Journal of Engineering and Applied Sciences 12 (3): 609-616, 2017
ISSN: 1816-949X

Performance Analysis of MPI Approaches and Pthread in Multi-Core System

Ali A. Alabboud, Sazlinah Hasan, Nor Asilawati Abdul Hamid and Ammar Y. Tuama
Faculty of Computer Science and Information Technology,
University Putra Malaysia, Selangor, Malaysia

Abstract: Comparison among the HPC techniques has been made in order to address the highest and lowest performance of each in terms of execution time, speedup and efficiency when it is used with the HPc multicore system. The matrix multiplication in a variant size is used as a common complex task to examine the performance of each approach. FSKTM server has been used as an HPC multicore system to perform the approaches and tasks. Based on the results, it shows that Hybrid MPI/OpenMP approach is the best in terms of execution time, speed up and efficiency than other approaches when the matrix size is very high (>1024×1024 size). Furthermore, the results show that the compiler version has a significant impact over the performance of Pthread. With a new compiler, the performance becomes much better due to the improvement in code translation.

Key words: Parallel computing, MPI, open MP, multithreading, hybrid

INTRODUCTION

High-Performance Computing (HPC) is a collection of independently connected computers or processors that collaborate together to solve high complex problems even though continuous development of the data storage, computer power, communication speed and the available computational re-sources are failing to keep up with the current complex application requirements. Therefore, HPC infrastructure becomes the new trend for research. The HPC applications are also known as supercomputers. The concept of the HPC systems is in the ability of solving the complex problems and executing the applications in parallel. Therefore, the performance of the HPC system is mainly depending on the algorithms, protocols and techniques that are used to manage and allocate the available resources for the parallel processes such as OpenMP and MPI.

Recently, many research studies have focused on parallel techniques and algorithms in order to enhance the response time and performance for the computational parallel applications. Those algorithms are arranged as approaches in order to simplify the application development and code maintenance. There are three main programming approaches in HPC system which are Message Passing Interface (MPI) Open Multi-Processing (OpenMP) and Multithreading.

With all of these available approaches users are faced with the challenge to select the best one that is suited for their hardware architecture. In view of this research, it is seen crucial to examine each performance of the four techniques (OpenMP, MPI, Hybrid OpenMP and MPI and Pthread) in order to distinguish the strength of their performance respectively. With the experiment results, we can determine which framework has the best performance with a particular problem complexity.

A number of research has compared between the performance of OpenMP and MPI to determine one of which that meets for a specific application. A few latest works of research have added a hybrid OpenMP and MPI to the compression as a new trend that mixes shared memory and message passing to produce a better performance.

Wu *et al*. (2012) have proposed a solution to design two-level parallel loop self-scheduling schemes by adopting hybrid MPI and OpenMP programming model. In the first level an MPI process is run by computing node for inter-node communication where in the second level an OpenMP is used to execute the iteration in each processor core. The solution had been outperforming the previous researches that are compared with. However, the performance of the proposed solution had not been calculated using a real research station processor; therefore, the compared previous solutions may not run with full advantage of multicore that were designed to. Furthermore, the implementation needs to be tested with more types of application programs to verify the performance and it calls for theoretical analysis.

The researchers in Wu *et al*. (2012) have studied a hybrid approach to program for distributed memory across node and shared memory access within each node system. The proposed solution has combined two of

Correspondong Author: Ali A. Alabboud, Faculty of Computer Science and Information Technology, University Putra Malaysia, Selangor, Malaysia

traditional programming models which are MPI and OpenMP to improve the performance of multi-core based systems. The multi-zone NAS parallel benchmarks with two full applications have been used as a performance measurement when running the solution on SGI Altix 4700 and an SGI Altix ICE 8200EX. Furthermore, the solution also presented a new data locality extension for OpenMP to improve the matching of the memory hierarchical structure. However, the proposed solution was compared with pure MPI only without including the other parallel models solution such as Open MP and standard C++parallel programming to the comparing list.

Sharma and Kanungo (2011) compared the performance of multithreading find-grained and course-grained problems as data intensive and computation-intensive problem. The researchers used MPI and hybrid MPI/Open MP approach to evaluate the problems. They evaluated the programming suitability model based on computational problems type. Unlike the other researchers which are comparing programming models on cluster of SMP nodes they compared the models on cluster of commodity multicore nodes. The result has shown that hybrid model produces better performance in most cases. However, the study did not use the pure OpenMP to show the performance of it with such problems.

Denis (2015) the researchers have presented a generic framework to be used with MPI implementation named Pioman. The proposed solution brought a seamless asynchronous communication progression using available cores. The solution was compatible with any runtime system because it used standard threads programming model. It was compared with the OpenMP and MPI programming model and the result has shown an improvement regarding overlap, multithreading and progression with outperforming the MPI models. However, the solution has not been compared with the advanced parallel programming models such as hybrid OpenMP/MPI which may be yielded better performance than the proposed solution.

The MPI+OpenMP programming model and matrix multiplication which are based on column wise and row wise block striped decomposition in the multi-core cluster system have been outlined by He *et al.* (2010). The experiment result has shown that the performance of parallel algorithm gain significant improvement when using MPI+Open MP matrices decomposition. However, the experiment has some limitations that may affect the reliability of the results. Matrix size used in the experiment was small with two sizes only (1400×1400 and 2100×2100). On top of that, the number of processor in the experiment did not make sense (5, 7 and 10). Apart from that, the experiment did not use a pure OpenMP in the implementation. Thus, the performance may be increased or decreased when these parameters are changed.

In 2015, the researchers Klawonn *et al.* (2015) using OpenMP with PETSc+MPI in the finite element assembly with shared memory parallel direct solver Pardiso to hybrid MPI/OpenMP parallelization in FETI-DP. Thus, the solution used OpenMP on subdomains where MPI used in between subdomains. The efficiency of the proposed solution has been investigated from two-dimensional nonlinear hyperlasticity. The solution improves the scalability for up to four threads for each MPI ranked on Ivy Bridge processor architecture with incremental improvement for up to ten OpenMP threads for each MPI rank.

Parallel programming models: There are four main techniques that are used to manage and code the applications in parallel in order to solve the complex problems using multicore system.

MPI (Message Passing Interface): MPI is known as a standardized and portable message-passing system to which it is exploited to function on an extensive variety of parallel computers. MPI indeed is the most popular programming model which uses message passing technique. It is structured by passing the message between the processors. It can be programmed in C/C++ or Fortran Having said that in view of this research, C++ is the only tool selected as programing language (Kotobi *et al.*, 2014).

Open MP (Open Multi-Processing): Open multi-processing, widely known as open MP is portable, scalable and compiler directives library is written to manage and execute the programs in parallel (Sharma and Kanungo, 2011). It is written to target the shared memory system by using fork and join models to run the code in parallel on multi-core server.

Open MP and MPI are identified hybrid when these two are combined by Mixing Shared Memory and Message Passing and they are used to measure better performance (Rabenseifner *et al.*, 2009).

Hybrid open MP/MPI: The differences between two previous techniques are not only in the way they manage the resources and execute the code but also in the core of the library itself. The open MP does not require a multicore compiler to research because of the compiler's directive nature which gives the ability to embed the library in the standard compilers. On the other hand, the

MPI requires a specific compiler to execute the MPI codes. However, instead of those differences between the two libraries, the developer and multicore designer are still able to write a code using both libraries at the same time. This combination takes the advantages of both to advance the system performance and extend the system abilities when solving the problems. This combination is called hybrid openMP/MPI.

Muli threading: Multi-threading which is also known as Pthread is defined as a POSIX standard which designates a thread model (Nichols *et al.*, 1996). Pthread allows a program to take charge of multiple different flows of research which overlaps in time. Each flow of research is referred to as a Thread and the creation and control over these flows is achieved by making calls to the POSIX Threads API (Application Programming Interface) (Kuhn *et al.*, 2000).

MATERIALS AND METHODS

Each technique will be classified based on the size of its performance. We will use the matrix multiplication as a complex issue to solve in parallel. The program will be coded in C++ 14 which is the latest standard version of C++ and executed with GCC 5.0 which is the latest version of standard C/C++ compiler. The FSKTM server which we have provided its specifications in objective section will be used to run the code.

Performance measurements: There are three main terms used to measure the performance of the HPC system which are: execution time, speedup and efficiency. Each of those terms describes a specific characteristic in term of performance. Due to the importance of those three terms, the developers of HPC approaches have included many tools in the coding library to calculate those terms.

Execution time: This term is measured by milliseconds or nanoseconds and present the time between starting execution the process till the execution is completed and retrieve the proper results. The value of execution time can be calculated by the following Eq. 1:

$$\text{Execution time} = \text{Time}_{\text{end}} - \text{Time}_{\text{begin}} \quad (1)$$

Speed-up: This term presents the differences in execution time between multicore and single core. In addition, it can be used to get the differences between multicores when using different number of cores. The value of speed-up can be calculated by the following Eq. 2:

$$\text{Speed-up} = \frac{\text{Execution time single core}}{\text{Execution time multi core}} \quad (2)$$

Table 1: Number of instructions

Matrix size	Number of instructions
16	4096
32	32768
64	262144
128	2097152
256	16777216
512	134217728
1024	1073741824
2048	8589934592
4096	68719476736
8192	5.49756E+11

When the speed-up value equals to the number of processors, the speed-up is linear. However, when it is less, the speed-up is poor. Theoretically, the speedup value cannot exceed the number of processor value.

Efficiency: Improving the performance of the system by adding more cores or processors can be inefficient sometimes. Therefore, we need a measurement to find the efficiency of the system and the amount of speed that is gained when adding more processors or improving the approaches. The $ efficiency $ value can be used for this purpose. It shows in percentage the amount of speed or performance that is gained when adding or using more processors compared with an optimal value linear speed up. The following equation can be used to calculate the efficiency:

$$\text{Efficiency} = (\text{Speed-up/number of processor}) \times 100 \quad (3)$$

The higher the efficiency, the more processor can be added to improve the performance. Theoretically, the value of efficiency cannot exceed 100%.

Experiment analysis: To understand the complexity of the matrix multiplication, an analysis has to be done to calculate the number of instructions that need to be executed. The complexity of the matrix multiplication is $O(n^3)$ where n is the size of matrix, e.g., matrix size 1024×1024 the n value will be 1024. Table 1 shows the amount of instructions that have to be executed to calculate the matrix multiplication results. With matrix sized <1024, the number of instruction will be <1 billion.

RESULTS AND DISCUSSION

In this study, the results of the experiment are introduced. First, the execution time is calculated for each model with a different size of matrix and number of processors. Then, the speed up and efficiency are calculated based on the execution time.

Compiler performance: Compiler of the programming language is responsible for converting the code that is written in any language to the machine language. However each language has different instructions, operations and way to write the code. Therefore, each programming language has its own compiler. It is an interpreter between the high-level language and computers hardware. For that, the programming language designers are trying to make the compiler better by frequent updates. For instance, GNU C/C++ Compiler (GCC) is updated every year. These updates not only support new features but also improve the translation from high-level code to more efficient machine code.

Many research papers have been conducted to optimize the compiler-transformed machine code in order to provide a better parallel code (Chen and Wu, 2003; Munir *et al*., 2015). Furthermore, the hardware venders have developed a compilers designed especially for parallel coding such as intel C++ parallel compiler.

For all above said, we can conclude that using a different compiler or old version may affect the performance of the parallel models. The FSKTM server that is used in the experiment has a 10 year-old GCC version which is 4.2 from 2007. Using such compiler may lead to:

- Inefficient machine code
- Not supporting all new features that developed in the latest version of C/C++2014 such as new execution time measurement and Pthread functions
- Lack in multithreading management since the multicore system was not widely used or fully supported by the C/C++compiler in 2007 hence making the developer design a new compiler for MPI

Now a days, the multicore system is widely used even with microcomputers and smartphones. Therefore, the programming language developers added fully supported models to multicore application in their language and updated the compiler to be more efficient when translating the code for such machine.

To examine the effects of the compiler on the performance of the application, we calculate the performance of Pthread application using two different compilers which are 4.2 from 2007 with 64 processors and 6.1 from 2016 with 8 processors.

Table 2 shows the execution time in seconds of 64 processors@4.2 and 8 processor@6.1 GCC. The latest GCC compiler has a significant improvement in the multithreading management with up to 100 times better in small matrix size and 10 times better with large matrix size. For that, the Pthread model is excluded from the experiment because it is not a fair comparison to compare

Table 2: Execution time of different GCC

Size	Pthread (GCC 4.2) (64 cores)	Pthread (GCC 6.1) (8 cores)
16	0.003	0.000030
32	0.020	0.000215
64	0.100	0.001732
128	0.420	0.010109
256	1.810	0.087348
512	7.630	0.749541

Table 3: Compiler performance with openmp

Size	Pthread (GCC 4.2) (64 cores)	Pthread (GCC 6.1) (8 cores)
16	0.0031	0.000488997
32	0.0043	0.000540972
64	0.00522	0.00135612
128	0.00529	0.00372005
256	0.0125923	0.032939
512	0.151071	0.279898

Table 4: Execution time of single core

Size	Execution time (sec)
16	0.0001
32	0.0003
64	0.0030
128	0.0200
256	0.2100
512	1.9000
1024	24.300
2048	353.56
4096	235000
8192	20.270

code generated with not fully supported the parallel application like GCC 4.2 with a fully parallel supported like MPICC compiler MPI or OpenMP library.

The open MP code also uses GCC compiler and that may also affect OpenMP code performance as shown in Table 3. However, the functions which are developed in OpenMP library helps to generate a better parallel code with better multi-processor management. Therefore when comparing the OpenMP performance on GCC 4.2 and 64 processor with GCC 6.1 and 8 processor, the performance of 64 cores is better than 8 processor but still inefficient. When we increase the number of processors 8 times (from 8-64 processors) and the matrix size is 512, the performance increases about 84% when it should be increased to at least about 300-400%. From the results, we conclude that the compiler version has an impact on the performance of the system and it should be updated frequently to improve the system performance.

Execution time: The execution time represents the time that is needed to run a program. The value of this performance measurement can be in second, millisecond and nanoseconds. Table 4 represents the execution time of the matrix multiplication using a single core measured by seconds. It can be noticed that the matrix size which is $<1024\times1024$ items runs very fast with <2 sec. However, with a larger matrices size, the execution time becomes

Table 5: Execution time (2 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.007214500	0.09083000	0.018812500
32	0.014858800	0.12599000	0.056046200
64	0.025112600	0.15294600	0.057370600
128	0.061269000	0.15499700	0.115553900
256	0.202703900	0.36895439	0.250733000
512	1.286105000	4.42638030	1.771174300
1024	15.17621820	58.3702880	19.70358040
2048	141.1732955	674.014270	140.8765484
4096	1387.597220	4117.73410	1222.258058
8192	11883.86630	50821.4360	11121.78057

Table 6: Execution time (4 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.0043005	0.06231	0.0120625
32	0.0088572	0.08643	0.0359366
64	0.0149694	0.104922	0.0367858
128	0.030561	0.106329	0.0740927
256	0.1148691	0.25310523	0.160769
512	0.826245	3.0365271	1.1356699
1024	9.3444558	40.042416	12.6338572
2048	88.9208895	462.37839	90.3294812
4096	856.9390593	2824.7937	783.706994
8192	7143.477303	34863.852	7131.24136

Table 7: Execution time (8 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.0020445	0.03038	0.0061875
32	0.0042108	0.04214	0.0184338
64	0.0071166	0.051156	0.0188694
128	0.014529	0.051842	0.0380061
256	0.0546099	0.12340454	0.082467
512	0.392805	1.4804958	0.5825457
1024	4.4424462	19.523168	6.4805796
2048	42.2738655	225.43822	46.3348116
4096	407.3972577	1377.2626	402.005142
8192	3396.079374	16998.296	3657.994273

Table 8: Execution time (16 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.001457	0.02201	0.0040625
32	0.0030008	0.03053	0.012103
64	0.0050716	0.037062	0.012389
128	0.010354	0.037559	0.0249535
256	0.0389174	0.08940533	0.054145
512	0.27993	1.0726041	0.3824795
1024	3.1658812	14.144336	4.254926
2048	30.126203	163.32769	30.421846
4096	290.3290802	997.8127	263.94277
8192	2420.194496	12315.092	2401.713412

much higher due to the high complexity and number of instructions. With 8192 matrix size, the application needs about 5.6 h to be completed.

From that, we conclude that the one billion instructions can classify the matrix into two categorize: low complex and high complex multiplication issue.

The performance of each model has been calculated based on different matrix size and number of processors starting from 2-64 processors. Table 5-10 show the execution time of the models with 2, 4, 8, 16, 32 and 64 processors, respectively. The performance is clearly improved by increasing the number of processors and

Table 9: Execution time (32 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.000846	0.01302	0.0024375
32	0.0017424	0.01806	0.0072618
64	0.0029448	0.021924	0.0074334
128	0.006012	0.022218	0.0149721
256	0.0225972	0.05288766	0.032487
512	0.16254	0.6344982	0.2294877
1024	1.8382536	8.367072	2.5529556
2048	17.492634	96.61638	18.2531076
4096	168.5781756	590.2554	158.365662
8192	1405.274224	7284.984	1441.028047

Table 10: Execution time (64 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.000235	0.0031	0.000625
32	0.000484	0.0043	0.001862
64	0.000818	0.00522	0.001906
128	0.00167	0.00529	0.003839
256	0.006277	0.0125923	0.00833
512	0.07315	0.151071	0.098843
1024	0.510626	1.99216	0.654604
2048	4.859065	23.0039	4.680284
4096	46.827271	140.537	40.60658
8192	390.353951	1734.52	369.494371

that is what the parallel models founded for. Even when u 2 processors are used, only the performance is higher than a single core processing. With the low complexity matrix multiplication, the performance of the models is extremely high with up to 4.42 sec in worst case (open MP with dual cores). The highest performance is with MPI model when the lowest is with open MP model. However, the complexity and the development life cycle of MPI and Hybrid MPI/OpenMP is much higher than OpenMP therefore the OpenMP can be a good choice with a low matrix multiplication to produce an acceptable performance.

Nevertheless, if we compare the performance of parallel model with a sequential program in a very low matrix size (16, 32 and 64) we notice that the differences are very low. Running parallel models require initializing some libraries and executing extra instructions to manage the parallel code. The time that is needed to run all of these instructions increases the total execution time. Therefore, both parallel and sequential program have almost the same performance. What is concluded from here is that instead of developing a parallel application for such issue, sequential application is worth using.

Speed up results: Based on the execution time experiment of both parallel and sequential program, the speed up can calculated as an additional performance measurement. The speed up does not only describe the performance the model itself but also the optimality of code and the amount of improvement when using extra processors. Speed (Table 11-16). The speed up value should be less or equal to the number of processors used to execute the

Table 11: Speed up (2 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.013860974	0.001100958	0.005315615
32	0.020190056	0.002381141	0.005352727
64	0.119461943	0.019614766	0.052291592
128	0.326429353	0.129034756	0.173079403
256	1.035993881	0.569176044	0.83754432
512	1.477328834	0.429244636	1.072734626
1024	1.601189419	0.416307694	1.233278394
2048	1.654420542	0.346520853	1.657905469
4096	1.693575027	0.570702222	1.9226709
8192	1.705673851	0.398847447	1.822549894

Table 12: Speed up (4 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.02325311	0.001604879	0.008290155
32	0.033870749	0.003471017	0.008348035
64	0.200408834	0.028592669	0.081553208
128	0.654428847	0.18809544	0.269932126
256	1.828167888	0.829694432	1.306221971
512	2.299560058	0.625714817	1.673021359
1024	2.600472464	0.606856489	1.923403092
2048	2.626604404	0.505127413	2.585645316
4096	2.742318692	0.831919159	2.998569641
8192	2.837553637	0.581404487	2.842422375

Table 13: Speed uP (8 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.048911714	0.003291639	0.016161616
32	0.071245369	0.007119127	0.016274452
64	0.421549616	0.058644147	0.158987567
128	1.37655723	0.385787585	0.526231315
256	3.845456593	1.701720212	2.546473135
512	4.837005639	1.28335386	3.261546691
1024	5.469959321	1.244675045	3.749664613
2048	5.524926506	1.036026633	5.040702486
4096	5.768325524	1.706283174	5.845696372
8192	4.968647304	1.192472469	5.541288063

Table 14: Speed up (16 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.06863418	0.004543389	0.024615385
32	0.09997334	0.0098264	0.024787243
64	0.5915293	0.080945443	0.242150295
128	1.93162063	0.53249554	0.801490773
256	5.396043929	2.348853251	3.878474467
512	9.787411138	2.771389835	7.267586498
1024	8.67558808	1.718002174	7.711027642
2048	7.752719452	1.430008592	7.977377632
4096	7.094263235	1.655151423	8.903445243
8192	9.375359928	1.345947915	10.43980797

program. Therefore, we notice that the speed value has increased when more processors are added. As mentioned before, the speed up also describes the amount of improvement in the performance when using parallel model. It is noticeable that the low complexity matrix (16, 32 and 64) has a very low speedup (<1) and that supports the conclusion that using parallel models with these size is not worthy.

The speedup of open MP is the lowest when the highest speedup is tested with Hybrid followed by MPI. The best speedup acquired from the experiments is when using Hybrid models with 64 processors. Therefore, we conclude that using MPI and Hybrid models is the best choice for both high complexity issues and high number of processors.

Table 15: Speed up (32 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.11820331	0.007680492	0.041025641
32	0.172176309	0.016611296	0.041312071
64	1.018744906	0.136836344	0.403583824
128	3.326679973	0.900171032	1.335817955
256	9.293186767	3.970680495	6.464124111
512	21.97431083	6.994492341	16.3207692
1024	19.21906836	5.904241771	12.518379403
2048	17.35190572	4.417395477	11.79562939
4096	15.94012002	3.981327405	13.83907541
8192	18.42423099	2.782435761	21.06634662

Table 16: Speed up (64 processors)

Size	MPI speed	OpenMP speed	Hybrid speed
16	0.11820331	0.007680492	0.041025641
32	0.172176309	0.016611296	0.041312071
64	1.018744906	0.136836344	0.403583824
128	3.326679973	0.900171032	1.335817955
256	9.293186767	3.970680495	6.464124111
512	21.97431083	6.994492341	16.3207692
1024	19.21906836	5.904241771	12.518379403
2048	17.35190572	4.417395477	11.79562939
4096	15.94012002	3.981327405	13.83907541
8192	18.42423099	2.782435761	21.06634662

Table 17: Efficiency (2 processors)

Size	MPI efficiency	OpenMP efficiency	Hybrid efficiency
16	0.6930487	0.0550479	0.26578075
32	1.0095028	0.11905705	0.26763635
64	5.97309715	0.9807383	2.6145796
128	16.32146765	6.4517378	8.65397015
256	51.79969405	28.4588022	41.877216
512	73.8664417	21.4622318	53.6367313
1024	80.05947095	20.8153847	61.6639197
2048	82.7210271	17.32604265	82.89527345
4096	84.67875135	28.5351111	96.133545
8192	85.28369255	19.94237235	91.1274947

Efficiency results: Efficiency is the third performance measurement in parallel system. This measurement is not related to the speed of the model. It describes the efficiency of the code and the system. The higher the efficiency is the better the code will be. In addition, measuring the efficiency of the system with different processors gives an indicator of it is worthy to add more processors to the system or not. For example, we have a system with 50% efficiency and 64 processor; then another 64 processors are added which turns the efficiency into 40%. In other words, the system performance cannot be improved by merely adding processors alone. The efficiency of the three models is increased when adding more processor. However, we notice that the OpenMP model has the lowest efficiency due to the compiler issue discussed in the beginning of this chapter (Table 17-22).

Table 19: Efficiency (8 processors)

Size	MPI efficiency	OpenMP efficiency	Hybrid efficiency
16	0.611396425	0.041145488	0.2020202
32	0.890567113	0.088989088	0.20343065
64	5.2693702	0.733051838	1.987344588
128	17.20696538	4.822344813	6.577891438
256	48.06820741	21.27150265	31.83091419
512	60.46257049	16.04192325	40.76933364
1024	68.37449151	15.55843806	46.87080766
2048	69.06158133	12.95033291	63.00878108
4096	72.10406905	21.32853968	73.07120465
8192	62.1080913	14.90590586	69.26610079

Table 20: Efficiency (16 processors)

Size	MPI efficiency	OpenMP efficiency	Hybrid efficiency
16	0.428963625	0.028396181	0.153846156
32	0.624833375	0.061415	0.154920269
64	3.697058125	0.505909019	1.513439344
128	12.07262894	3.328097125	5.009317331
256	33.72527456	14.68033282	24.24046542
512	61.17131961	17.32118647	45.42241561
1024	47.9724255	10.73751359	35.69392276
2048	48.45449658	8.9375537	47.9836102
4096	50.58914522	14.71969639	55.64653277

Table 21: Efficiency (32 processors)

Size	MPI efficiency	OpenMP efficiency	Hybrid efficiency
16	0.369385344	0.024001538	0.128205128
32	0.538050966	0.0519103	0.129100222
64	3.183577831	0.427613575	1.26119945
128	10.39587492	2.813034475	4.174431109
256	29.04120865	12.40837655	20.20038785
512	68.66972134	21.85778857	51.00240375
1024	41.30958863	9.075755534	29.74493563
2048	41.72470538	7.554360866	39.98634184
4096	43.56287506	12.44164814	46.37211066
8192	57.57572184	8.695111753	65.83233319

Table 22: Efficiency (64 processors)

Size	MPI efficiency	OpenMP efficiency	Hybrid efficiency
16	0.664893617	0.050403227	0.25
32	0.968491736	0.109011628	0.251745434
64	5.730440098	0.897988506	2.45933893
128	18.71257484	5.9073724	8.140140661
256	52.27417556	26.05759075	39.3907563
512	68.87804541	27.46385598	55.13955038
1024	74.35725952	19.05908663	58.00262448
2048	75.10446969	15.86415781	77.97336658
4096	78.41317509	26.12746109	90.42561575
8192	71.7612993	18.25973469	77.90429973

CONCLUSION

This study an experiment analysis, design and results are presented and analysed. First, we described the effects of using old compiler on the parallel system and amount of improvement that we get when using a newer version of compiler to compile the parallel codes. The results have shown that the GCC compiler version has a significant impact on the performance. Next, the three models OpenMP, MPI and Hybrid are examined with a different size of matrix and number of processors. With a low complex issue, there are two choices to write a high performance application which are sequential code with a very simple issues and OpenMP with a medium complexity issues. However with a high complexity issues, the MPI and Hybrid models are the optimal solution with a high performance, speedup and efficiency. All in all, the four parallel models produce a high performance when solving the issues in parallel.

REFERENCES

Chen, L.L. and Y. Wu, 2003. Aggressive compiler optimization and parallelization with thread-level speculation. Proceedings of the IEEE International Conference on Parallel Processing, October 6-9, 2003, IEEE, California, USA. isBN:0-7695-2017-0, pp: 607-614.

Denis, A., 2015. Pioman: A pthread-based multithreaded communication engine. Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), March 4-6, 2015, IEEE, Bordeaux, France isBN:978-1-4799-8491-6, pp: 155-162.

He, L., W. Shen, Y. Li, A. Shi and D. Zhao, 2010. MPI+Open MP implementation and results analysis of matrix multiplication based on row wise and column wise block-striped decomposition of the matrices. Proceedings of the IEEE 3rd International Joint Conference on Computational Science and Optimization (CSO), Vol. 2, May 28-31, 2010, IEEE, New York, USA. isBN:978-1-4244-6812-6, pp: 304-307.

Klawonn, A., M. Lanser, O. Rheinbach, H. Stengel and G. Wellein, 2015. Hybrid MPI-Open MP Parallelization in FETI-DP Methods. In: Recent Trends in Computational Engineering-CE2014, Mehl, M., M. Bischoff and M. Schafer (Eds.). Springer, Switzerland isBN:978-3-319-22996-6, pp: 67-84.

Kotobi, A., N.A.W.A. Hamid, M. Othman and M. Hussin, 2014. Performance analysis of hybrid Open MP-MPI based on multi-core cluster architecture. Proceedings of the IEEE International Conference on Computational Science and Technology (ICCST), August 27-28, 2014, IEEE, New York, USA. isBN:978-1-4799-3242-9, pp: 1-6.

Kuhn, B., P. Petersen and E. O'Toole, 2000. Open MP versus threading in C/C++. Concurrency Pract. Experience, 12: 1165-1176.

Munir, A., A. Gordon-Ross and S. Ranka, 2015. Modeling and Optimization of Parallel and Distributed Embedded Systems. John Wiley & Sons, New Jersey, USA.,.

Nichols, B., D. Buttlar and J. Farrell, 1996. Pthreads programming: A POSIX standard for better multiprocessing. O'Reilly Media Inc., Sebastopol, California, USA.

Rabenseifner, R., G. Hager and G. Jost, 2009. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, February 18-20, 2009, Weimar, Germany, pp: 427-436.

Sharma, R. and P. Kanungo, 2011. Performance evaluation of MPI and hybrid MPI+open MP programming paradigms on multi-core processors cluster. Proceedings of the IEEE International Conference on Recent Trends in Information Systems (ReTIS), December 21, 2011, IEEE, New York, USA., pp: 137-140.

Wu, C.C., L.F. Lai, C.T. Yang and P.H. Chiu, 2012. Using hybrid MPI and open MP programming to optimize communications in parallel loop self-scheduling schemes for multicore PC clusters. J. Supercomputing, 60: 31-61.