Int. J. Advance Soft Compu. Appl, Vol. 12, No. 3, November 2020
ISSN 2074-8523; Copyright © ICSRS Publication, 2020
www.i-csrs.org

Efficient Study on Evaluating Processor's Affinity in Multi-Core Architecture and Multi-Processor Systems

Saleh Ali Alomari

Software Engineering Department, Faculty of Science and Information Technology, Jadara University, Jordan
e-mail:omari08@jadara.edu.jo

Abstract

Processor's technologies are advancing at a fast pace, introducing higher computational power. However, this requires the presence of data closer to the processing unit in order to utilize the introduced computational power. Allowing processes to execute in any processor, results in exploiting the potential of the computational power of the processors, However, this diminishes the locality factor 1, in other words, every time a process is scheduled to execute, it is required to migrate its data to the new processor. This has two consequences: Firstly, the data should reside in the main memory all the time. Second, when the process finishes execution, it is required to update the copy of the data in the cache to the main memory, and to load the data in the next execution period. The significance of this paper comes from the fact that new systems are equipped with more processors and more cores per processor. In the near future, processors with many cores are going to be available commercially, hence, it is important to provide an insight on how to execute the processes on such processors. In this paper the processor Affinity helps in achieving better performance, as the process's data is expected to reside in the processors cache rather than the main memory. However, this might affect the performance, as limiting the number of processors that processes allowed to execute in, which might degrade the performance of the application. This paper evaluates the processor affinity (or core affinity) in order to measure the factors of locality and computational power on the processes execution.

Keywords: *Chip Multiprocessors, processor affinity, Multi-Processor Systems, SMPs*

1 Introduction

Currently, it is clear that Chip Multiprocessors (CMPs) or Multi-core processors are inevitable [1]. Processors manufacturers are providing Dual- and Quad-core processors over the past decade, and the plans are to provide processors that are equipped with 80 cores in the near future. In 2013, Intel announced the release of Knights Landing CPU, which is equipped with 72 cores and 16GB of DRAM stacked on the same chip, with the ability to process 500GB/s of data [2-3] In CMPs there are different cache hierarchies that enable the cores to store, process and share data. Generally, each core is associated with its own cache, named L1, and the cores are sharing the Last Level Cache (LLC). CMPs architecture is becoming more coherent with Shared Memory Processors (SMPs) architecture. Hence, CMPs and SMPs are used interchangeably in this paper. This also can be applied to the cores and processors, different cores in the same processor can be understood as different processors in the same system. For modern operating systems with multiprocessors, schedulers influence high store partiality by booking a procedure on a recently utilized processor at whatever point possible [4]. If the main process runs on a high affinity processor, it can already get the majority of its state in the cache and therefore run more efficiently [5]. Although exploitation of the cache affinity on Unicore multiprocessors are known to improve performance, their impact has not been studied on multicore processors. Understanding this effect is key to the development of efficient algorithms for multicore scheduling. There are a lot of these related algorithms (aimed at improving efficiency [6-9], reducing energy consumption [10] or improving thermal control [11-12]) operate by regular process migration between the cores CPU. Frequent migrations prevent the scheduler from exploiting affinity with the cache and can hurt performance. The affinity to the processor is a propensity for an application to run on a specific processor and to avoid migration. The soft affinity is known as, if the scheduler does not want to migrate a process to another Processor unless necessary. With hard affinity allocations in source code this can be overridden. Hard affinity APIs allow the developer to make specific allocations to a processor or processor party. We should determine where our code runs and on which processor our code runs that is not user-decidable which can be achieved by setting a CPU bit mask for each thread using sched_set affinity) (and sched_get affinity) (calling functions in LINUX[13].

Accordingly, application's architecture should be reviewed such that it utilizes the computational power that is introduced by CMPs. Cores computational power is relatively lower than single core processors, hence, applications that are not developed with concurrency in mind is expected to run at slower speed [14]. Threads, within the same application, are executed concurrently; generally, these threads share the data among each other. The location of the data at run time is determined based on the execution units that are used to execute these threads and the current status at runtime. Threads might be executed at the same processor at different cores, or at different processors [14]. Processors technology are advancing at high pace, hence, it is essential to introduce techniques that help in utilizing its

power. Processor affinity is one of these techniques. The significance of this paper comes from the fact that new systems are equipped with more processors and more cores per processor. In the near future, processors with many cores are going to be available commercially, hence, it is important to provide an insight on how to execute the processes on such processors. The paper written by "James Donald and Margaret Martonosi" entitled "An efficient Practical Parallelization Methodology for Multi-Core Architecture Simulation" presented a programming methodology that converts the uniprocessor simulators into multi-core simulators. The approach they use requires less development effort compared to other techniques in programming. [15].

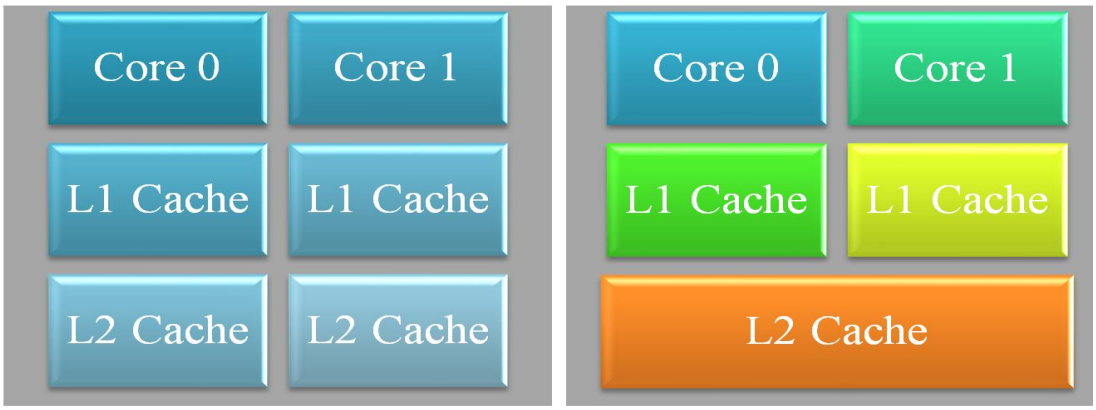
2 Related Work

Processor affinity or CPU sticking, empowers official a prepare or a string to a processor or a run of processors, such that the method or string is executed as it were on the assigned processor(s) instead of on any accessible processor on the framework. This is a modification of the standard central queue scheduling algorithm in a multiprocessing system. Processor affinity has the advantage that a process that was run on a given set of processors is expected to have its details (or data) remain in that processor, process's data in the cache memory. Scheduling a process to execute on the same processor improves its performance by reducing the possibility of cache misses, and the amount of time required to make the data available for that process; in other words, minimize the cost of data migration. Torrellas et al. proposed the impact of affinity-conscious scheduling on traditional (Unicore) multiprocessors [16][18]. According to their report, affinity-conscious scheduling decreased cache miss rates between 7 % to 36 % percent and improved efficiency by 10 percent. We followed the same goal but targeted multicore processors. We replied to a slightly different question, too. Unlike the Torrellas study which calculated the performance impact of a particular affinity-aware scheduling algorithm, we evaluated the upper limit on performance gains that could be achieved by exploiting affinity to the cache. Constantine et al. considered the effects of migrating a cycle between cores on a multicore processor [17] on output. Before migrating the process to that core (as opposed to leaving the caches cold), they studied performance effects of warming up instruction L1 and data caches on the new core. Warming up the caches builds synergy between the core and the mechanism being migrated.

2.1 Cache Memory

In CMPs, cores share data using the cache that resides inside the processor, accordingly, it is expected that a multi-threaded application in which threads share data among each other might not suffer delays, as the transfer of data within the same processor is significantly faster compared to sharing data using the main memory, as in the SMPs case. There are many cache memory hierarchies that is

developed for different purposes. Generally, the first level cache is dedicated for the core, while it is possible to share L2 and L3 cache among two or more cores. Figure 1 (a) and (b) shows a dual core processor, with L2 cache unshared and shared, respectively. A multi-threaded application running on the processing presented in figure 1 (b) threads can share their data using L2 cache; this is expected to significantly post the performance of the processor and the application.



(a) (b)

Figure 1 (a): Dual core processor, unshared cache. (b): Dual core processor, shared L2 cache

Figure 2 shows a quad core processor with committed L1 and L2 cache and then will shared (L3) cache. It is possible in this design to execute different threads and have the ability to store their details in L2 cache, allowing the thread re-scheduled on the same processor to perform faster than the case being re-scheduled on another core or another processor.

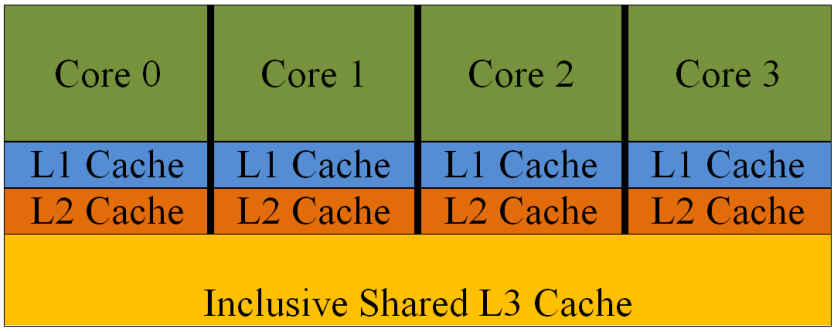


Figure 2: Quad core processor, shared L3 cache

2.2 Scheduling Mechanism

The Operating System (OS) scheduler handles the execution of different applications running on the same system. The scheduler manages the threads or processes allocation to the execution units based on different algorithms that is

designed for this purposes. To achieve the best performance, the scheduler tries to ensure that all execution units are allocated all the time. However, this might have negative influence on the performance. It is possible that the allocation of the processes might results in high level of data migration, which degrades the performance. Based on that, the scheduler should have a log that indicates which processes executed on a certain core or processor, trying to re-allocate those processes to the same core or processor. This is expected to enhance the performance of the execution. On the other hand, it is possible that between two successive allocations of a process that the process's details are overwritten, hence, it is possible for that process to be allocated on any available execution unit in the system. Generally, there are two main scheduling methodologies: partitioned and Global. In Global scheduling the process can be allocated to any core or processor on the system, hence, there is one scheduler that manages the scheduling of the processes in the system. In partitioned scheduling, the available execution units are divided into partitions, and each partition is managed by a single scheduler. The number of scheduler is determined by the number of partitions. In this case, a process allocated to a partition is not allowed to migrate to another partition while being executed. Partition scheduling can be viewed as a sort of processor affinity scheme, thus, enhancing the performance of the application by keeps the processes closer to their data. On the other hand, it is possible that at runtime a partition is able to finish execution of the process allocated to it faster than the other partitions, resulting in busy and idle partitions. This can be avoided using a balanced scheme of designing the partition and the allocation of processes to these partitions.

2.3 Image Histogram

Image histogram is a number of pixels in an image depending on its intensity; this can be presented as a graphical drawing. Image histogram can be developed for gray scale and colored images. In gray scale image the histogram represents the 256 degrees of the gray scale levels. In color images the histogram represents the distribution of the three basic colors; Red, Green, and Blue. Image histograms are an important tool for image processing as it can be used to view different features of the image as quantization noise. The calculation of the image histogram requires accessing every pixel in the image to read its value, and to develop the image histogram. Hence, the computational requirements of this function are dependent on the image size. The usage of the image histogram function is based on the fact that the image, which represents the data in our case, is divisible. Hence, it is possible to divide a certain image into multiple segments and to provide each segment to a thread. This ensures that the number of threads factor is highly tested and contributes to the overall execution time.

3 Research Methodology

According to the discussion in the previous chapter, cache size, localization, and the number of processing threads are the main factors in determining the performance of the systems. Hence, the design of the paper is considering these three factors.

3.1 Research Process

The research considered image processing application to perform the testing. This is based on the following facts:

- Data size varies: in image processing applications the size of the images might be very small or very large, which enables the testing of cache thoroughly.
- Data can be partitioned: in image process applications, the image can be partitioned and processed by different threads. This enables the testing of multithreading and the different loads for each thread.
- Computational prone: it is well-known that an image processing application is computational prone, which enables the testing process to have variance time requirements. The available machine for testing is a DELL PowerEdge 2950 server, equipped with two Intel XEON processors (the processor's model is E5345). Each processor is quad core with dedicated L1 data cache of 32KB and shared L2 cache of 8MB. The server has a main memory of 32GB.

3.2 Data Load Selection

Based on the available server, the images are selected to satisfy all testing requirements. Images' size varies between 10KB and 13MB, as shown in table 1.

Table 1: Selected image sizes and the purpose of each case

Image Size	Testing Requirement
10KB	Single load in L1 data cache
50KB	Single load in L2 cache and two loads in L1 data Cache
800KB	Single load in L2 cache and more than 10 time loads in L1 data Cache
3MB	Single load in L2 cache and more times loads in L1 data Cache compared to 800KB size
13MB	Two Loads in L2 data cache from main memory and more number of transfers between L2 and L1 data Cache.

Figure 3 (a) shows the image size of 10KB. Figure 3 (b) shows the image size of 50KB. Figure 3 (c) shows the image size of 800KB. Figure 3 (d) shows the image size of 3MB. Figure 3 (e) shows the image size of 13MB.

(a)

(b)

(c)

(d)

(e)

Figure 3 (a): image of size 10KB. (b): image of size 50KB, (c): image of size 800KB. (d): image of size 3MB and (e): image of size 13MB

3.3 Testing scenarios

All the tests are performed with one processor or two processors enabled. This two measure the degree to which the increased computational power may influence the performance.

3.3.1 Test scenario 1

In this test, the image size selected is 10KB, which can be loaded into L1 data cache in one time. The number of threads in this test varies between 2 and 16 threads, with increments of 2. Hence, the goal of this test is to measure the increased number of threads while having the same number of execution units and data load. In the case of the number of threads more than 8, the first 8 threads are assigned to the 8 cores and the rest of the threads are going to wait for the any of the cores to be available. Each thread is assigned the image in whole, so, the data size for each thread is 10KB.

3.3.2 Test scenario 2

In this test, the image size selected is 50KB. Since the size of L1 data cache is 32KB, and then it is required to perform the load of the image into L1 data cache in two steps, rather than 1 time as in 10KB image. As in test scenario 1, the number of threads varies between 2 and 16 threads with increments of 2. Each thread is assigned the image in whole, so, the data size for each thread is 50KB.

3.3.3 Test scenario 3

In this test, the image size selected is 800KB. The image is divided among the available threads. So, by increasing the number of threads the data size to be processed by each thread is going to decrease, as shown in table 2. The increase of the number of threads is going to utilize the available cores in the system; however, this requires more transfer of data between L2 cache and L1 data cache. As the number of threads increase, the threads are not guaranteed to be re-scheduled on the same core, or the same processor. Besides that, the increased numbers of threads will influence the contents of L1 data cache and it will be required to re-fill the cache every time a thread starts execution. In another case, the size of the image is selected to be 3MB. The only difference in this case is the number of times required for each thread to transfer the data to be processed from L2 cache into L1 data cache. For example, in the case of 8 threads, it is required to transfer the share of data of each thread, which is 100KB, from L2 cache into L1 data cache on four times, while in the case of 3MB, the share of each thread is 375KB, which requires 12 times of transfer to process that data.

3.3.4 Test scenario 4

In this test, the size of the image is 13MB. As the size of L2 cache is 8MB, then it is required to perform higher number of transfers between main memory and L2 cache. Besides that, as the number of threads increase, it is expected to have high level of variation in the execution time of each thread, which might increase the level of re-scheduling of the threads on different cores in other processor. This case increases the data migration between the cores, if the thread is rescheduled on the same core, or between the processors, if the thread is re-scheduled on a different processor.

4 Implementation and Modeling

The target of the paper is to measure the performance of the system; hence, the concentration is on the functionality of the application rather than its Graphical User Interface (GUI). An image processing functionality is selected to perform the evaluation process, which is the image histogram, discussed in section 2.3. A Different data load is used to measure the diversity of the factors that contribute to the performance. Accordingly, a Java application is developed to perform the required tasks. As Java provide versatile tools for image processing and execution environment control. There are many functions within the image processing. However, as the interest is in the testing rather than the functionality itself, the selection function is the processing of the image to build its histogram. The code shows the image histogram for the images with gray scale.

```
import java.awt.image.BufferedImage;
public class ProcAff extends Thread {
  int start=0;
  int end=0;
  BufferedImage img = null;
  public ProcAff(BufferedImage image, int s, int e){
  // assign the values to the local variables
    this.start=s;
    this.end=e;
    this.img=image;
  }
  public void run(){
    int rows=0;
    int cols;
    long starttime = System.nanoTime();
    int grey=0;
    int[] hist = new int[265];
    for(cols=0;cols<img.getWidth();cols++){
      for(rows=start;rows<end;rows++){
        grey= img.getRGB(cols, rows) & 0xFF;
        hist[grey]++;
      }
    }
    System.out.println((System.nanoTime()-starttime));
  }
}
```

5 Assumptions

All tests are performed on a dedicated server with no other application interference the testing process. Hence, L2 and L1 data cache of all the cores are utilized by the testing case. Global scheduling is used; hence, any thread can be scheduled or rescheduled on any available core.

6 Evaluation Process

The evaluation process is designed such that the different related factors are considered either individually or combined. The evaluation is designed to consider the following four factors:

- The data size to be processed: the amount of time required to perform the operation is dependent on the data size. Hence, different data loads are considered such that the execution of the threads achieves high diversity. The data size considered for this evaluation varies between 10KB and 5MB.
- The number of processing threads: for the same data load, if the number of threads is increased the data size for each thread is decreases, as shown in table 2. The available machine is equipped with 2 processor - Quad core each. Hence, the number of threads is selected between 2 and 16.
- The number of L1 cache refills: the available processors have L1 data cache of 32KB. The data size and by each thread. To ensure the coverage of all possible diversities, image sizes of 10KB, 50KB, 800KB, 2MB, 5MB, and 10MB are used. For each of these images different number of threads used to perform the execution; the selected number of threads are: 2, 4, 6, 8, 10, 12, 14, and 16. The selection of these values is based on the machine used to perform the evaluation. The machine used is equipped with two Intel E5345 processors. Each processor has a L2 cache of 8MB and L1 data cache of 32KB.

Table 2 shows the distribution of the images and threads. For example, in the test of image with size of 800KB, if the number of threads is 14, each thread is supposed to process 57KB of data.

Table 2: Processing threads data share according to the number of threads, all values are in KB, unless otherwise mentioned

	2	4	6	8	10	14	14	16
10	10	10	10	10	10	10	10	10
50	50	50	50	50	50	50	50	50
800	400	200	133	100	80	67	57	50
3M	1.5M	0.75M	500	375	300	250	214	188
14M	7 M	3.5 M	2.33 M	1.75 M	1.4 M	1.67 M	1 M	875

7 The Results

7.1 Small Size Images' Results

The two images selected for this test have sizes of 10KB and 50KB. The target of this test is to measure the influence of the time required to transfer the data between L2 and L1 data cache; in other words, the degree of the locality influence

on the execution time. In the 10KB case, the image can be loaded in single transfer to L1 data cache and processed by the thread, however, at least two transfers is required for the 50KB image. Figures 4 (a) and (b) show the results of performing the execution using different number of threads for the image of size 10KB, on 2 processors and 1 processor, respectively. The results in Figure 4 (a) present the number of threads increase the time required to perform the operations decrease. This is due to the fact that, by the first load of the thread the image is loaded into L2 cache and all subsequent accesses to the images are performed from L2 cache.

The results of Figure 4 (b) shown the number of threads increase the time required to perform the execution decrease. However, there is a limit to that, which is when the number of threads exceeds twice the number of cores. This is due to the time a thread is waiting in the queue for one of the cores to be available. The results indicate that the time required performing the execution one a single processor is significantly better than 2 processors, this is due to the facts that in the 2 processors case, the threads might be re-scheduled on a different processor each time, which requires migration of data between the processors. While in the single processor case, all the threads are scheduled in the same processor and the data resides in L2 cache all the time.

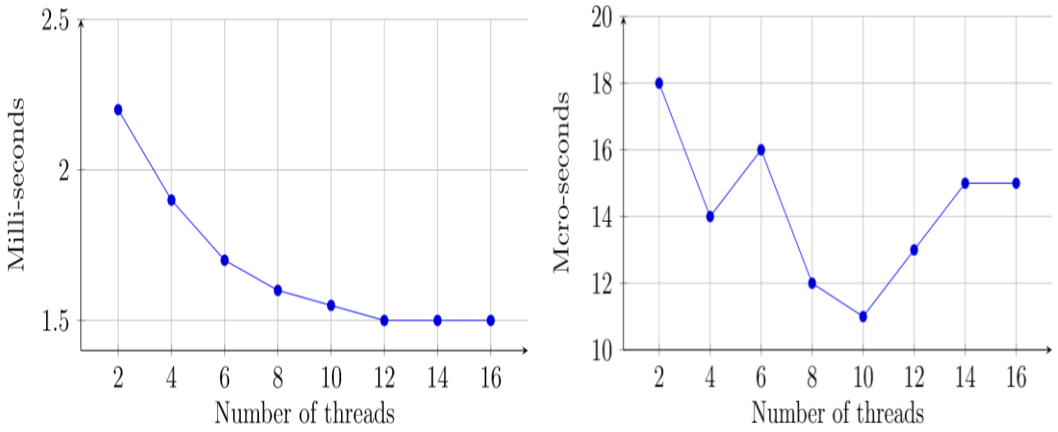


Figure 4: (a) Image of 10KB - 2 processors. (b): Image of 10KB - 1 processor

Figures 5 (a) and (b) show the results of performing the execution using different number of threads for the image of size 50KB, on 2 processors and 1 processor, respectively. The results in both cases indicate the number of threads increase the time required to perform the operations decrease. As noticed the time required to perform the execution on the single processor case is significantly lower than the 2 processors case, which is due to no need in migrating the data between the processors in the case of rescheduling the threads among the processors.

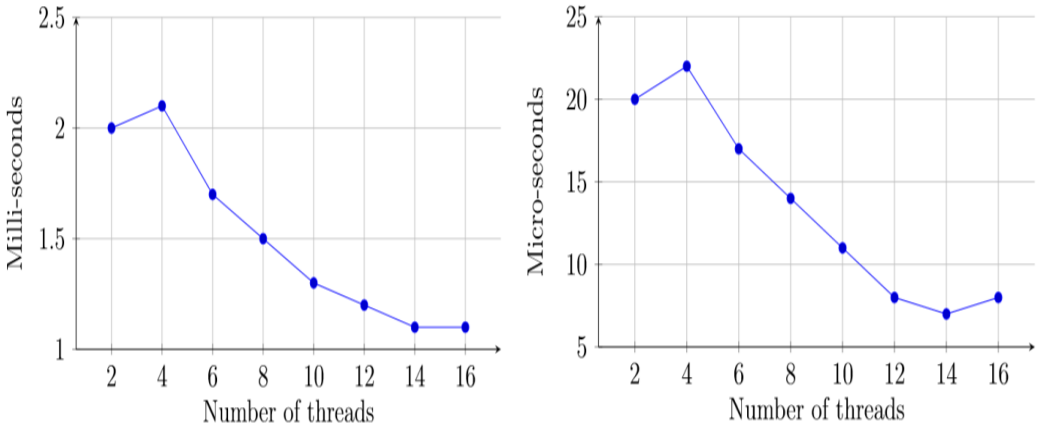


Figure 5: (a) Image of 50KB - 2 processor. (b): Image of 50KB - 1 processor.

7.2 Medium Size Images' Results

Figures 6 (a) and (b) show the results of performing the execution using different number of threads for the image of size 800KB, on 2 processors and 1 processor, respectively. Generally, the results in both cases indicate the number of threads increase the time required to perform the operations decrease.

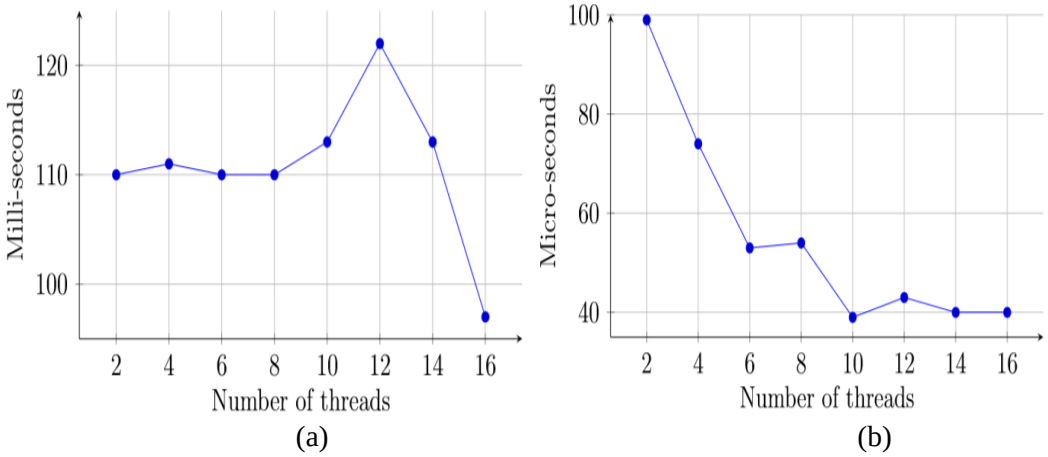


Figure 6: (a) Image of 800KB - 2 processor. (b): Image of 800KB - 1 processor

The results in figure 6 (a) shown the number of threads did not exceed the number of cores the execution time is relatively constant. However, as the number of threads exceeds the number of cores there is an increase in the execution time, this is due to the waiting time and the possible requirements of migrating the data among the processors in the case of rescheduling the thread on a different processor. It is

noticed that the execution time significantly enhanced after certain number of threads. This is due to that the data size to be processed by each thread is decreasing, hence, the migration requirement significantly decreases. In the case of single processor, figure 6 (b) the execution time decreases as the number of threads increase, as there is no migration required when the thread is rescheduled.

Figures 7 (a) and (b) show the results of performing the execution using different number of threads for the image of size 3MB, on 2 processors and 1 processor, respectively. Generally, the results in both cases indicate the number of threads increase the time-required to perform the operations decrease.

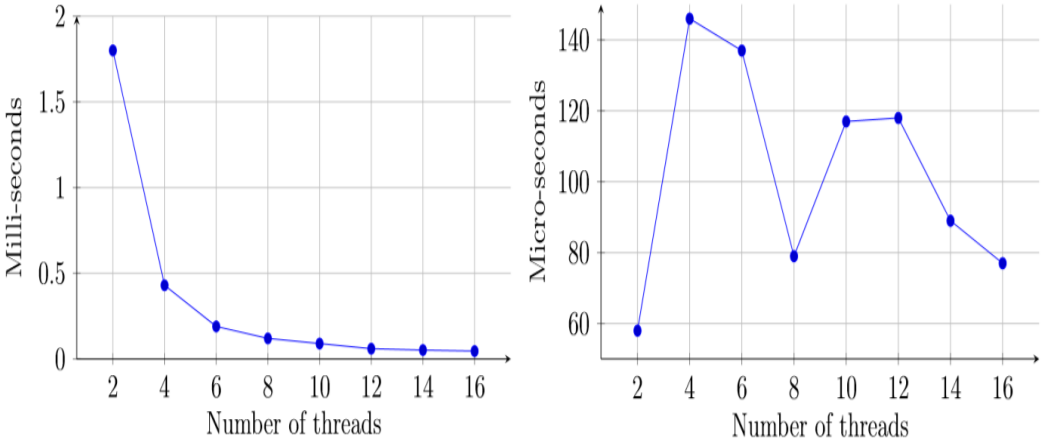


Figure 7: (a) Image of 3MB - 2 processor. (b): Image of 3MB - 1 processor

The results in figure 7 (a) indicate the number-of-threads increase the execution-time decrease. In this case, the decrease in the data size to be processed by each thread compensates for the waiting time and the time required performing any migration if the thread is rescheduled on a different processor. The results in figure 7 (b) indicated that the execution time for the single processor is significantly better that the execution time for the 2 processors. This is due to the locality of the data for the threads, as the image can be loaded into L2 cache and used by all the threads.

7.3 Large Size Image Results

Figures 8 (a) and (b) show the results of performing the execution using different number of threads for the image of size 13MB, on 2 processors and 1 processor, respectively. Generally, the results in both cases present the number of threads increase the time required to perform the operations increase. The results in figure 8 (a) indicate the number of threads exceeds the number of cores the execution time increases significantly.

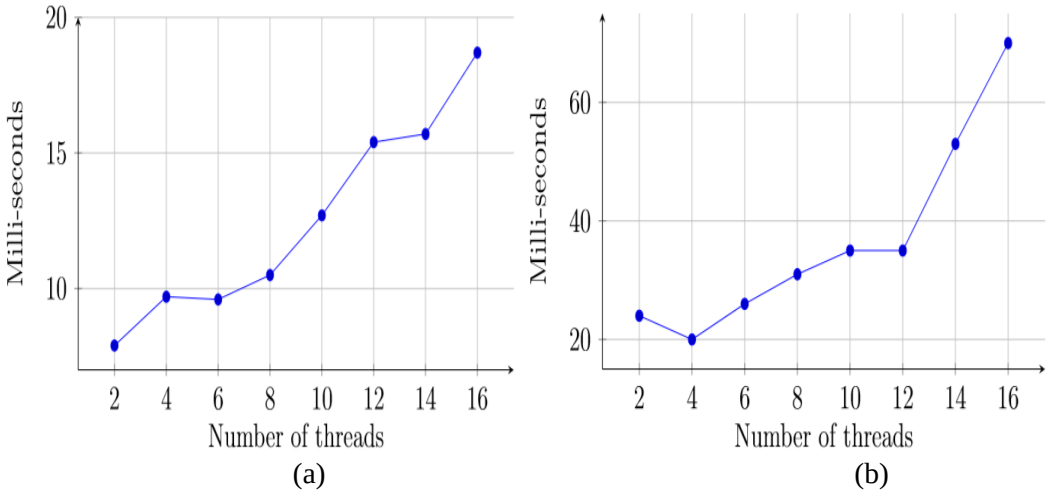


Figure 8: (a) Image of 13MB - 2 processor. (b): Image of 13MB - 1 processor

The increase in the number of threads is combined with a decrease in the data size should be processed by each thread, however, this did not enhance the performance, as this results in more transfer of the data between the main memory and L2 cache.

8 Results, Analysis and Discussions

The test cases are designed to ensure that different factors in the execution environment are challenged, which are: the data size, the number of execution units, the number of threads, and the transfer of data required, either within the processor or among the processor and the main memory. Transfer of data within the processor has no noticeable influence on the execution time, as the transfer of data within the processor can be performed instantly. The machine used to perform the evaluation is able to transfer 5GB/sec internally. The number of threads used to perform the execution has influence on the execution time, however, there is no specific, thus, it should be discussed along with the data size should be processed by each thread and the number of execution units. The transfer of data between the processor & memory is the factor that has the most negative influence on the execution time; in other words, the miss penalty is very expensive and cannot be tolerated. This paper intends to measure the performance of SMPs by executing application with different loads in order to determine the effect of locality and computational power. The machine that is used to perform the testing is a dedicated machine, hence, there is no influence of any other application executing concurrently on the same machine. The server runs Linux Mageia 5, with Global scheduling; hence, there is no update or change to the scheduling algorithm.

ACKNOWLEDGEMENTS

The author would like to thank the Faculty of Science and Information Technology (FSIT), Jadara University for their support and assistance with this project and for their appreciation of the benefits to be gained from independent research. Special thanks also go to the late Dr. Imad Al-Aqili, may God have mercy on him and make him spacious in his paradise, for the advice, counsel, and expertise.

References

[1] H. B. Jang, I. Yoon, C. H. Kim, S. Shin, and S. W. Chung (2009). The impact of liquid cooling on 3d multi-core processors. In 2009 IEEE International Conference on Computer Design, pages 472–478, Oct 2009.

[2] https://www.extremetech.com/extreme/171678-intel-unveils-72-core x86-knights-landing-cpu-for exascale-supercomputing.

[3] Y. Ben-Asher (2012). Multicore Programming Using the ParC Language. Undergraduate Topics in Computer Science. Springer London, 2012.

[4] McDougall, R., Mauro, J. (2006): SolarisTM Internals: Solaris 10 and OpenSolaris Kernel Architecture. Prentice Hall, Englewood Cliffs

[5] Torrellas, J., Tucker, A., Gupta, A. (1995): Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. Journal Of Parallel and Distributed Computing 24, 139–151

[6] Becchi, M., Crowley, P. (2006): Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In: Proceedings of the Conference on Computing Frontiers

[7] Fedorova, A., Vengerov, D., Doucette, D.: (2007) Operating System Scheduling On Heterogeneous Multicore Systems. In: Proceedings of the PACT 2007 Workshop on Operating System Support for Heterogeneous Multicore Architectures

[8] Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N., Farkas, K.: (2004) Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance. In: Proceedings of the 31st Annual International Symposium on Computer Architecture

[9] Snavely, A., Tullsen, D.M. (2000): Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)

[10] Kumar, R., Farkas, K., Jouppi, N., Parthasarathy, R., Tullsen, D.M. (2003): Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture

[11] Coskun, A., Rosing, T. (2007): Temperature aware task scheduling in MPSoCs. In: Proceedings of the DATE

[12] Powell, M.D., Gomaa, M., Vijaykumar, T.N. (2014): Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In: Proceedings of the ASPLOS

[13] Mike Aderson, "Understanding and Using SMP/MultiCore Processors, New Hardware and How to use it", The PTR Group, Inc. 10/28/2018.

[14] D. Morley and C.S. Parker (2017). Understanding Computers: Today and To morrow, Comprehensive. Cengage Learning,

[15] Tomasz Bawej., (2015), "Achieving High Performance With TCP over 40 GbE on NUMA Architectures for CMS Data Acquisition", IEEE TRANSACTIONS ON NUCLEAR SCIENCE, February 26, 2015.

[16] V. Doraisamy, et al., "Video on Demand Caching System using NIPBCS over Mobile Ad Hoc Network,"International Journal of Digital Content Technology and its Applications,vol. 5, no. 6,pp. 142-154, 2011

[17] Torrellas, J., Tucker, A., Gupta, A. (1995): Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. Journal Of Parallel and Distributed Computing 24, 139–151

[18] Constantinou, T., Sazeides, Y., Michaud, P., Fetis, D., Seznec, A., (2015). Performance Implications of Single Thread Migration on a Chip MultiCore. In: Proceedings of the Workshop on Design, Architecture and Simulation of Chip Multi-Processors

Notes on contributor

Dr. Saleh Ali K. Alomari obtained his MSc and PhD in Computer Science from Universiti Sains Malaysia (USM), Pulau Penang, Malaysia in 2008 and 2013 respectively. He is a lecturer at the Faculty of Science and Information Technology, Jadara University, Irbid, Jordan. He is Assistance Professor at Jadara University, Irbid, Jordan 2019.

He is a Vice Dean of the Faculty of Science and Information Technology and Director of the E-Learning Center 2020. He was Assistant Dean of the Faculty of Science and Information Technology for Student Affairs & Quality Assurance. Head of the Software Engineering Department, 2019/2020. He was a head of the Software Engineering, 2019-2020 and Computer Network department at Jadara University, 2014 until 2016. He is the candidate of the Multimedia Computing Research Group, School of Computer Science, USM. He is managing director of ICT Technology and Research and Development Division (R&D) in D&D Professional Consulting

Company. He has published over 50 papers in international journals and refereed conferences at the same research area. He is a member and reviewer of several international journals and conferences (IEICE, ACM, KSII, JDCTA, IEEE, IACSIT, etc). His research interest are in area of multimedia networking, video communications system design, multimedia communication specifically on VOD system, P2P Media Streaming, MANETs, caching techniques and for advanced mobile broadcasting networks as well.