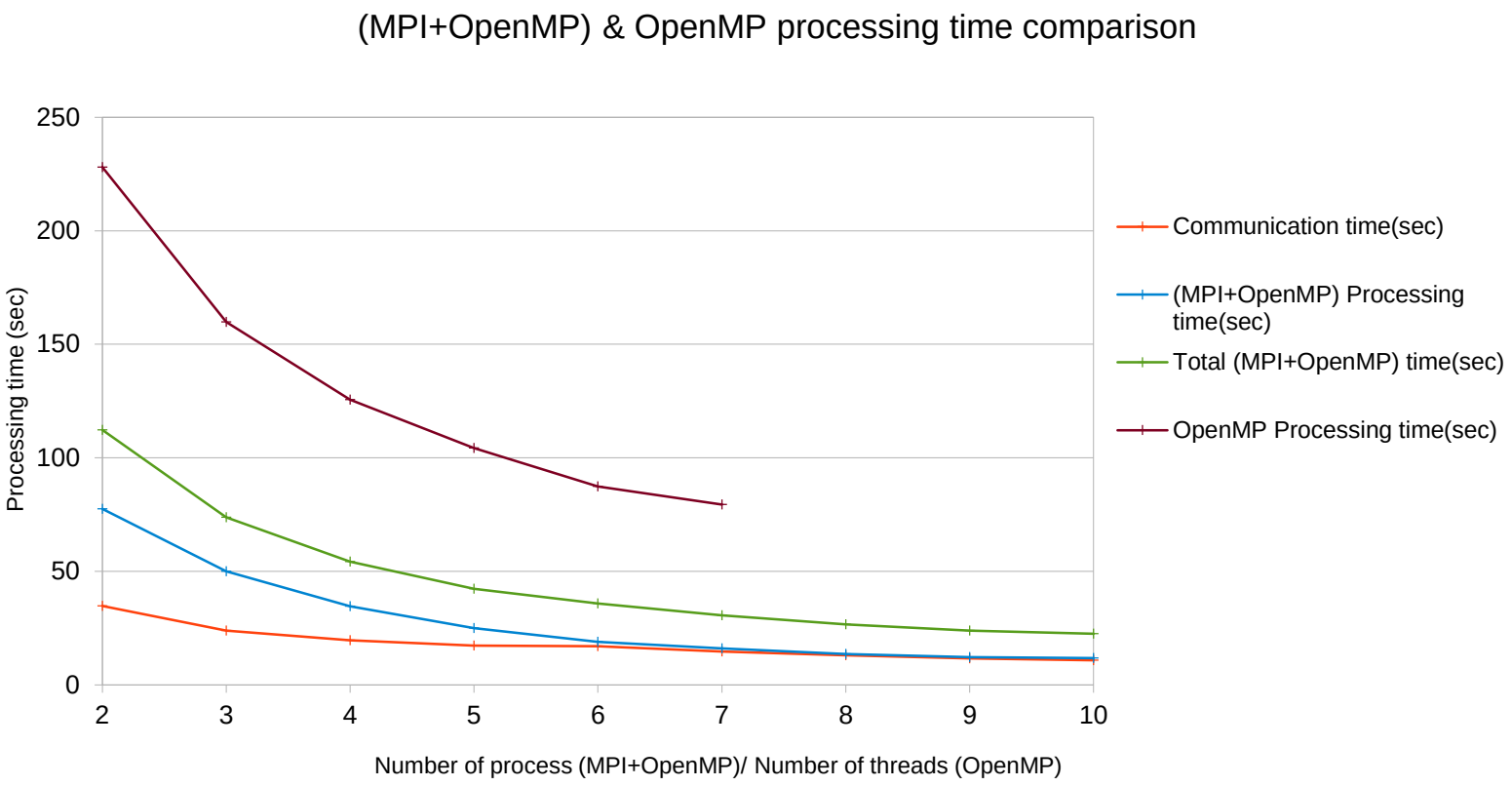
Case 1: Matrix size= 100*100 Iteration=50000



The above graph shows the processing time vs number of process/thread variation in the field of OpenMP and (MPI+OpenMP) implementation. OpenMP results were obtained by executing the program at same host with increasing number of threads.

MPI program has taken lower processing time since it has processed small chunk of matrix in every iteration. Total communication time was high when number of process is low and it has continuously decreased when number of process increased. Here we have obtained some spikes in the graph when process number was 5 and 6, because same MPI instances have been repeatedly submitted to same host and couldn't able to spawn more threads to process them. Another reason was communication time between processes has continuously fluctuated since we couldn't ensure the network traffic in between process. Since the size of the matrix (problem size) is relatively small, we clearly observe the huge different between communication time and processing time in any given number of process.

Case 2: Matrix size= 1000*1000 Iteration=5000



The above graph shows the results of 1000*1000 with 5000 generation in both OpenMP and MPI implementation. When the problem size increases the MPI performs better than OpenMP. Initially when number of process is low, communication time was too low and processing time was high and after some point both have converged and gave optimized results.