Assignment 3
Report

Submitted by :
Punit Kumar Jha
NETID: punit2
Graduate Student
Department of Chemistry

November 16, 2017

Contents

List of Figures

List of Tables

1 Introduction:

1.1 Objective

This assignment requires us to utilize a multi-core CPU using OpenMP and MPI to implement a dense matrix-matrix multiplication routine and understand its performance behavior. We shall also learn more about PAPI (**P**erformance **A**pplication **P**rogramming **I**nterface) to measure the cache misses and to analyze the performance based on the profiling output.

1.2 PAPI Documentation

The PAPI documentation was read and I learnt about its architecture[1]. The figure below shows PAPI's architecture. PAPI has two layers of architecture:

- The Framework layer – this consists of the API (low level and high level) and machine independent support functions.
- The Component Layer – defines and exports a machine independent interface to the machine dependent functions and data structures. These functions are defined in the components, which may use kernel extensions, operating system calls, or assembly language to access the hardware performance counters on a variety of subsystems.

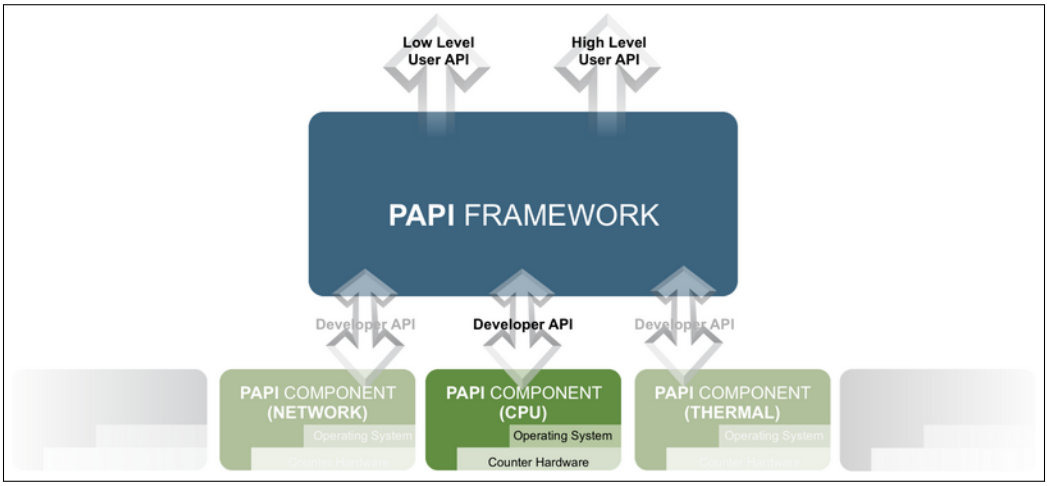

Figure 1: Internal design of the PAPI architecture

- The `src` folder of the tarball provided to us with the `main.c` file. This was the main function which linked the 3 different matrix multiplication implementation in the files `basic_seq.c`, `mpi.c` and `openmp.c`, with the PAPI implementations in the files `papi_helpers.c` and `papi_support.c`. The files `papi_helpers.h` and `papi_support.h` contain the list of functions defined in the `.c` files. I went thorough the files and found the following functions defined in the `papi_support.c` files:
  - `void papi_init()` - this function initializes the PAPI system. It uses the `PAPI_library_init()` function to initialize the PAPI library and `PAPI_create_eventset()` function to initilize the event set that are to be profiled..
  - `void papi_prepare_counter(int eventcode)` - This function is used to initialize various counters that one might need. It uses `PAPI_add_event()` function.
  - `void papi_start()`- starts the PAPI counter. It uses `PAPI_start()` function.
  - `papi_stop_and_report()` - this function stops the counter and prints the results. The functions used within it are `PAPI_stop()` and `PAPI_reset()`

The following functions defined in the `papi_helpers.c` file:

- `char papi_error_to_string()` - this function returns a string description of the given error code. This function has a list of different PAPI errors that are listed such as - `PAPI_EINVAL`, `PAPI_ENOMEN` etc.
- `void handle_error()` - this function prints error message and exits.
- `void check_error()` - this function is used to wrap the calls to PAPI libraries.
- `double current_time()` - this function returns the current time using the gettimeofday() function of C++.

Finally the `main.c` files links and uses all the above mention functions to measure the events of interest using PAPI. In our problem we have to measure the L2 data cache misses and accesses these events/counters are defined in the file as

```
#ifndef NOPAPI
papi_prepare_counter(PAPI_L2_DCM);
papi_prepare_counter(PAPI_L2_DCA);
#endif
```

2 PART A- Setting up the environment

The basic implementation of the matrix multiply was provided to us in the `basic_seq.c`. The tarball was uploaded to the cluster and the modules were loaded. After extracting the source code and compiling, two binaries were obtained as `test_mpi_1000_1000` and `test_omp_1000_1000`. The run script was then submitted on the cluster and the output file was produced as `out-JOBID.txt`.

- **What are `test_mpi_1000_1000` and `test_omp_1000_1000` doing ?**
- The two binaries that are created, `test_mpi_1000_1000` and `test_omp_1000_1000` are used to run the MPI and OpenMP implementation of the dense matrix multiplication with PAPI.

- **What is needed to run OpenMP? How many threads are being used in OpenMP and how is that controlled?**

- OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most platforms, instruction set architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. The `omp.h` header file is used in the main program to be able to use OpenMP in the codes.

  The compiler options needed to run OpenMP are listed below:

Compiler	Compiler Options	Default # of threads (`OMP_NUM_THREADS` not set)
GNU (gcc, g++, gfortran)	-fopenmp	as many threads as available cores
Intel (icc ifort)	-openmp	as many threads as available cores

- The codes that were provided used 8 threads in the program.

- In the codes that were provided the number of threads were controlled by the environmental variable `OMP_NUM_THREADS`, in the submission script. This provides run time control of the number of threads to be used in the program. Another way of declaring the number of threads is to use `omp_set_num_threads` directive inside the program.

- **What is needed to run MPI? How many processes are being used in MPI and how is that controlled?**

- Open MPI is an open source Message Passing Interface implementation that is developed and maintained by a consortium of academic, research, and industry partners. To be able to use MPI, `mpi.h` header file is used in the main program.

- For the GNU compiler and the Intel compiler, the `mpicc` command is used to compile the codes. In the codes that were provided, 8 processes were being used.

- To execute the code the command is `mpirun -np <no. of Processors> ./a.out` for GNU compilers, for Intel compilers `mpiexec` is sometimes used instead of `mpirun`. The `<no. of Processors>` option is used in the submission script to control the number of nodes available.

3 PART B- OpenMP implementation

The OpenMP implementation of the matrix multiplication is shown below.

```
extern double A[N][M];
extern double B[M][N];
extern double C[N][N];

/**
 * The basic dense matrix multiply.
 * Use this as a stub for your other implementations.
 */
void openMP_MM()
{
    int i, j, k;
    #pragma omp parallel shared(A,B,C) private(i,j,k)
        {
        #pragma omp for schedule(static)
        for (i = 0; i < N; ++i) {
                for (j = 0; j < N; ++j) {
                C[i][j] = 0;
                        for (k = 0; k < M; ++k) {
                                C[i][j] += A[i][k] * B[k][j];
                }
               }
              }
             }
}
```

3.1 Optimization report on the OpenMP code

Using the `-qoptrpt` option in the Make file the compiler optimization report for the written OpenMP code were generated. The optimization details are presented below:

- The compiler could not vectorize the outermost loop instead the loop was transformed to a *memset*.
- The inner loops were already vectorized
- The two inner loops were unrolled by the compiler.

3.2 OpenMP Scheduling

The various scheduling options that are available with OpenMp are:

- **Static Schedules** - By default, OpenMP statically assigns loop iterations to threads. When the parallel for block is entered, it assigns each thread the set of loop iterations it is to execute.
- **Dynamic Schedules**- OpenMP assigns one iteration to each thread. When the thread finishes, it will be assigned the next iteration that hasnt been executed yet. However, there is some overhead to dynamic scheduling. After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration.
- **Guided Schedules** - Instead of static, or dynamic, we can specify guided as the schedule. This scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.
- **Auto** - The auto scheduling type delegates the decision of the scheduling to the compiler and/or runtime system.

Table 1: Time taken by different OpenMP Scheduling schemes for M=1000 and N=1000 using 8 threads:

Serial No.	**Scheduling Type**	**Time Taken in sec**
1.	Static	0.0477
2.	Dynamic	0.0780
3.	Auto	0.0527
4.	Guided	0.0526

3.2.1 Conclusions

- OpenMP automatically splits for loop iterations. Depending on the program, the default behavior may not be ideal.
- For loops where each iteration takes roughly equal time, static schedules work best, as they have little overhead. For loops where each iteration can take very different amounts of time, dynamic schedules, work best as the work will be split more evenly across threads.
- Specifying chunks, or using a guided schedule provide a trade-off between the two.
- As can be seen form the Table above, that for dense matrix multiplication, static scheduling works the best.

3.3 OpenMP Strong Scaling Experiment

The common measures of parallel scaling are:

- **Strong Scaling**: Time for fixed problem size as number of processor is increased.
- **Weak Scaling**: Time for fixed computational work per processor as problem size is increased.

In our experiment we measured the strong scaling of our codes by doubling the number of threads every time. Let the total parallel work to be done be p and the total serial work be s.

- Single worker runtime: $T_f^s = s + p = 1$
- Same problem on N workers: $T_f^p = s + \frac{p}{N}$
- Similarly for performance in strong scaling: The serial work performed is $P_f^s = \frac{s+p}{T_f^s} = 1$
- The parallel performance is : $P_f^p = \frac{s+p}{T_f^p} = \frac{1}{s+\frac{1-s}{N}}$

Table 2: Time taken by the OpenMP program with different threads for M=2000 and N=2000

Serial No.	**No of threads**	**Time Taken in sec**	**L2 Cache Accesses**	**L2 Cache Missses**
1.	1	2.844	1057980389	40143684
2.	2	1.445	1059256023	42718047
3.	4	0.73630	1059293880	42602357
4.	8	0.3980	1119141780	52650017
5.	16	0.25667	66123881	2557736

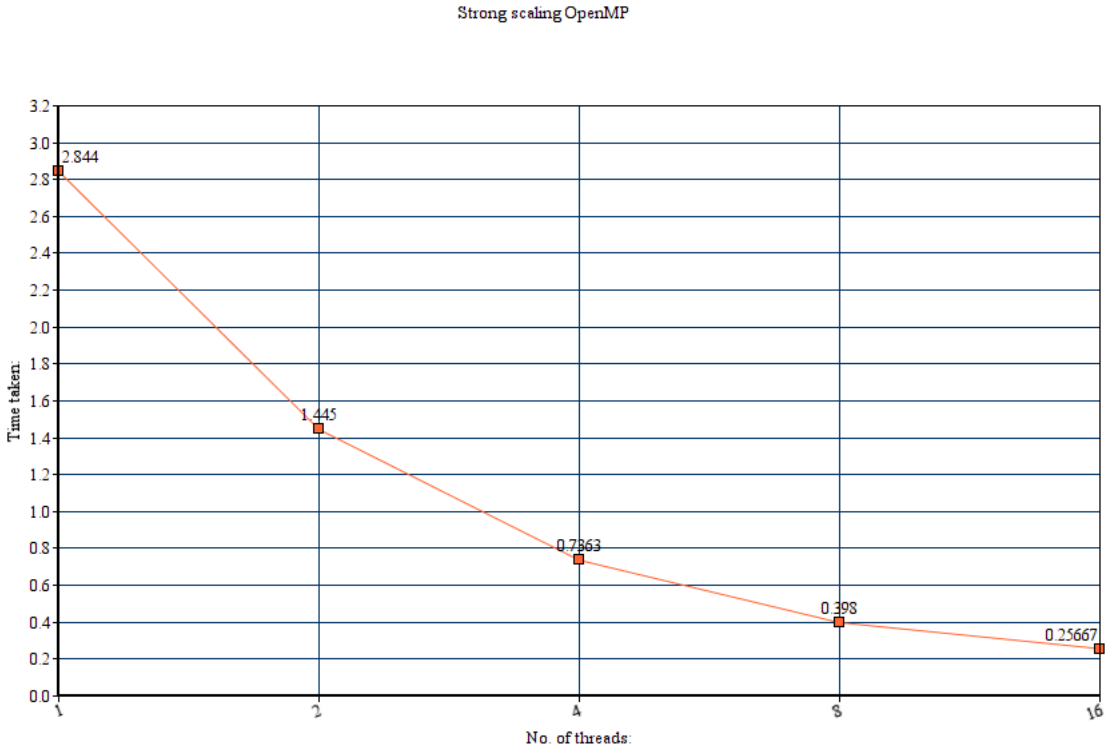

Figure 2: Plot of Time taken vs No. of Threads for OpenMP

3.3.1 Conclusions: Strong Scaling OpenMP

- We see that as we increase the number of threads the time taken to do the matrix multiplication by the OpenMP code goes down exponentially.
- Although the time taken goes down, however the efficiency decreases.

3.4 OpenMP: Experiment with varying matrix sizes

The OpenMP code above was used for dense matrix multiplication using different matrix sizes. The time taken, L2 data cache accesses and misses are presented in the table below. For all these experiments the number of threads was kept fixed at 8. The time taken for the serial code is also shown in Table 5 for comparison.

Table 3: Time taken by the OpenMP program with matrix multiplication of different sizes with constant 8 threads.

No.	N	M	Time Taken in sec	L2 Cache Accesses	L2 Cache Missses
1.	100	100	0.0001030	14120	1965
2.	100	1000	0.0005980	116285	25121
3.	1000	100	0.0057251	1592201	165311
4.	1000	1000	0.0477390	17521940	242980
5.	2000	2000	0.3980551	140177968	6628070

Table 4: Time taken by the serial program with matrix multiplication of different sizes.

No.	N	M	Time Taken in sec	L2 Cache Accesses	L2 Cache Missses
1.	100	100	0.0003181	102491	1113
2.	100	1000	0.0035162	903190	168605
3.	1000	100	0.0385139	12732669	1228321
4.	1000	1000	0.3339472	140070372	1736701
5.	2000	2000	2.7713308	1119094783	52008927

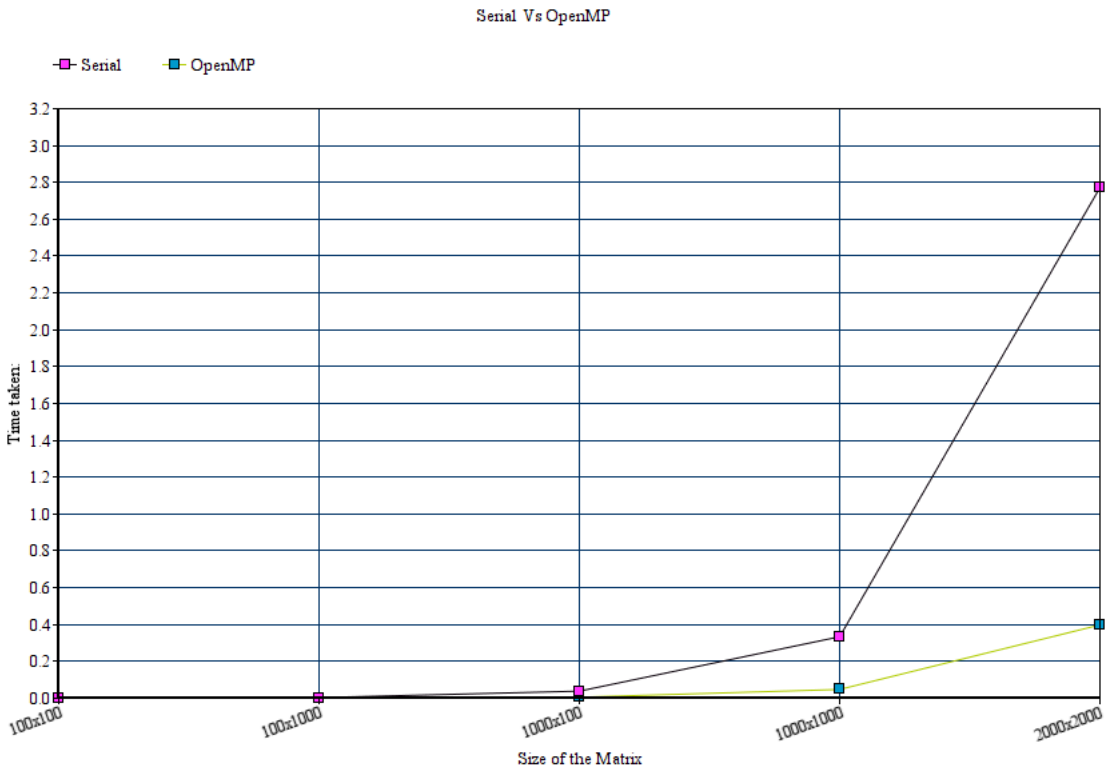

Figure 3: Plot of Time taken for Serial Codes vs OpenMP code

3.4.1 OpenMP Conclusion

- From the data and the graph presented above, we see that as the size of the matrix increases the time taken by both the serial version and the OpenMP. However, we observe that time taken by the parallel OpenMP code is 5-7 times faster than the serial version - when using 8 threads. Using 16 threads will cause the parallel codes to execute even faster.
- It can also be observed that the L2 cache accesses and misses also go up as the size of the matrix increases.
- We notice that the time taken for multiplication of two a N=100 and M=1000 matrices is almost 10 times lower than the time taken for the multiplication of two N=1000 and M=100 matrices. This is accompanied by significant increase in cache access and misses for the later case. This implies that accessing and multiplying matrices that are taller (have more number of rows) is needs more time and cache access. Hence, tiling them might be a good idea.

4 PART C- MPI implementation

Using the better implemented version in Part B , the tiled version was implemented. The code is shown below. It can be seen from the code below that every dimension was tiled. Different experiments were run using different matrix sizes and the results are shown below:

```
#include <mpi.h>

extern double A[N][M];
extern double B[M][N];
extern double C[N][N];

/**
 * The basic dense matrix multiply.
 * Use this as a stub for your other implementations.
 */
void mpi_MM()
{
    int numprocs;  //This is the number of processes in the group.
    int procid;    //This is the process identifier
    int source;    //This is the ID of the message source
    int numworkers; // This is the number of processors
    int dest;      //This is the ID of the message destination
    int rows;      //This is the rows to be sent
    int averow;    //This is the the aver no. of rows that are sent
    int extra;     //this is the number of left over rows after calculating the average
    int offset;    //This is the shift in the row indexes
    int i, j, k, rc;

    MPI_Comm_rank(MPI_COMM_WORLD,&procid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

//*****************************************************************
    if (procid ==0) // If its a master process
    {
     numworkers=numprocs-1;
//sending the data to worker tasks starts here
     averow = N/numworkers;
     extra = N%numworkers;
     offset=0;
     for (dest =1; dest<=numworkers; dest++)
     {
      rows = (dest <= extra ) ? averow+1: averow;
      MPI_Send(&offset,1,MPI_INT,dest,0,MPI_COMM_WORLD);
      MPI_Send(&rows,1,MPI_INT,dest,0,MPI_COMM_WORLD);
      offset=offset+rows;
     }

//receiving reuslts form worker processes
    for(source=1;source<=numworkers;source++)
    {
     MPI_Recv(&offset,1,MPI_INT,source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     MPI_Recv(&rows,1,MPI_INT,source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     MPI_Recv(&C[offset][0],rows*N,MPI_DOUBLE,source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     }
     } // the if statement ends here

//*********************************the if statements for workers start here****
    else{
     MPI_Recv(&offset,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     MPI_Recv(&rows,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
     for (i = 0; i < rows; ++i) {
        for (j = 0; j < N; ++j) {
```

```
            C[i][j] = 0;
            for (k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    MPI_Send(&offset, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Send(&C, rows*N, MPI_DOUBLE,0,0, MPI_COMM_WORLD);
    }
}
```

4.1 Optimization report on the MPI code

Using the `-qoptrpt` option in the Make file the compiler optimization report for the written MPI code were generated. The optimization details are presented below:

- The loop process with rank 0, which sends matrix indexes to worker processes was not able to be vectorized by the compiler since it had FLOW and ANTI dependencies.
- For similar reasons, the for loop on the process with rank 0, which receives computed matix elements form the worker processes could not be vectorized.
- The for loops on the worker processes which received data from the root process was not vectorized but was instead converted to a **memset** or **memcpy**.
- The matrix multiplication loops were already vectorized.

4.2 MPI Strong Scaling Experiment

MPI strong scaling experiment was performed with the number of processors double each time. The results of the experiment are reported below:

Table 5: Time taken by the MPI program with different processes for M=2000 and N=2000

Serial No.	**No of processes**	**Time Taken in sec**	**L2 Cache Accesses**	**L2 Cache Missses**
1.	1	Same as serial implementation		
2.	2	2.9244	4797	3589
3.	4	1.0697	7369	5587
4.	8	0.5041	22127835	9362
5.	16	0.27749	19098224	12761

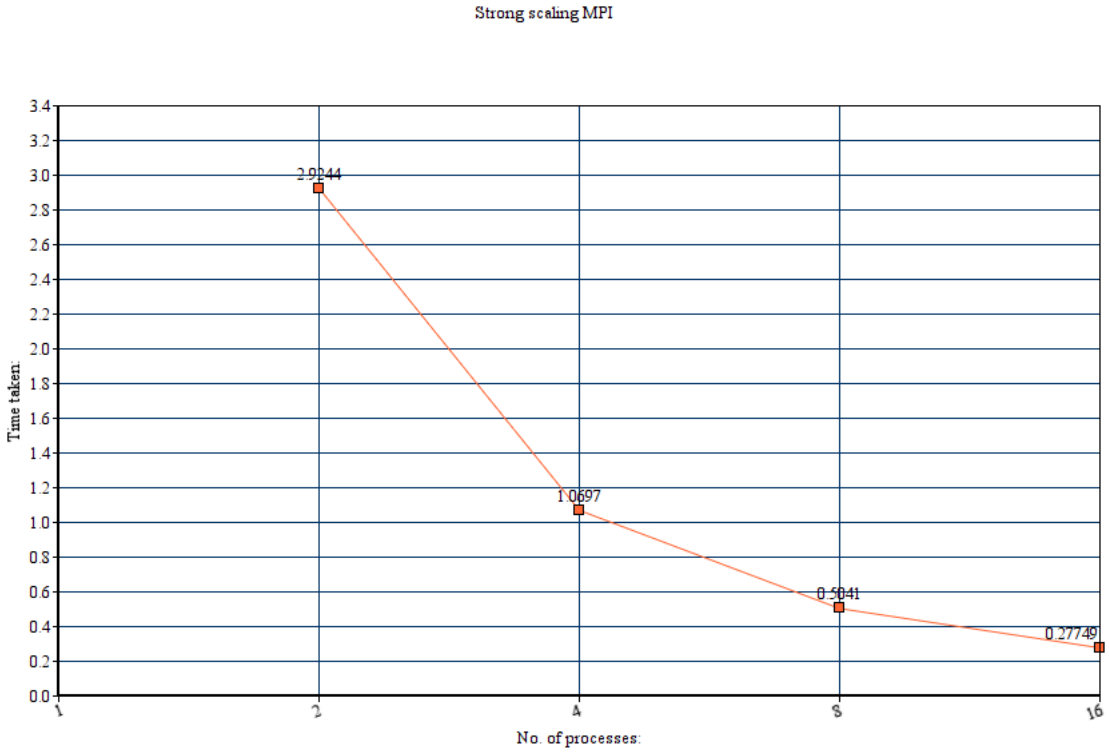

Figure 4: Plot of Time taken vs No. of Processes for MPI

4.2.1 Conclusions: Strong Scaling MPI

- We observe that as we increase the number of processes the execution time of the dense matrix multiply goes down.
- However as stated earlier, the efficiency of the process may go down in strong scaling (the detailed theoretical description in OpenMP section).

4.3 MPI: Experiment with varying matrix sizes

The MPI code above was used for dense matrix multiplication using different matrix sizes. The time taken, L2 data cache accesses and misses are presented in the table below. For all these experiments the number of processes was kept fixed at 8. The time taken for the serial code is also shown in Table 6 for comparison.

Table 6: Time taken by the MPI program with matrix multiplication of different sizes with constant 8 processes.

No.	N	M	Time Taken in sec	L2 Cache Accesses	L2 Cache Missses
1.	100	100	0.0007169	7186	3427
2.	100	1000	0.0012338	7465	3659
3.	1000	100	0.0671892	5434	4357
4.	1000	1000	0.0706789	6259	4662
5.	2000	2000	0.4985089	2557736	9408

Table 7: Time taken by the serial program with matrix multiplication of different sizes.

No.	N	M	Time Taken in sec	L2 Cache Accesses	L2 Cache Missses
1.	100	100	0.0003181	102491	1113
2.	100	1000	0.0035162	903190	168605
3.	1000	100	0.0385139	12732669	1228321
4.	1000	1000	0.3339472	140070372	1736701
5.	2000	2000	2.7713308	1119094783	52008927

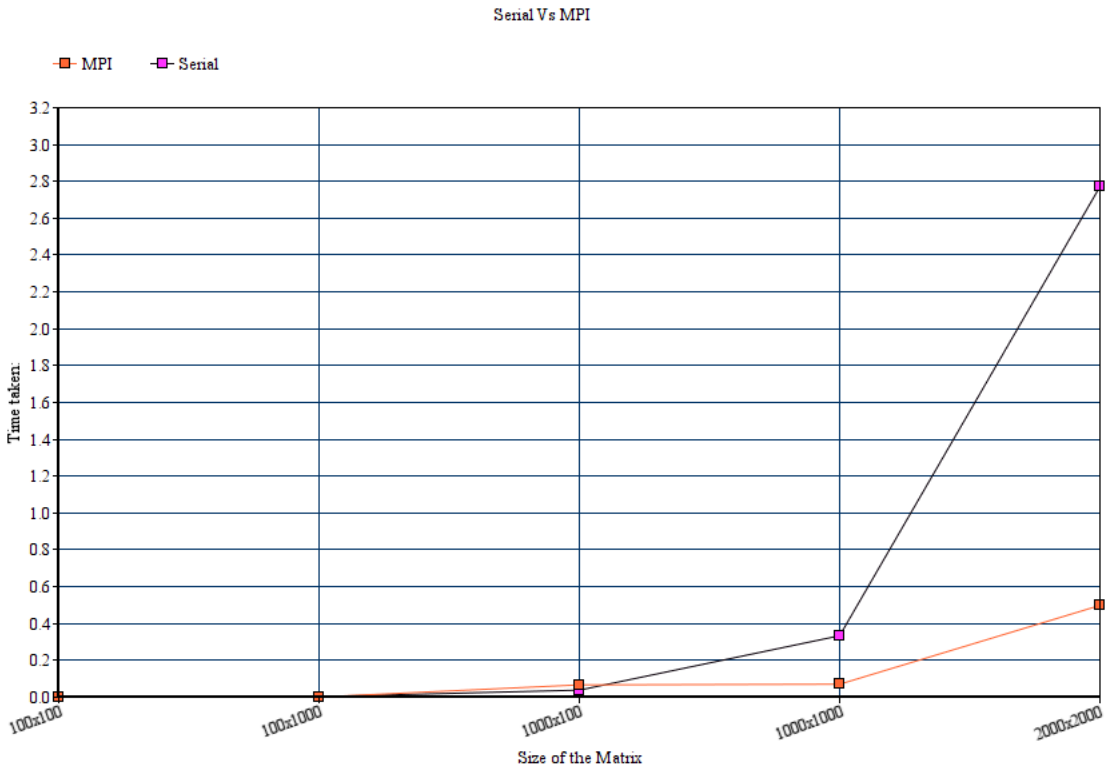

Figure 5: Plot of Time taken for MPI and Serial Codes vs Size of the Matrix

4.3.1 MPI Conclusion

- From the data and the graph presented above, we see that as the size of the matrix increases the time taken by both the serial version and the MPI. However, we observe that time taken by the parallel MPI code is 5-6 times faster than the serial version - when using 8 processes. Using 16 processes will cause the parallel codes to execute even faster.
- It can also be observed that the L2 cache accesses and misses also go up as the size of the matrix increases. However, the increase is not significant.
- We observe an abrupt increase in the cache accesses as for 2000x2000 matrix.
- We notice that the time taken for multiplication of two a N=100 and M=1000 matrices is almost 16 times lower than the time taken for the multiplication of two N=1000 and M=100 matrices. This is accompanied by slight increase in cache misses and a decrease in cache misses for the later case. This implies that accessing and multiplying matrices that are taller (have more number of rows) is needs more time and cache access. Hence, tiling them might be a good idea.

5 MPI Vs. OpenMP

In this section, we compare the performance of the MPI and the OpenMP versions of our codes.

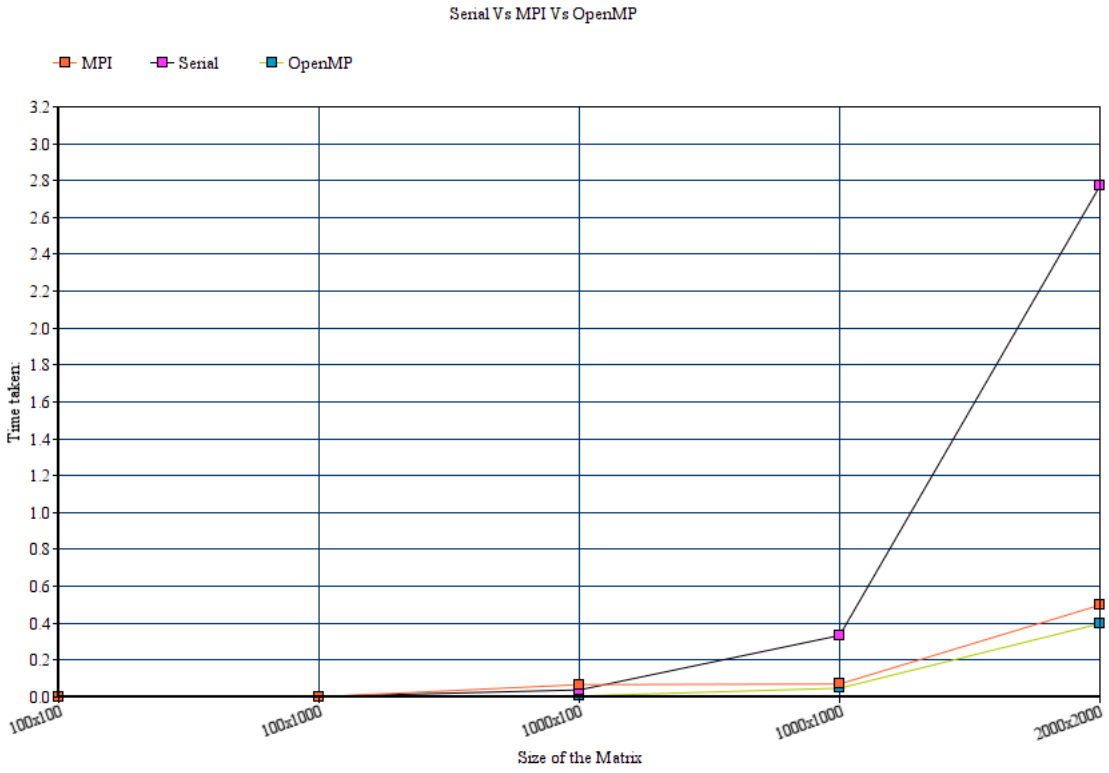

Figure 6: Plot of Time taken for Serial Codes vs MPI vs OpenMP

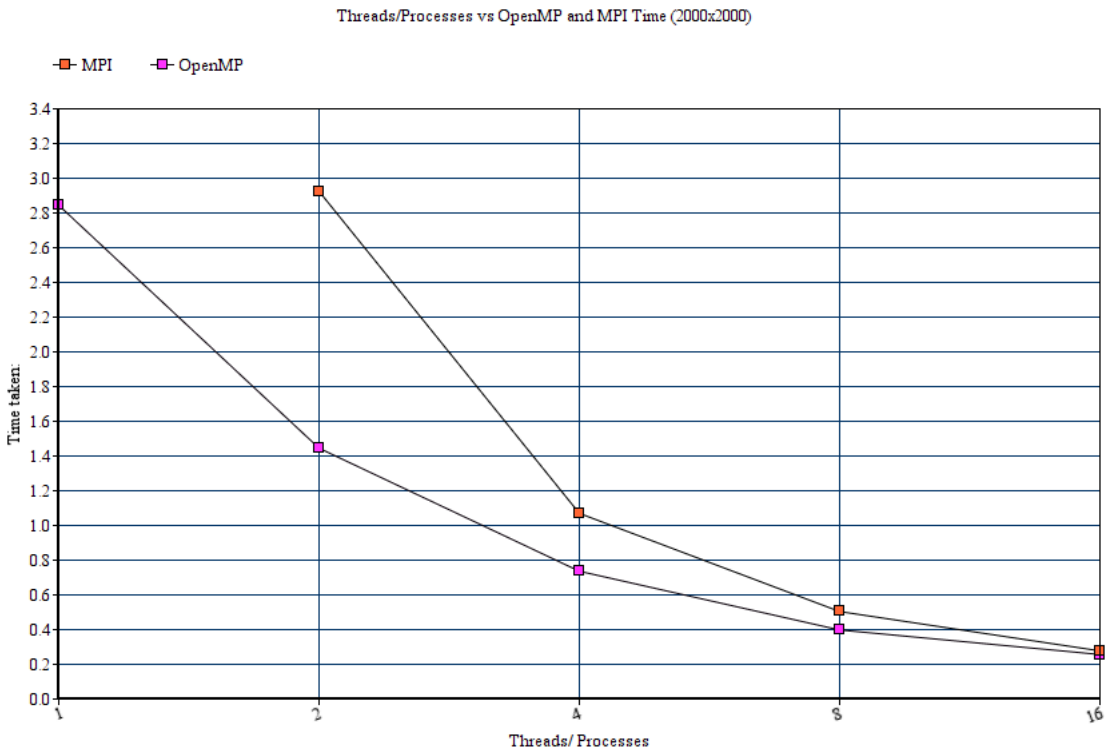

Figure 7: Plot of Time taken for MPI vs OpenMP as the No. of threads/processes increased.

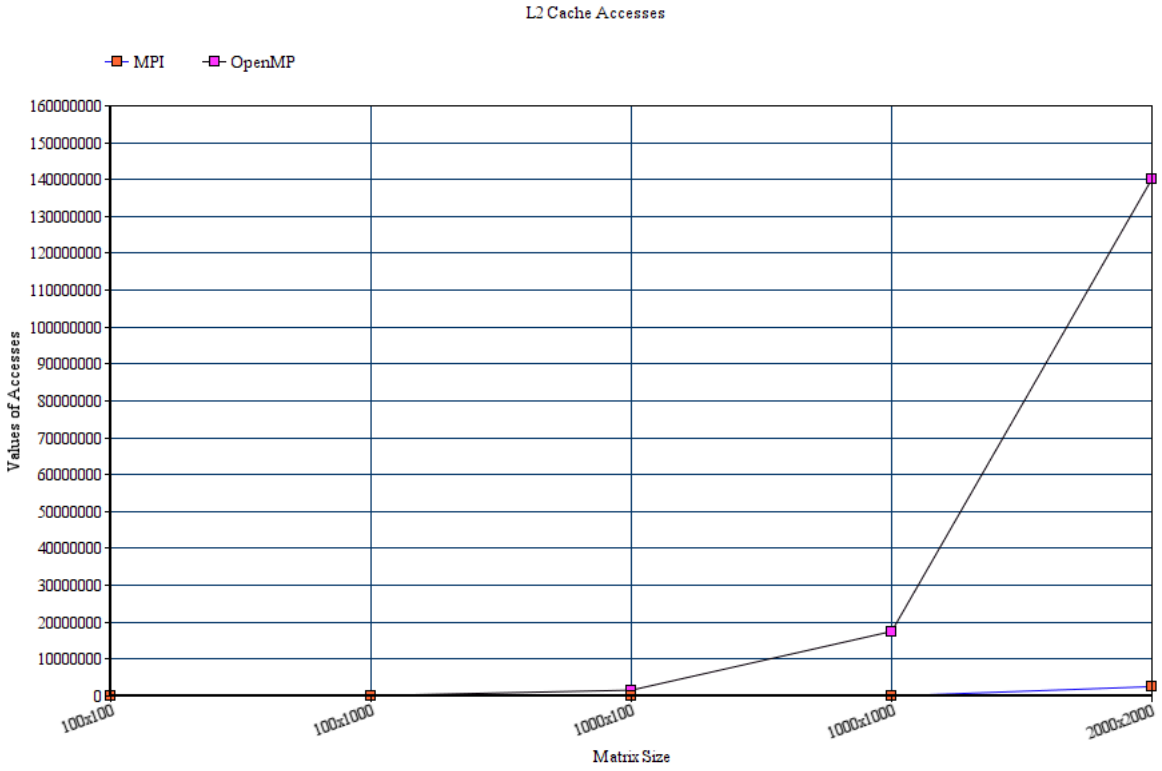

Figure 8: Plot of L2 cache accesses for MPI vs OpenMP

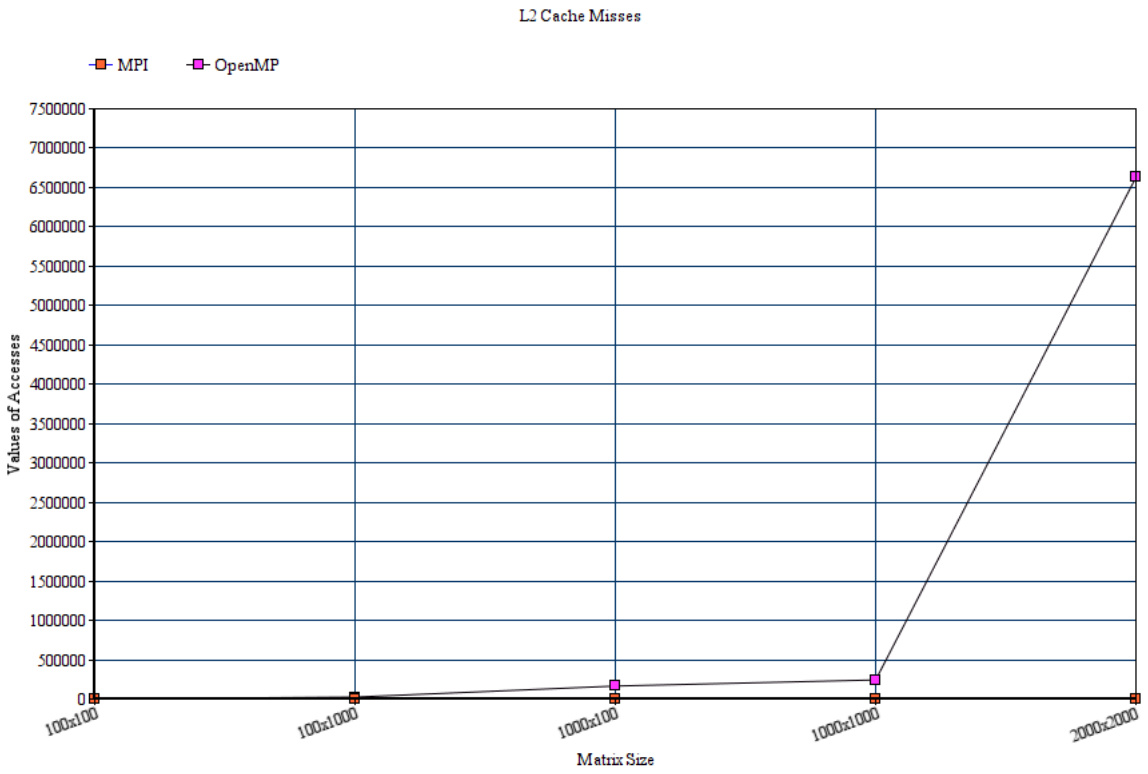

Figure 9: Plot of L2 cache misses for MPI vs OpenMP

Points of Comparison	OpenMP implementation	MPI implementation
Time taken for Multiplication	The OpenMP codes take comparatively less time since we are running on a SMP system and	MPI codes take slightly more time than the OpenMP code. MPI is not very efficient in SMP architectures while using only one node.
L2 Cache Misses	The L2 Cache Misses arealmost 100 times larger	The L2 Cache Misses are far lower for MPI
L2 Cache Accesses	The L2 Cache Accesses are almost 100 times larger	The L2 Cache Accesses are far lower for MPI
Decrease in time as a function of processes/threads	The time declines in general	Sharper decline in time taken as compared to OpenMP

Table 8: OpenMP vs MPI Comparison

6 Comparisons and Conclusions

- Detailed observations on both OpenMP and MPI programs are listed in their respective sections above.
- Both OpenMP and MPI programs are faster than the serial version by atleast 4-5 times.
- Both OpenMP and MPI implementations distribute the matrix among the processors and assign more rows per processors. This distribution increases the spatial memory locality.

- Since It takes longer to access data in main memory or another processors cache than it does from local cache, MPI is slightly slower than the OpenMP version.
- From the experiment and study presented above list the major differences

Pros and Cons of OpenMP and MPI.

OpenMP Advantages	**MPI Advantages**
Easy to implement parallelism	Portable to be distributed and shared memory machines.
Low latency, high bandwidth	Scales beyond one node
Communication is Implicit	No race condition problem
Coarse and fine granularity	
Dynamic load balancing capablities	
OpenMP Disdvantages	**MPI Disdvantages**
It is usefull only on shared memory machiness	Suffers from High latency, low bandwidth
Scale within one node only	Difficult to develop and debug as codes are very verbose
Race condition can be a problem	Communication is explicit
No specific thread order	Load balancing may be difficult

7 References

[1] http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview.

[2] CS-420 Notes by Prof. Mark Snir