

# **Project Report**

## **CS6230: High Performance Parallel Computing**

### **Parallelized Logic Circuit Simulation**

**Georgia Institute of Technology**

Pankaj Chawla, MS CS (pankaj.chawla@gatech.edu)

Dhanik Shah, MS CS (dhanikshah@gatech.edu)

## **Title** - Parallelized Logic Circuit Simulation

**Participants** – Dhanik Shah, MS CS (dhanikshah@gatech.edu)  
Pankaj Chawla, MS CS (pankaj.chawla@gatech.edu)

**Goal** – Our primary goal was to implement the parallelized version of the logic circuit simulator and compare the performance of the serial version of the code with the parallelized version. Our secondary goal was to compare the performance of the MPI version of the code and OpenMP version of the code. We were able to meet both the goals. We learnt a lot from this project. The most important lesson learnt is that many factors need to be considered while parallelizing an application. One of the major factors is whether communication can be overlapped with computation.

**Methods** – We have implemented a serial version of the logic circuit simulator. We have also parallelized the code using MPI and OpenMP. Thus, we have

1. Implemented the serial version of logic circuit simulator and evaluated time to solution.
2. Implemented parallel version of the code using MPI and evaluated time to solution.
3. Implemented parallel version of the code using OpenMP and evaluated time to solution.
4. Found out the best version of the code for our dataset (with minimum time to solution).
5. Run the parallel versions of the code for different number of nodes/threads and analyzed their performance.
6. Repeated the above process for input files of various sizes.

Now we describe the details of each version of the code:

Serial code Implementation – The process runs in a loop where it continues to evaluate whatever gates it can in each iteration until it has evaluated all the gates.

MPI Implementation - The problem with MPI version was that if we do a regular division of work; something like all nodes evaluating equal number of gates which would be equal to total number of gates divided by the number of processors, it would be difficult for a node to know the node to which it should send a request for a particular net (net is the numbered value given to every connection in the circuit). The nodes would either have to parse the input file and find out about the node or store this information about each net. Hence a different implementation had to be thought of.

Here is how we divided the work among processors. Suppose that there are  $n$  gates and  $m$  processors. Each node evaluates a gate only if this condition holds:

$$n \% m == \text{rank of the node}$$

We analyzed the work distribution for this formula on input files and found out that it does a nearly equal distribution of work and simplifies the problem of requesting nets because each node now knows, by this formula, the node which is evaluating that net.

Every node iterates through its gates, evaluates whatever gate it can, and sends a request for others. Each node periodically probes for any incoming messages, fulfills the requests if it can and queues the requests which it can't, for a later time. Also every node checks its queue to see if it can fulfill any queued request whenever it evaluates a new gate. All the nodes send a done signal to node 0 after they are done evaluating

their gates. The node 0, after receiving done signal from all the nodes, signals them to go ahead and output their values. After this all the nodes exit the program.

OpenMP Implementation – In OpenMP we used the simple static scheduling policy as it gave us the best results. Every thread evaluates equal number of gates. In this program there are two nested loops. The outer loop iterates until all gates have been evaluated, the inner loop parallelizes the work. It creates threads and divided the work equally among them. The nodes evaluate whatever they can and terminate. They do not wait for other threads. This process goes on until all gates have been evaluated.

**Metrics** – We have used “time to solution” as a metric for performance evaluation. Scalability with respect to number of nodes/threads and with respect to input file size has also been analyzed.

## **Experimental Setup –**

Hardware(Cluster): Warp cluster nodes

Software: Open MPI 1.1.2 and OpenMP (programmed in C)

Compiler: GNU C compiler (gcc) for serial version and MPI version of the code  
Intel C compiler (icc) for OpenMP version of the code

Data sets: Input Files (bench format) have been downloaded from <http://www.fm.vslib.cz/~kes/asic/iscas/> and test vectors have been generated using serial version of the code. Smallest file has 120 gates and the largest file has 2636 gates. The files have been formatted a little to embed the inputs.

**Issues** – Data dependency and load balancing were the two main issues that we had to deal with. Minimizing the communication overhead in conjunction with load balancing in MPI version of the code was a big challenge.

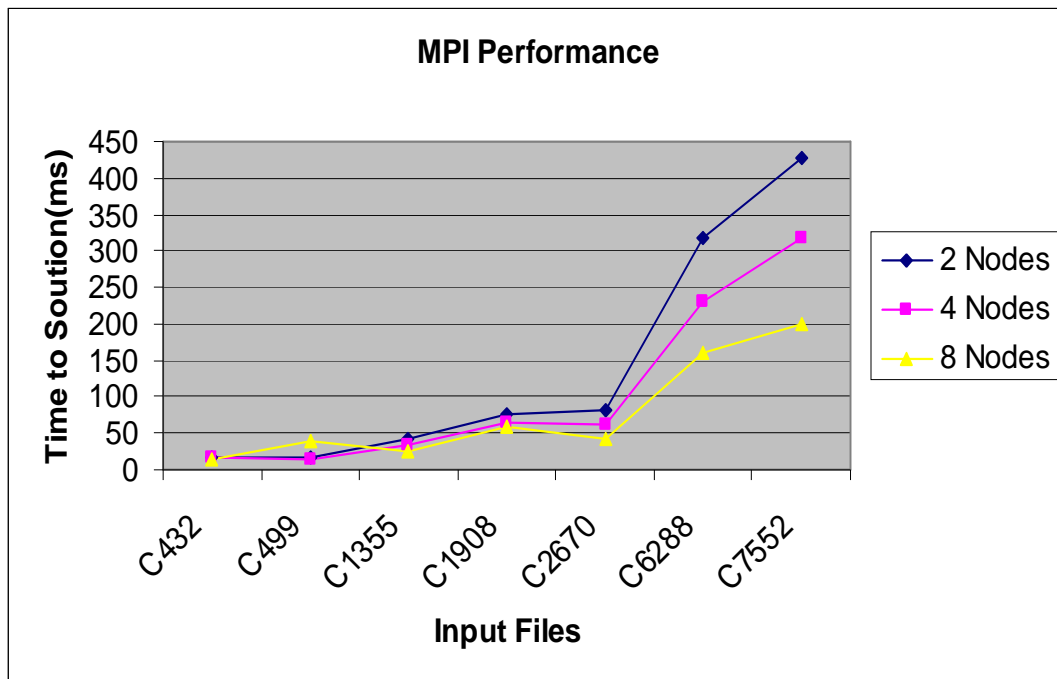
## **Results –**

Serial Version:

It always took less time to execute than both the MPI and the OpenMP versions of the code.

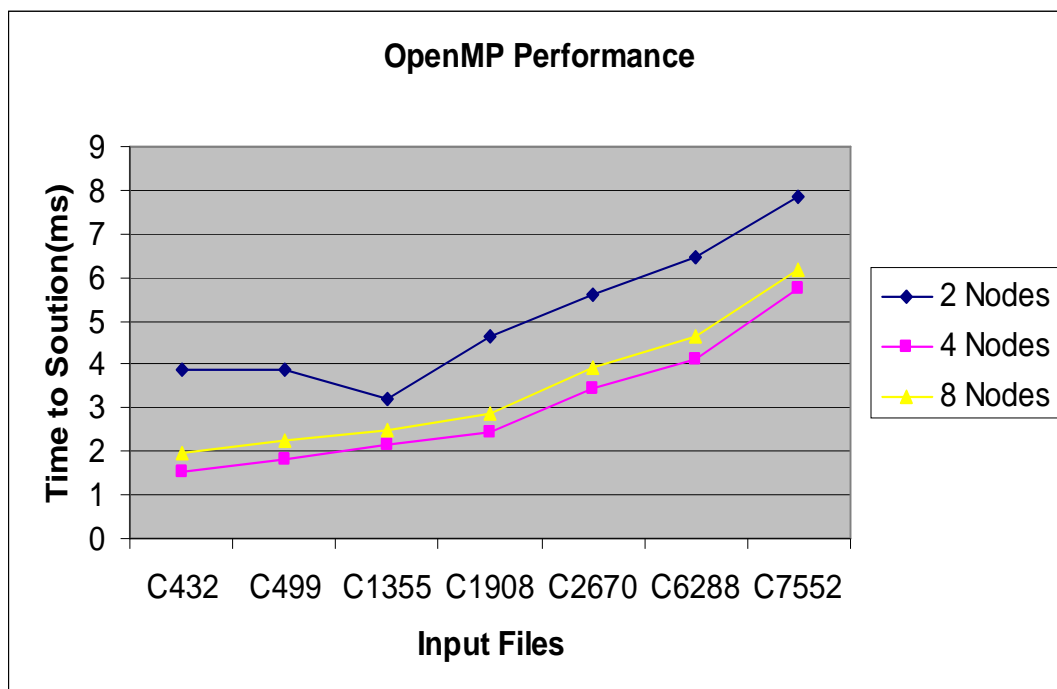
MPI Version:

In general, the performance of MPI version of the code improves as the input file size is increased. This means that the time taken does not scale linearly with file size. The increase is much slower. This is because of greater increase in computation as compared to the increase in communication hence improving the computation/communication ratio. Also for big files the performance improves as the number of nodes is increased while it remains constant or even degrades for small files. This is because, for big files the increase in communication overhead is less as compared to the increase in parallelism obtained whereas for small files the opposite holds true.



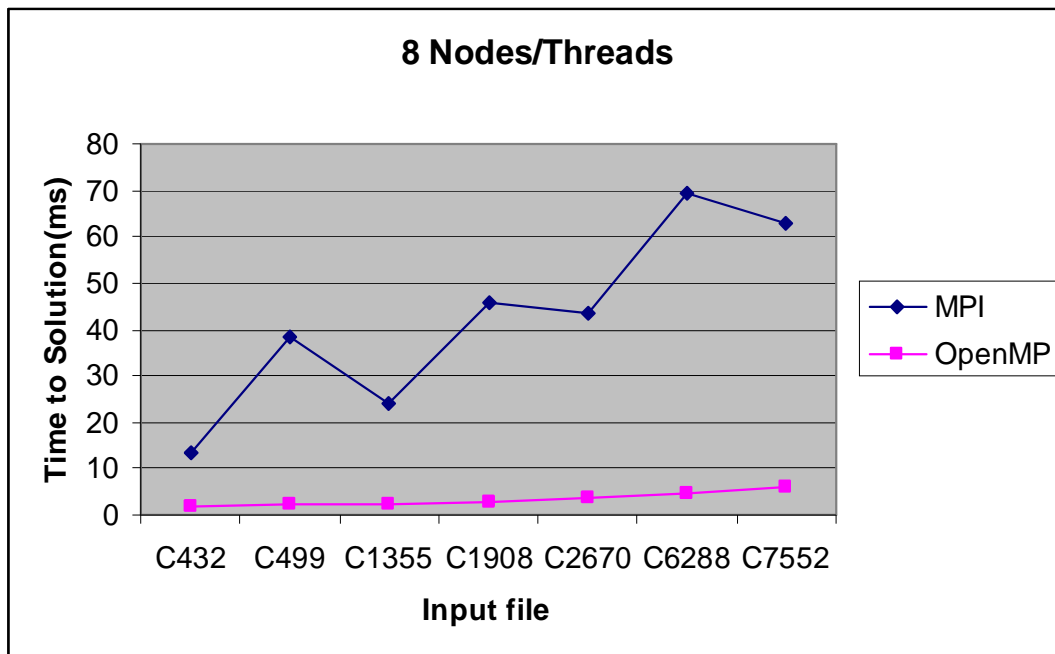
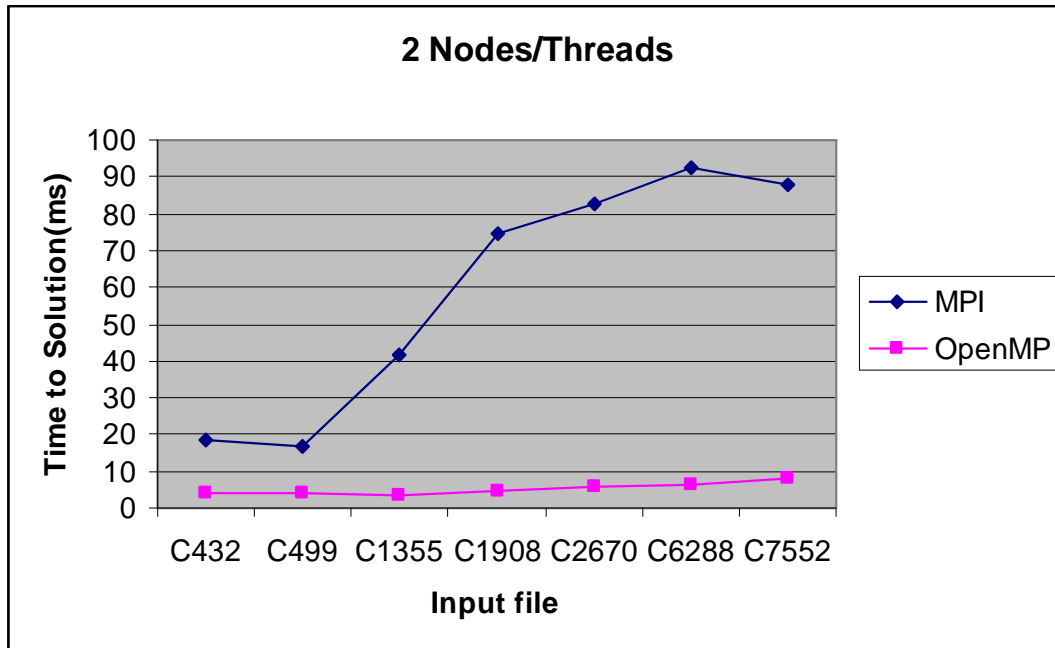
OpenMP Version:

Time to solution for OpenMP version of the code continuously kept increasing as input file size was increased. This was true even when we increased the number of threads. OpenMP version of the code gave the best performance when the number of threads was set to 4. This was because we ran our code on quad-core (4 processors) machines. So the parallelism was best exploited when one thread ran on each core. When the number of threads was increased further the scheduling overhead increased and performance degraded.



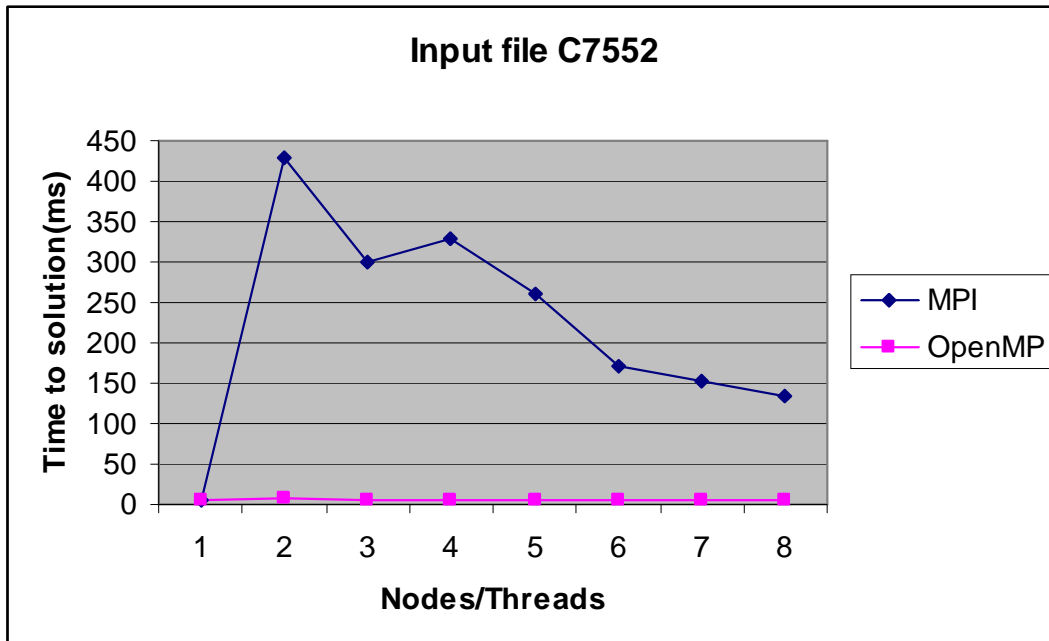
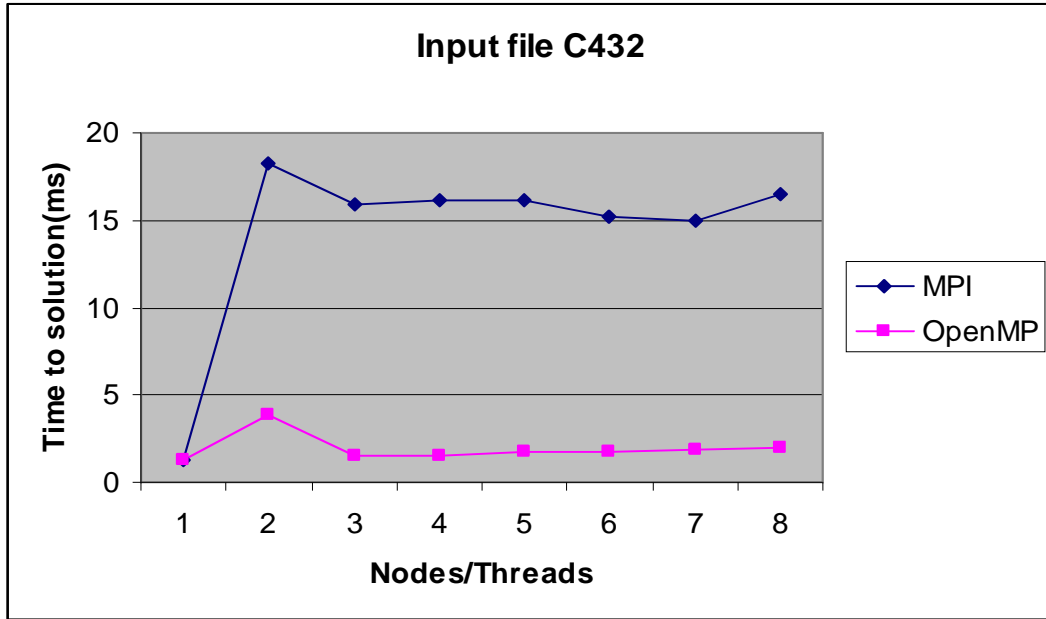
## MPI vs OpenMP

For increasing input file sizes (on 2 and 8 nodes/threads):



## MPI vs OpenMP

For increasing number of nodes (on the smallest and biggest file):



### Final result –

Serial Version > OpenMP > MPI

MPI does not give a good performance because it incurs a lot more communication overheads compared to the amount of parallelism it achieves, but the performance gets better with increase in file size and number of nodes. OpenMP gives better performance than MPI because there are no communication overheads but the performance degrades with increase in number of threads and hence it is not scalable. The serial version gives best performance for the dataset we experimented on. It is possible that on testing on much bigger files we would have got better performance from MPI than what we got from the serial version.

### **References** –

- [1] ISCAS-85 Benchmarks: <http://www.fm.vslib.cz/~kes/asic/iscas/>
- [2] Parallel logic simulation on a workstation cluster by Murakami, T. Wada, K. Okano, S. Tsukuba Univ., Ibaraki: [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?tp=&arnumber=519459&isnumber=11343](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=519459&isnumber=11343)
- [3] <http://www-unix.mcs.anl.gov/mpi/>
- [4] <https://computing.llnl.gov/tutorials/mpi/>
- [5] <http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html>
- [6] <https://computing.llnl.gov/tutorials/OpenMP/>