Delft University of Technology
Parallel and Distributed Systems Report Series

A Detailed Performance Analysis of the OpenMP Rodinia Benchmark

Jie Shen, Ana Lucia Varbanescu
{j.shen,a.l.varbanescu}@tudelft.nl

Report number PDS-2011-011

TU Delft PDS

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.ewi.tudelft.nl

Information about Parallel and Distributed Systems Group:
http://www.pds.ewi.tudelft.nl/

J. Shen, A.L. Varbanescu

Performance Analysis of Rodinia Benchmark

Abstract

The development of multi-core processors has aroused extensive interests in parallel programming model research. OpenMP, as one of the most widely used shared memory programming model, has been proposed and improved for more than ten years. In our work, we carry out a set of intensive performance experiments by using the OpenMP Rodinia benchmark. We cover eleven different types of applications and conduct the experiments on three multi-core CPUs. In order to investigate the OpenMP scalability and the best performance that OpenMP can achieve, we vary the scale of datasets in each application and deploy different numbers of OpenMP threads for each experiment run. We use IQR method to guarantee a more reliable analysis. According to our results, we find that OpenMP shows diversified performance behaviors among different applications, platforms, and datasets. For most applications, it performs reasonably and scales well, reaching the maximum performance around the number of hardware cores/threads of the underlying hardware platforms. The results will be used for further research on comparing programming models on multi-cores.

Contents

List of Figures

List of Tables

1 Introduction

Multi-core processors has become mainstream in parallel computing. With their quick development, programmers have begun to consider the corresponding parallel programming models from multi-angles. Instead of focus entirely on performance, ease of programming and cross-platform portability also become increasingly important. Therefore, several parallel programming models have been either refurbished or created to address these new challenges [1], such as CnC [2], ArBB [3], or OmpSs [4].

However, despite these new, well-built programming solutions, OpenMP is still one of the dominants. OpenMP is a simple, traditional shared memory parallel programming model, with a solid background in the HPC community. In this paper, we study the performance behavior of OpenMP programming model on three multi-core platforms by using eleven applications from the Rodinia benchmark suite [5]. We run three datasets of different sizes for each OpenMP application, and we vary the number of OpenMP threads for each experiment case. Our aim is to (1) understand the OpenMP scalability; (2) find the best OpenMP performance per application, platform, and dataset.

After a set of experiments and analysis, we find that the OpenMP performance behaviors are diverse. OpenMP scales well for most applications, and the best performance always happens around the number of hardware cores/threads of the multi-core platforms. Since OpenCL is another popular programming model among multi-cores, having the benefit of cross-platform portability, we plan to use the results for further investigation on the comparison of OpenMP and OpenCL programming models.

The rest of the paper is organized as follows: We introduce OpenMP and Rodinia benchmark suite in Section 2 and 3. Section 4 illustrates our experiment methodology, followed by Section 5 with a thorough performance analysis. Finally, in Section 6 we draw conclusions and discuss future work directions.

2 OpenMP

OpenMP [6] comprises a set of compiler directives and a library, originally targeted at structured parallelism in loops. In OpenMP, programmers annotate sequential C, C++ or Fortran code by a set of compiler directives for parallel execution. Therefore, sequential algorithms are parallelized incrementally, and without major restructuring. OpenMP operates in the fork-join model. Programmers start an `omp parallel` section in which they can for example annotate a `for` loop. The iterations will run in parallel and join at the end of the `for`. This works best for independent iterations, but it is possible to have different forms of synchronization. Reduction operators are built in and programmers are able to define atomic operations and critical sections. OpenMP parallelism granularity can be controlled manually by adjusting the number of OpenMP threads in combination with a scheduling type, such as `static` or `dynamic`, which insures a coarse-grained parallelism. OpenMP supports both task and data parallelism. In OpenMP 3.0, programmers can define tasks. Tasks are similar to the `sections` statement that defines that multiple threads should execute different tasks in parallel, but tasks are more dynamic, high-level and allow nesting. With a `task` construct, programmers can define a block of code of which the execution can be deferred. In contrast to sections, tasks are not bound to a specific thread, which means that a thread can execute multiple tasks. Tasks can also be `untied` which means that tasks can be suspended and resumed by a thread. The directive-based approach and coarse-grained parallelism make this model easy for programming existent and new applications on multi-core hardware platforms.

3 Rodinia Benchmark Suite

Rodinia benchmark suite is designed for research in heterogeneous parallel computing [7]. In order to help computer science researchers to have an insight in emerging hardware platforms, this benchmark suite are implemented for both multi-core CPUs and GPUs using three different parallel programming models - OpenMP, CUDA, and OpenCL. To address the diversity characteristics, the applications in the benchmark suite are

selected carefully to cover different types of application behaviors according to the Berkeley's dwarfs [8], therefore they exhibit various types of computations, data access patterns, problem partitions, and optimizations. In each application, different input sizes can be specified for various uses. In our experiments, we target the OpenMP part of the benchmark suite: We choose eleven OpenMP applications. We run three datasets of different sizes for each application, and we vary the number of OpenMP threads for each test case, in order to (1) understand the OpenMP scalability; (2) find the best OpenMP performance per application, platform, and dataset.

4 Methodology

In this sections, we present our experimental setup, including the chosen platforms, datasets and the number of OpenMP threads. We also discuss the OpenMP performance measurement method.

4.1 Experimental Setup

Platforms We choose three different multi-core hardware platforms provided in DAS-4 [9] to carry out the performance experiments.

- N8 - 2.40GHz Intel Xeon E5620 dual quad-core platform with 2-way hyper-threading.
- D6 - 2.67GHz Intel Xeon X5650 dual six-core platform with 2-way hyper-threading.
- MC - 2.10GHz AMD Opteron 6172 Magnycours platform consisting of quad twelve-core processors.

Applications and Datasets We choose eleven different applications from the Rodinia benchmark suite (Table 1). For each application, we choose three different datasets (varying dataset sizes) as inputs, aiming to find out if the performance trend in each application is preserved or affected by the scale of datasets. Besides, for the same dataset, whether different platforms behave similarly or not is also observed.

Table 1: The eleven Rodinia OpenMP applications we choose

Application	Dwarf
BFS	Graph Traversal
HotSpot	Structured Grid
K-means	Dense Linear Algebra
CFD	Unstructured Grid
LUD	Dense Linear Algebra
NW	Dynamic Programming
SRAD	Structured Grid
Streamcluster	Dense Linear Algebra
Particle Filter	Structured Grid
Back Propagation	Unstructured Grid
PathFinder	Dynamic Programming

Number of threads We investigate the scalability of our OpenMP benchmarks by varying the number of OpenMP threads (software threads) from two to a number at least equal (but usually larger) to the number of hardware threads the target platform has. Another goal is to find the best OpenMP performance per application, platform and dataset. Table 2 summarizes the number of OpenMP threads we have used for each multi-core platform.

Table 2: The number of hardware threads and OpenMP threads we use on the three hardware platforms

Platform	Abbreviation	Number of Hardware Threads	Number of OpenMP Threads
Dual quad-core	N8	16	2 4 8 16 24 32 48 64
Dual six-core	D6	24	2 4 6 8 12 16 24 32 36 48
Magnycours	MC	48	2 4 6 8 12 16 24 32 36 48 60 64 72 96

4.2 Performance Measurement Method

To test and analyze the parallel computing capability of our three multi-core platforms, we run each application with several numbers of OpenMP threads and measure the execution time (unit: ms) for each run. We measure only the compute intensive, OpenMP-enabled sections of each application. We use `gettimeofday()` to record the elapsed time from the start point to the end point of the parallel section.

Furthermore, for each number of OpenMP threads, we run ten consecutive tests to determine a more reliable result. Finally, we analyze the ten timing samples and report the minimum, first quartile, median, third quartile, maximum execution time. We do this because we have noticed several unstable executions, which show abnormal timing behavior.

5 Perfomance Analysis and Comparison

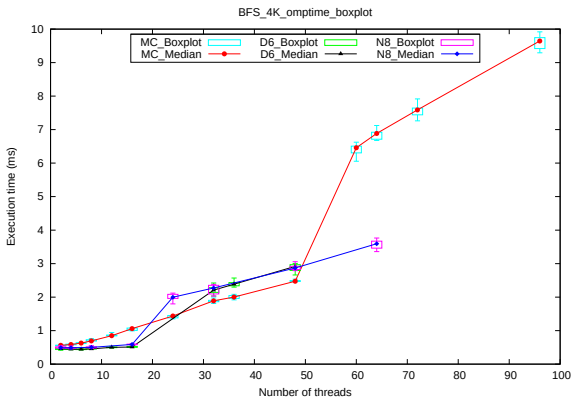
In this section, we analyze the performance of eleven applications from the OpenMP Rodinia benchmark, and make a thorough comparison among different platforms and datasets.

5.1 BFS

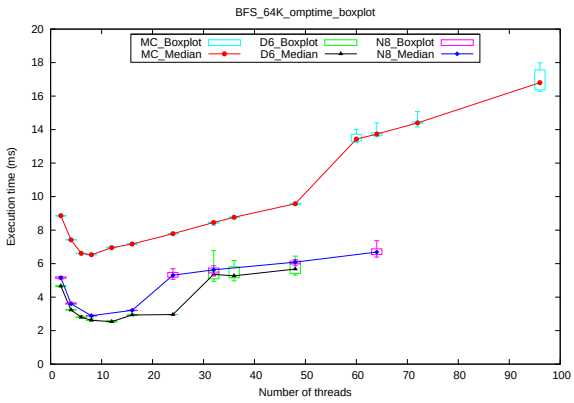
BFS is the Breadth-First Search algorithm [10] implemented in parallel which can be used in many graph related applications. For BFS, we have three graph datasets of 4K nodes, 64K nodes and 1M nodes.

In BFS, we observe that different platforms have a similar performance trend for the same dataset. Figure 1(a) shows the performance of the three platforms when using the 4K-node dataset. In this figure, all three curves have a general increasing trend. The execution time levels off (or increases slowly on the MC platform) when the number of OpenMP threads is smaller than the number of hardware threads, and increases sharply afterwards. In Figure 1(b), when the dataset has 64K nodes, the execution time of each platform drops down to the lowest point at 8, 12, 8 threads on N8, D6, MC, respectively, and then rises slightly. In Figure 1(c), when the number of nodes processed is 1M, the execution time of the N8, D6, MC platform decreases quickly from 2 threads to its lowest point at 8, 12, 16 threads, respectively, and then remains stable (increases or decreases slightly) as the number of OpenMP threads increases.

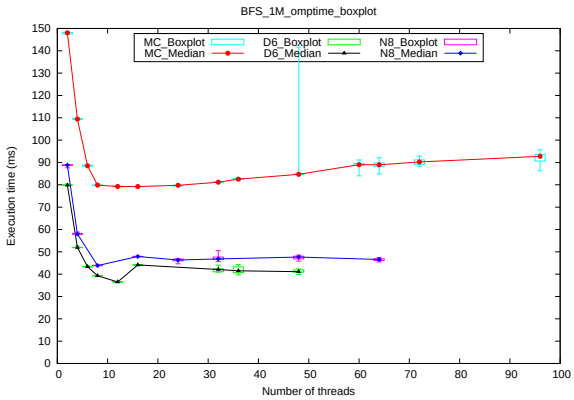
From these figures, we find that the optimal numbers of OpenMP threads that should be used for the BFS parallel section are 8, 12, 8/16 on the MC, D6, N8 platforms, respectively, especially when the scales of dataset are large. For the N8 and D6 platforms, the optimal numbers are half of their numbers of hardware threads, and equal to their numbers of cores. For the MC platform, although the optimal number of threads is much less than the number of hardware threads (also the number of cores), the corresponding execution times are relatively close. We also notice there are sharp increases from 48 to 60 threads on the MC platform, from 16 to 32 threads on the D6 platform, and from 16 to 24 threads on the N8 platform when using the 4K-node dataset and the 64K-node dataset. Actually, we test and discover that these sharp increases happen at (`#hardware threads + 1`) threads. This happens because using more threads than the number of hardware threads, we overload the multi-core platforms with thread switch overhead, which will shadow the advantage brought by multi-threading. An interesting fact on the N8 and D6 platforms with the 1M-node dataset is that after falling to the lowest points, the execution times first increase and then present a decreasing trend. We think that the



(a) 4K-nodes



(b) 64K-nodes



(c) 1M-nodes

Figure 1: The execution time of BFS on the three platforms using three datasets: (a) 4K-nodes, (b) 64K-nodes, (c) 1M-nodes.

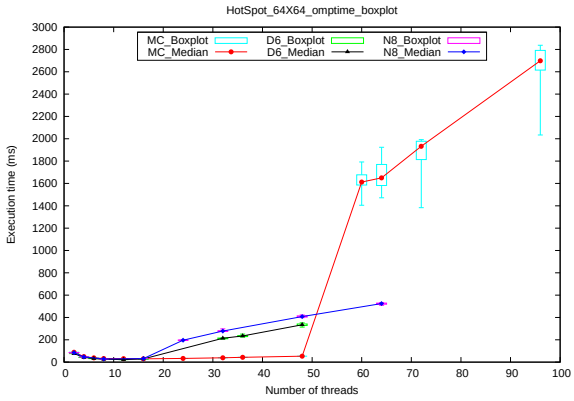
performance gain in execution time may be introduced by the cache, since one memory access brings more data into the cache, to be used by more threads as the number of threads increases.

Apart from that, we observe that there is an abnormal spike on the D6 platform at 24 threads (not shown in the figures). When using 24 threads, the execution time is one or two orders of magnitude larger[1] than using the other numbers of threads. This situation appears throughout the whole Rodinia benchmark experiments, independent of the applications and datasets used. We believe something is wrong on the D6 platform itself.

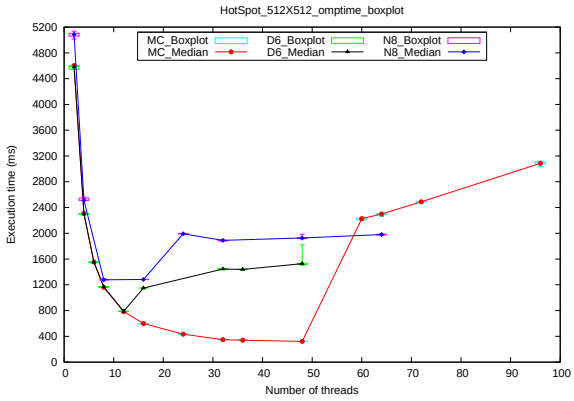
5.2 HotSpot

HotSpot is a 2D transient thermal modeling kernel [11], which computes the final state of a grid of cells when given the initial conditions (temperature and power dissipation per cell). The application iteratively updates the temperature values in all cells in parallel, and usually stops after a given number of iterations. For HotSpot, we have three datasets, for grids of sizes 64×64 (4K), 512×512 (256K), and 1024×1024 (1M) cells.

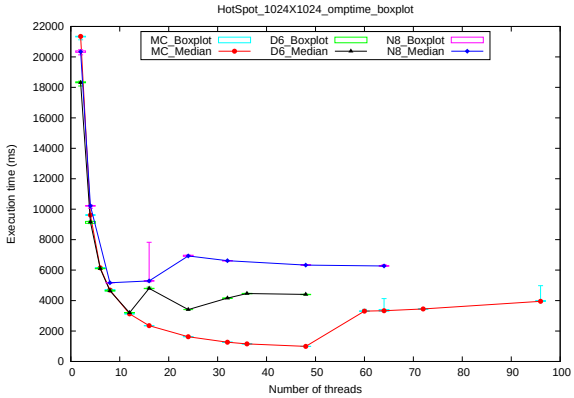
[1] Sometimes one out of ten timing samples has a normal execution time. We consider this normal result is correct result and use it for plotting in those cases.



(a) 64 × 64



(b) 512 × 512



(c) 1024 × 1024

Figure 2: The execution time of HotSpot on the three platforms using three grids: (a) 64 × 64, (b) 512 × 512, (c) 1024 × 1024.

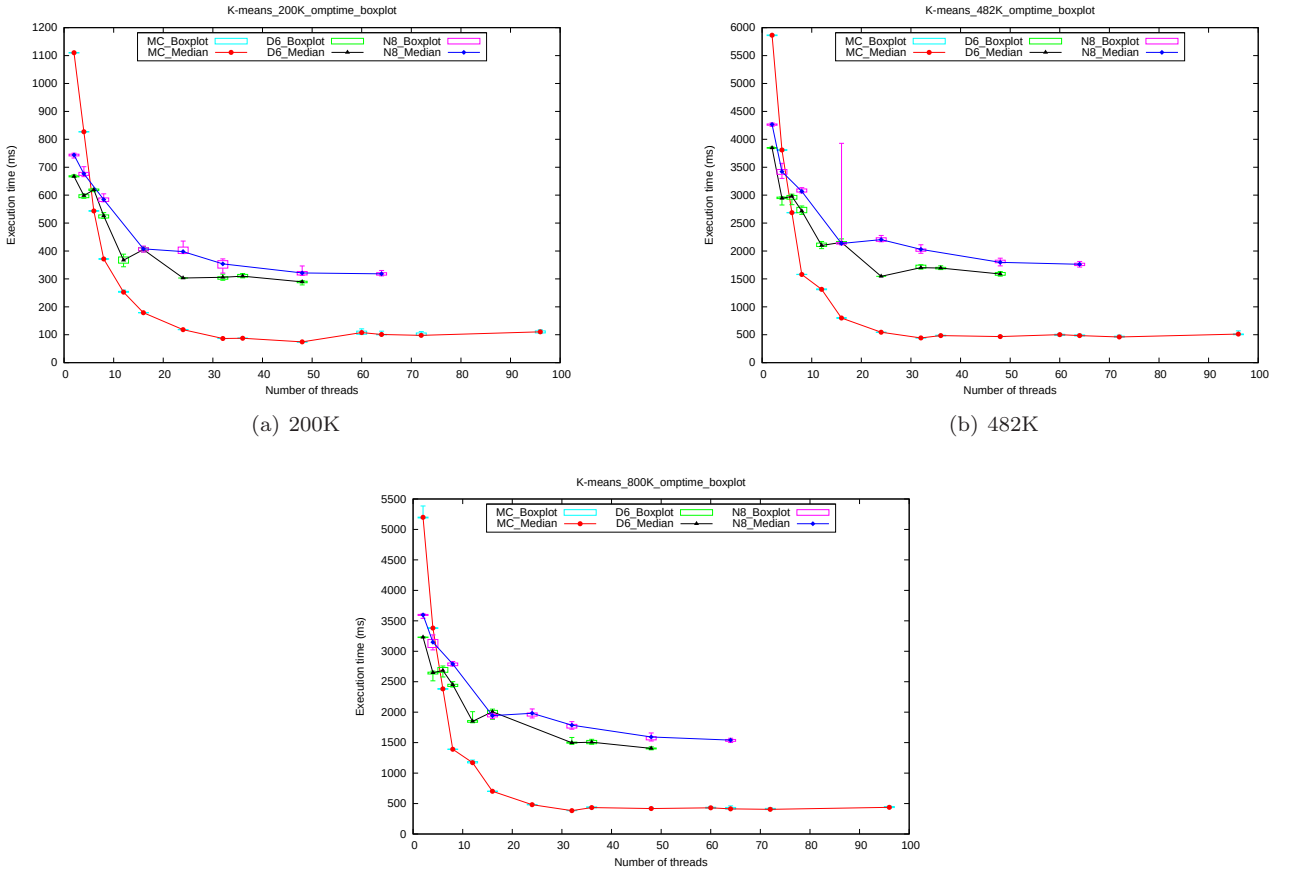
In HotSpot, we find that platforms perform similarly for the same dataset, but the achieved performance is different. In Figure 2(a) of the smallest dataset, each of the three curves goes down and then increases at a steady rate (almost constant) when the numbers of OpenMP threads are smaller than the number of hardware threads, and rises dramatically after that point, especially on the MC platform. For the largest dataset (Figure 2(c)), the execution time of the MC, D6, N8 platform decreases proportionally as the number of threads increases, reaching the minimum value at 48, 12, 8 threads, respectively. From these points onwards, it increases on MC, fluctuates on D6, and first increases then decreases on N8. Figure 2(b) shows the case of 512 × 512 cells. We see that the performance trends are more or less the same as those in Figure 2(c).

According to the results, we find that the optimal numbers of OpenMP threads for HotSpot are 48, 12, 8 (equal to/half of/half of the number of hardware threads) on MC, D6, N8 respectively, only except that the optimal is 16 threads for 64 × 64 dataset on MC. A point worth mentioning is the first half parts of the curves in Figure 2(a) when the numbers of OpenMP threads used are smaller than the number of hardware threads. As the scale of dataset is small, the workload per thread is relatively small, so even using a small number of OpenMP threads can achieve a good performance. When enlarging the number of threads, the overheads such as synchronization between multiple threads are introduced. Thus, the execution time difference between

different numbers of OpenMP threads is not apparent. Besides, the sharp increases re-emerge in HotSpot from 48 to 60 threads on MC, and from 16 to 24 threads on N8. On D6, the increases happen from 12 to 16 for the 512×512 dataset and the 1024×1024 dataset, and from 16 to 32 threads for the 64×64 dataset. We think the cause of the sudden growth is the same as that in the BFS application.

5.3 K-means

K-means (KM) is a clustering algorithm using mean based data partitioning method [12]. It contains dense linear algebra calculations and has a lot of data parallelism to be exploited. We have three datasets of 200K, 482K, 800K objects for K-means.



(a) 200K

(b) 482K

(c) 800K

Figure 3: The execution time of K-means on the three platforms using three datasets of sizes: (a) 200K-objects, (b) 482K-objects, (c) 800K-objects.

As we can see in Figure 3, K-means has different performance behaviors among different multi-core platforms when using the same dataset, but has similar performance behaviors per multi-core platform when using different datasets. The three execution time curves of the MC platform has a nearly ideal decreasing trend compared to the other two platforms. It drops fast till its lowest point (32 threads for the 482K and 800K dataset, 48 threads for the 200K dataset), then flatten out, showing a sign of performance saturation. This means even we

use more software threads in this application, it can not achieve a better performance. Because we also bring in penalties, such as thread switch and cache miss, when adding threads. Turing to the N8 and D6 platform, the general performance trends are both decreasing, but less steep than MC. The execution time of each curves falls quickly in the first half part (`#OpenMP threads` $<$ `#hardware threads`), while decreases much slowly in the other half part. We think the reason is the same as in the MC platform case. The minimum execution time of N8 and D6 are at the points of 48 and 64 threads, respectively, showing that these two multi-core platforms still have not reached their capability limits in the end.

An interesting point we note is for all three datasets, MC performs worse than N8 and D6 only at 2 and 4 threads, and surpasses the other two when the number of OpenMP threads are equal or greater than 6. Beyond that, an abnormal situation in K-means is that there are some leaps at 6, 16 threads on the D6 platform, making the three curves of D6 sawtooth shapes. For the leaps at 16 threads, we consider this is because 16 is not an integer multiple of the number of cores (12), which makes the workload partition per core unbalanced. As to the leaps at 6 threads, we are further researching processor profiling information to find out the causes.

5.4 CFD Solver

CFD Solver is an unstructured grid finite volume solver for the 3D Euler equations for inviscid, compressible flow [13]. For CFD, we use the single precision version with redundant flux computation scheme, which has reduced memory latency and high arithmetic intensity. We have three datasets of 97K, 193K, and 0.2M[2].

According to the results shown in Figure 4, we find that performance behaviors among different multi-core platforms and different datasets are substantially the same, and it is similar to that of a compute-intensive application. On MC, the best performance happens at the number equal to the number of hardware threads/cores (48). After that, it first undergoes a jump from 48 to 60 threads, then decreases slightly at 64 threads, and keeps increasing at a slow rate till the end. We think the mutual effects of thread switch overhead and less memory access benefit result in this performance behavior. On D6, the lowest points of execution time are different with different datasets. For the 193K dataset and the 0.2M dataset, they are 12 and 24 threads, respectively (half of and equal to the number of hardware threads). We believe this two situations are more of less the same, because we have not got a correct result at 24 threads for the 0.2M dataset. However, the lowest point shifts to 16 threads with a small peak at 12 threads for the 97K dataset, due to the unstable property of small scale dataset. On N8, the three curves all drop down till the number of threads equals to the number of hardware threads (16). After an apparent increase at 24 threads, they keep a mild decreasing trend. We think the causes of the performance behavior on N8 are the same as the causes on MC, but we see the outcomes vary among different platforms.

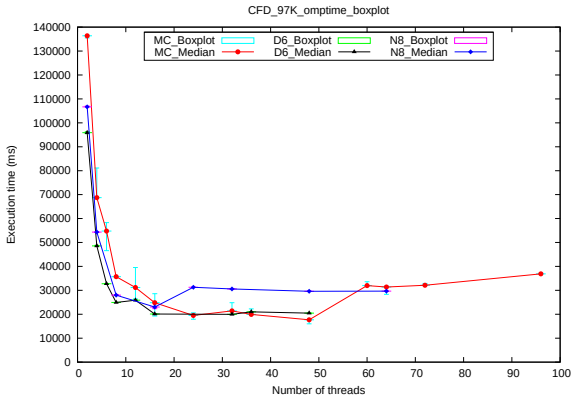
In addition, we notice that when the numbers of OpenMP threads are smaller, the execution times of the three platforms are (approximately) linearly decreased. This is mainly because all the software threads just find their right positions (in hardware) to execute on, the resource contention between each threads are rare.

5.5 LUD

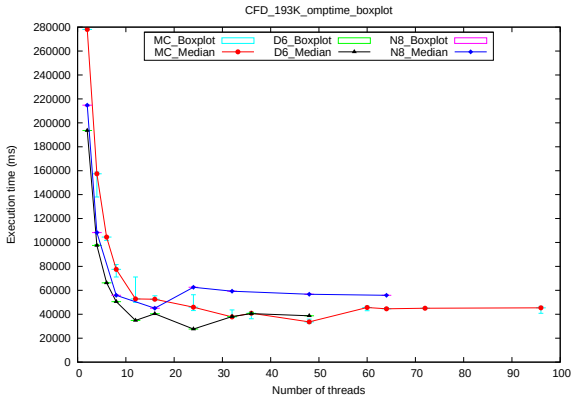
LUD (LU Decomposition) is an algorithm[14] to decompose a matrix as the product of a lower triangular matrix and an upper triangular matrix. The decomposition is done in parallel. For LUD, we use three matrices of 512×512, $1K \times 1K$, and $2K \times 2K$ elements.

In Figure 5(a), as the scale of dataset is small, the execution times on three multi-core platforms are below 150ms when the numbers of threads used are smaller than their numbers of hardware threads. After that they all have a sharp jump, especially on the MC platforms. In Figure 5(b), the execution time curves on three platforms all decrease to their lowest points (at the number equal to the numbers of hardware threads), and

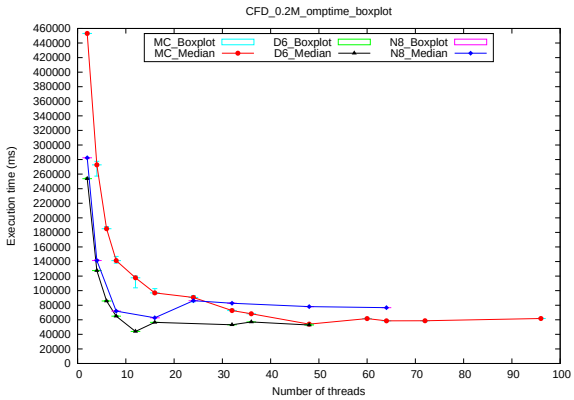
[2]Due to its large execution time, CFD is run three times for each number of OpenMP threads, and the three results become the minimum, median, and maximum for analysis.



(a) 97K



(b) 193K



(c) 0.2M

Figure 4: The execution time of CFD on the three platforms using three datasets of sizes: (a) 97K, (b) 193K, (c) 0.2M.

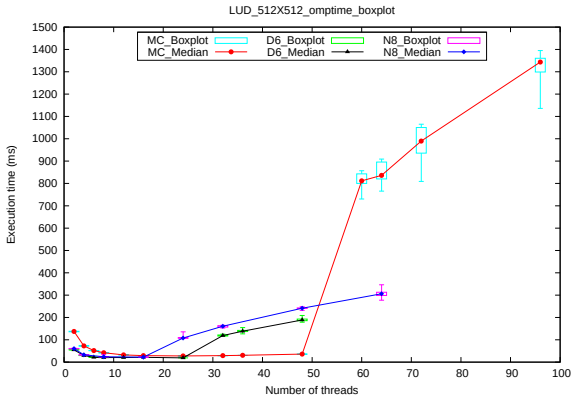
then have a significant leap as the case of 512×512 dataset[3]. For the largest dataset (Figure 5(c)), the total performance trends are more or less consistent, but the execution time of MC are much higher than the N8 and D6 ones. After reaching their minimum values (at 32, 16, 16 threads on MC, D6, N8, respectively), MC fluctuates, while N8 and D6 have a slight jump and continue increasing as the number of OpenMP threads increases.

Besides, the timing behavior is not very stable on MC for the 512×512 dataset and the $1K \times 1K$ dataset when the number of threads used are larger.

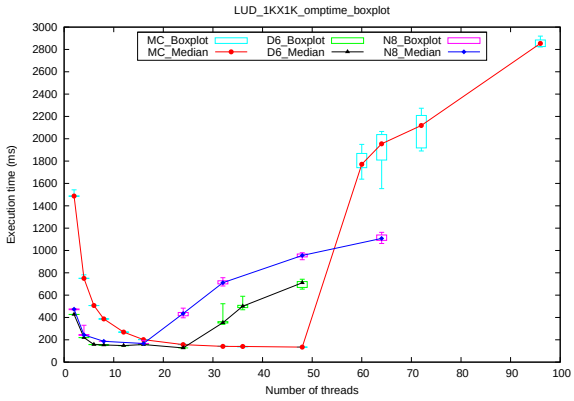
5.6 NW

NW (Needleman-Wunsch) is a dynamic programming algorithm[15] for sequence alignments, which builds up the best alignment by using optimal alignments of smaller subsequences. It consists of three steps: initialization of the score matrix, calculation of scores, and deducing the alignment from the score matrix. The second step is parallelized. For NW, we run three 2D matrix datasets of $1K \times 1K$, $2K \times 2K$, and $4K \times 4K$.

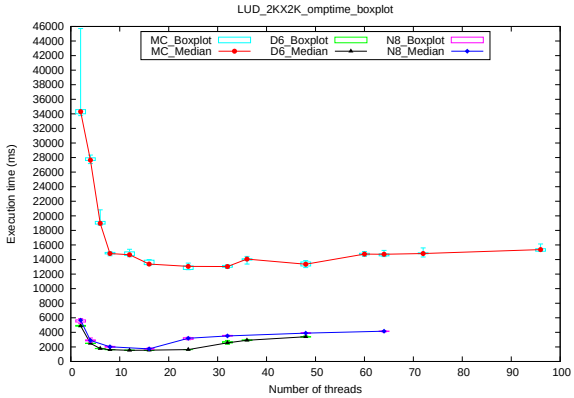
[3]As we have discussed the reasons of the jump in execution time curve before, we skip the same discussion here.



(a) 512×512



(b) $1K \times 1K$

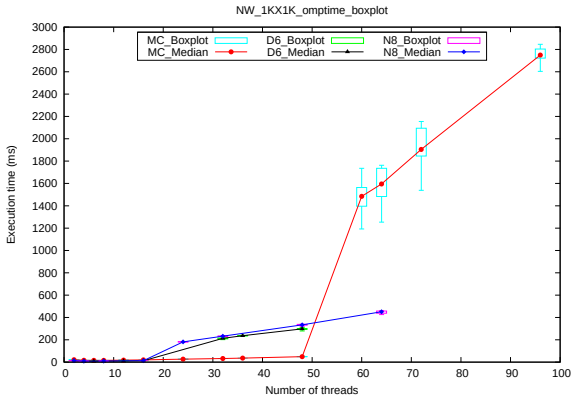


(c) $2K \times 2K$

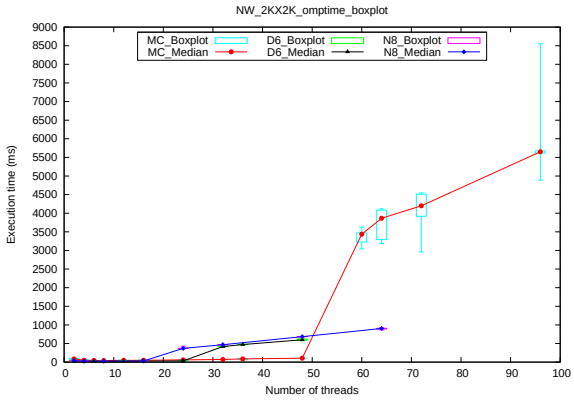
Figure 5: The execution time of LUD on the three platforms using three datasets of sizes: (a) 512×512, (b) $1K \times 1K$, (c) $2K \times 2K$.

From Figure 6, we notice that the execution times of the three multi-core platforms all undergo a first-decreasing-then-increasing trend when the numbers of OpenMP threads we use are less than the numbers of their hardware threads, but those execution times are relatively close and smaller than a certain value. While when the number of threads exceed the number of hardware threads, the sharp increases in these curves are obvious, especially on the MC platform. Thus, the performance of MC is comparable to the performance of N8 and D6 only when we use small amount of threads.

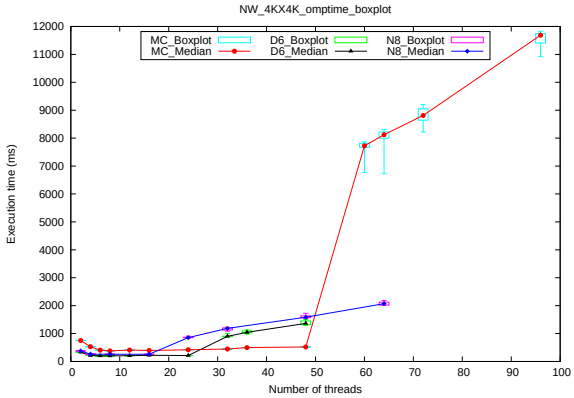
Because NW has diagonal stride memory access pattern, it is hard to exploit data locality to improve performance. Therefore, the performance gained by parallelization is diminished by the overhead of frequent memory accesses and memory latency. Moreover, with the increase of the number of OpenMP threads, there are more threads idle within the initial (`#OpenMP threads - 1`) iterations and the final (`#OpenMP threads - 1`) iterations, which also leads to poorer performance.



(a) $1K \times 1K$



(b) $2K \times 2K$



(c) $4K \times 4K$

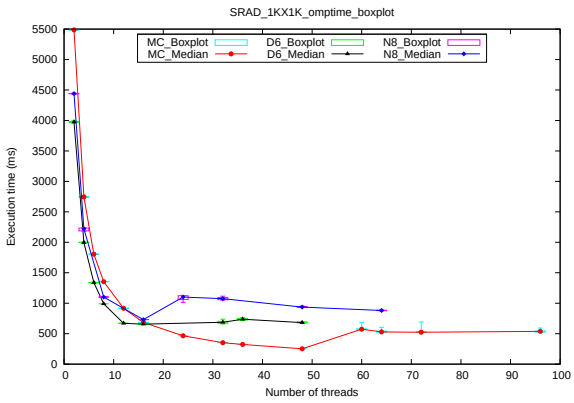
Figure 6: The execution time of NW on the three platforms using three datasets of sizes: (a) $1K \times 1K$, (b) $2K \times 2K$, (c) $4K \times 4K$.

5.7 SRAD

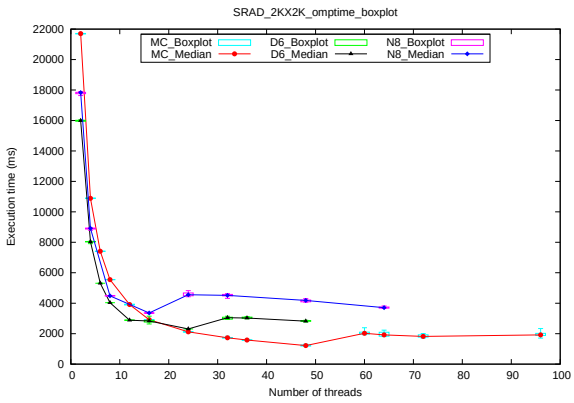
SRAD (Speckle Reducing Anisotropic Diffusion) is a diffusion method [16] used in ultrasonic and radar imaging applications. SRAD is iterative; in each iteration, computing and updating of the whole image are performed in parallel. For SRAD, we have three 2D matrix datasets: $1K \times 1K$, $2K \times 2K$, $4K \times 4K$ pixels.

Figure 7(a) and 7(b) show the cases of smaller datasets. The three platforms all have a quick performance increase till they use all their hardware threads. After that, we observe the execution time leap on each platform, and then N8 and D6 have a decreasing trend in execution time, while MC execution time stabilizes. The exception lies on the D6 platform, where the leap disappears in the case of the smallest dataset. An interesting thing we find is that the curve both have "peak" and "trough" at 16 threads on the N8 platform, which is equal to the number of its hardware threads. For the largest dataset (Figure 7(c)), the execution time of the MC platform keeps decreasing with the number of OpenMP threads increasing, while both N8 and D6 have a small increase at 16 threads. Besides, MC becomes quite unstable, showing large performance gaps between consecutive runs.

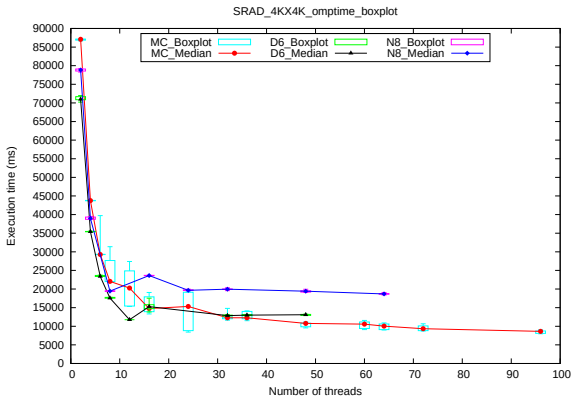
In SRAD application, each thread takes charge of a stripe of the input matrix, resulting in high intra-thread



(a) $1K \times 1K$



(b) $2K \times 2K$



(c) $4K \times 4K$

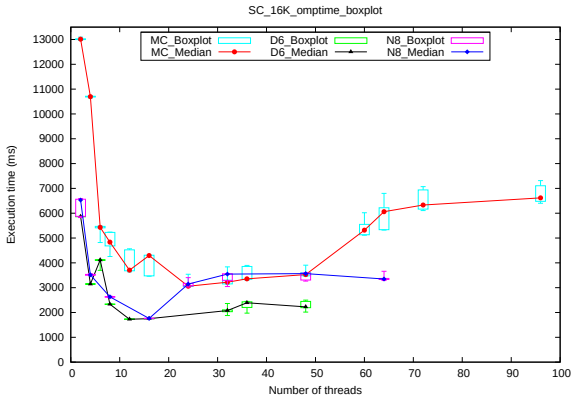
Figure 7: The execution time of SRAD on the three platforms using three datasets of sizes: (a) $1K \times 1K$, (b) $2K \times 2K$, (c) $4K \times 4K$.

data locality. Thus, even the value of each element is depended by its four neighbors, the thread can get these elements with fewer memory accesses. Therefore, the three platforms perform well when executing SRAD application in parallel, without much performance lose.

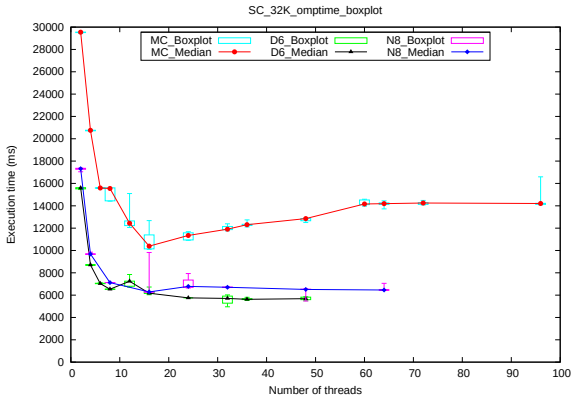
5.8 Streamcluster

Streamcluster (SC) is a data mining algorithm [17] for solving the online clustering problem: given a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. In Streamcluster, we run experiments with 16K, 32K, and 64K points.

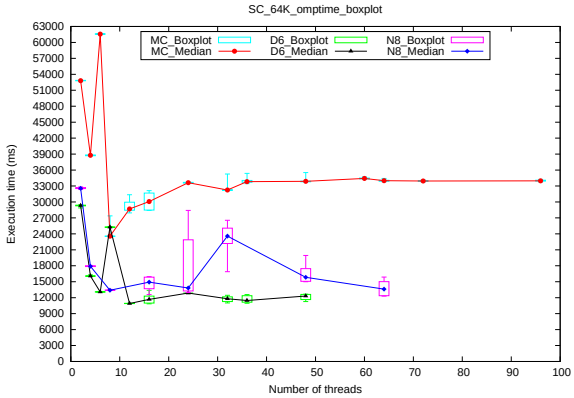
When we run the dataset of 16K-points (Figure 8(a)), we observe that the minimum execution time on N8, D6, and MC is at 16, 12, 24 threads, which is half of, half of and equal to its number of hardware threads. Apart from that, A sharp jump occurs from 48 to 60 threads on the MC platform, and from 16 to 24 threads on the N8 platform, respectively, while the D6 platform has a sudden spike at 6 threads. Figure 8(b) shows the case of median dataset. On MC, after decreasing to the lowest point of 16 threads, the execution time begins to go up with the increase of the number of OpenMP threads. While on N8 and D6, the execution times both undergo



(a) 16K



(b) 32K



(c) 64K

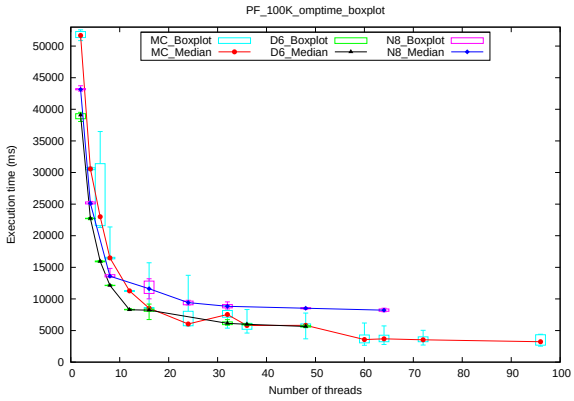
Figure 8: The execution time of Streamcluster on the three platforms using three datasets of sizes: (a) 16K-points, (b) 32K-points, (c) 64K-points.

a quick decrease, then a mild increasing trend (a slight peak at 16 and 24 threads, respectively). In Figure 8(c) with 64K dataset, all the three platforms have a abnormal spike but at different numbers of threads. On the MC platform, the spike happens at 6 threads, followed with its lowest execution time at 8 threads. After that, the curve keeps increasing till 24 threads and turns to level off. On the D6 platform, the execution time curve has a spike at 8 threads, followed with its lowest point as well at 12 threads, and then starts to fluctuates. With regard to the N8 platform, the spike lies at 32 threads, and there is also a small peak at 16 threads.

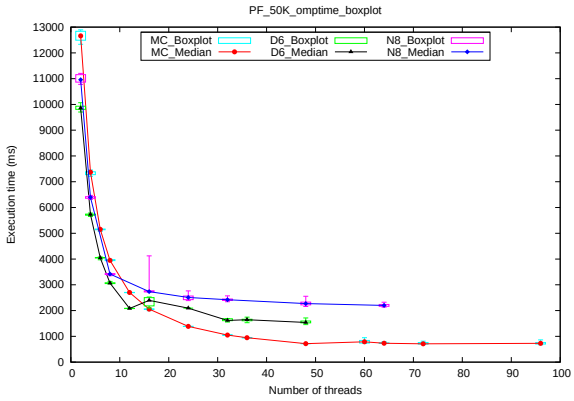
From the results above, we see that there is no general performance trend for the same dataset or on the same platform. Each curve has its distinct features. The causes of the abnormal spikes are still under investigation.

5.9 Particle Filter

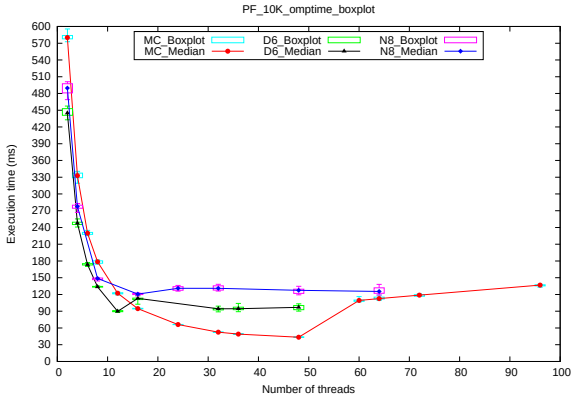
Particle Filter (PF) is a probabilistic model for tracking objects in a noisy environment using a given set of particle samples [18]. The application has several parallel stages, and implicit synchronization between stages is required. For Particle Filter, the three input datasets have 10^4 (10K), 5×10^4 (50K) and 10^5 (100K) particle samples.



(a) 10K



(b) 50K



(c) 100K

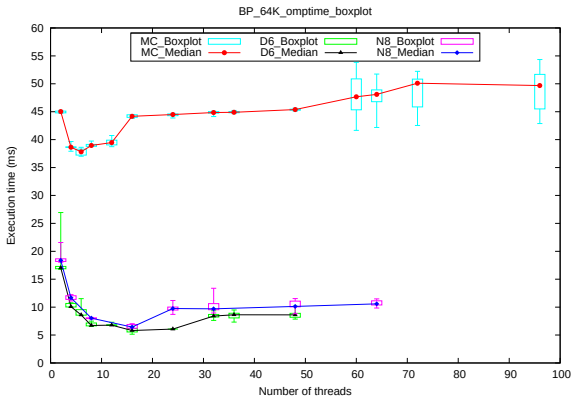
Figure 9: The execution time of Particle Filter on the three platforms using three datasets of sizes: (a) 10K-particles, (b) 50K-particles, (c) 100K-particles.

As the result of using small scale dataset, we see that all three execution time curves have a slight jump in Figure 9(a). They are from 16 to 24 threads on N8, from 12 to 16 threads on D6, and from 48 to 60 threads on MC. The lowest execution time on each platform just lies before the jump, at 16, 12, and 48 threads, which is equal to, half of, and equal to its number of hardware threads, respectively. After that, the curve increases on MC, and decreases on N8 and D6. Figure 9(b) illustrates the the results of using 50K particles. The execution times of the N8, D6 and MC platform all fall from 2 threads to the maximum OpenMP threads (64, 48, 96 threads), showing a general decreasing trend. The only exception occurs on the D6 platform at 16 threads, where a slight peak exists. In Figure 9(c), the execution times of the three platforms are similar to the case of 50K particles. The slight peak changes to MC platform at 32 threads, while the minimum execution time still happens at the point of (approximate) maximum OpenMP threads on each platform.

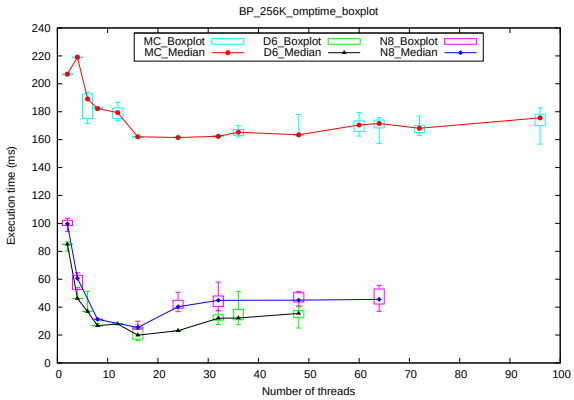
On the whole, the Particle Filter performance of all three multi-core platforms are similar and scale well, although there are several synchronization overheads. As to the slight jumps on the curves in these figures, we believe that they correlates with the issue of integer multiple of hardware threads/cores.

5.10 Back Propagation

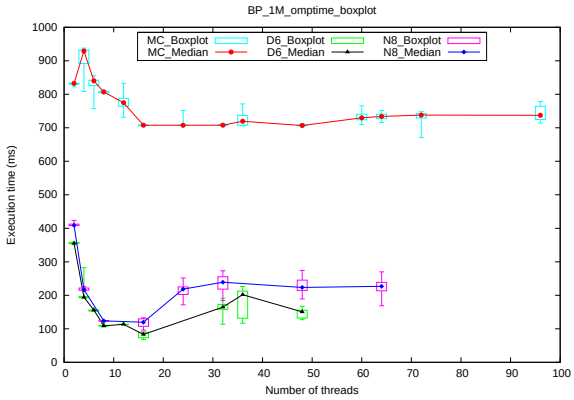
Back Propagation (BP) [19], a neural network learning algorithm, is one of the most effective approaches to machine learning when processing image data. It trains the weights of connecting nodes on a layered neural network; the processing of all the nodes can be done in parallel in each training step. For Back Propagation, we use three datasets of 2^{16} (64K), 2^{18} (256K), and 2^{20} (1M) nodes.



(a) 64K



(b) 256K



(c) 1M

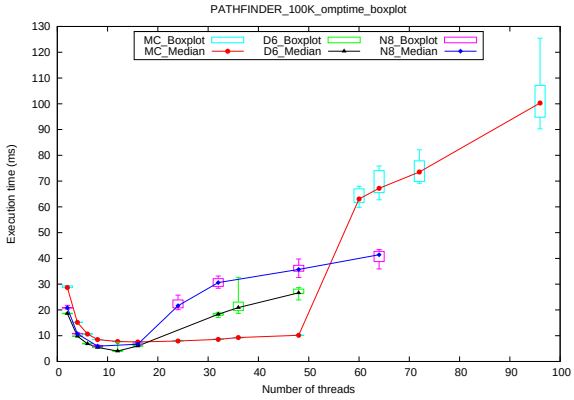
Figure 10: The execution time of Back Propagation on the three platforms using three datasets: (a) 64K-nodes, (b) 256K-nodes, (c) 1M-nodes.

According to the results in Figure 10, we find that the performance behaviors are quite different among different multi-core platforms, but relatively unified within each multi-core platform. On the N8 platform, the minimum execution time lies at 16 threads, which is equal to the number of hardware threads. From the lowest point onwards, there is first a sharp increase, and then a stable trend. On the D6 platform, after decreasing from 2 to 8 threads, there is a slight increase at 12 threads, which is equal to the number of cores. Then the decrease starts again, making the point of 16/24 threads the lowest point. After that, the execution time showing a general increasing trend. The exception happens when using 1M dataset: there is a sudden decrease from 36 to 48 threads. As to the MC platform, we observe quite special execution time behavior. Without a gradually decrease, the maximum values happen at 4 threads when using both the 1M and 256K datasets, and then these

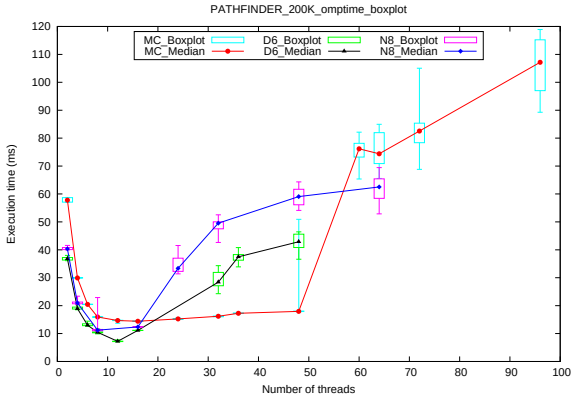
two curves decrease to the point of 16 threads. On the contrary, the curve of 64K dataset has a trough from 2 to 16 threads. From 16 threads onwards, all the three curves of different dataset sizes start to stabilize. In addition, the execution times on the MC platform are much higher than those on the N8 and D6 platform.

5.11 PathFinder

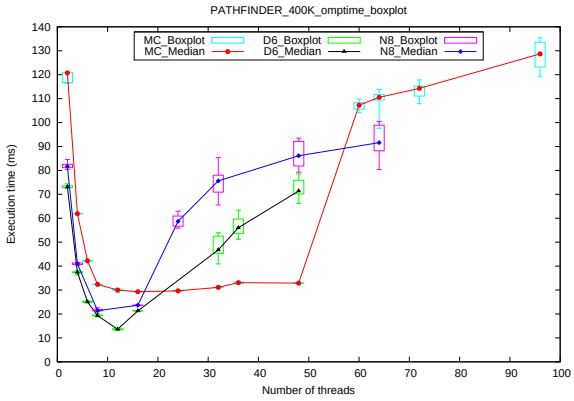
PathFinder [20] is a dynamic programming algorithm to find the shortest path of a 2D grid, row by row, by choosing the smallest accumulated weights. In each iteration, the shortest path calculation is parallelized. In PathFinder experiments, we use three grids with different widths (the number of columns) of 100K, 200K, and 400K elements.



(a) 100K



(b) 200K



(c) 400K

Figure 11: The execution time of PathFinder on the three platforms using three grids with different widths: (a) 100K, (b) 200K, (c) 400K.

From the results (Figure 11), we find that the timing behavior on each platform shows similarities. The N8 platform has tick-shape curves. The execution times drop to the minimum at 8 threads, and after a slight increase at 16 threads, they all begin to increase drastically. On the D6 platform, the curve shapes are coincident with those on the N8 platform, where the half of the number of the hardware threads (12) is the lowest point. Besides, we notice that the execution times increase fast after 16 threads. In the curves of the MC platform,

the execution times all drop quickly to the point of 8 threads, and flatten out till 48 threads, during which the minimum execution times are at the point of 32 threads. After a dramatical increase from 48 to 60 threads, the execution times keep rising.

According to these results, we see that the optimal numbers of OpenMP threads for PathFinder are within a small range around the number of hardware threads/cores. Smaller or larger than this range, the execution times are relatively high.

6 Conclusion and Future Work

So far we have seen diversified OpenMP performance behaviors: For some applications, like BFS and HotSpot, the three platforms perform differently for different datasets, but quite similar for the same dataset; For others, like LUD, NW, and PathFinder, the platforms show similar, steady performance for less OpenMP threads than hardware threads, and a sharp performance decrease afterwards. There are also some applications, like Streamcluster and Back Propagation, in which the performance behavior is irregular, but relatively unified within each platform. For most applications, OpenMP scales well when the OpenMP threads deployed are not larger than the platform hardware threads, and the best performance always happens near the number of hardware cores/threads.

Considering the programming model in a broader view, we see that OpenMP has the advantage of productivity. Parallelization in OpenMP is done in an incremental fashion by inserting the pragma directives at certain positions. Empirically, the programming effort required by OpenMP is quite low, especially when we start from a sequential code. In respect of portability, OpenMP are portable among different multi-core CPUs. In our experiments, we use the Intel or AMD CPUs with different architectures, and the OpenMP implementations can run on each hardware platform without modification.

OpenCL, on the other hand, as a newcomer from the GPGPU world, has the strength to exploit different classes of hardware platforms, such as GPUs, CPUs, Cell/B.E. etc. Its cross-platform portability is a developing trend in parallel computing. In the future, we will focus on the comparison of the two programming models, OpenMP and OpenCL, on the multi-core CPUs to help programmers to make appropriate choices.

Acknowledgments

We would like to thank Kees Verstoep for the DAS4 environment support. The first author is funded by China Scholarship Council - Delft University of Technology Joint Program.

References

[1] Ana Lucia Varbanescu, Pieter Hijma, Rob van Nieuwpoort, and Henri Bal. Towards an effective unified programming model for many-cores. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 681–692. IEEE Computer Society, 2011. 4

[2] Intel Inc. Concurrent Collections for C/C++, 2010. http://software.intel.com/en-us/articles/intel-concurrent 4

[3] Intel Inc. Intel Array Building Blocks (Intel ArBB). http://software.intel.com/en-us/articles/intel-array-buil 4

[4] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(2):173–193, 2011. 4

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization, IISWC 2009.*, pages 44–54. IEEE, 2009. 4

[6] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418, 2009. 4

[7] S. Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2010. 4

[8] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. 5

[9] ASCI. The Distributed ASCI Supercomputer 4, 2010. http://www.cs.vu.nl/das4/. 5

[10] Wikipedia, 2011. http://en.wikipedia.org/wiki/Breadth-first_search. 6

[11] Wei Huang, Shougata Ghosh, Siva Velusamy, K. Sankaranarayanan, K. Skadron, and M.R. Stan. HotSpot: A compact thermal modeling methodology for early-stage VLSI design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006. 7

[12] J. Pisharath, Y. Liu, W. Liao, A. Choudhary, G. Memik, and J. Parhi. NU-MineBench 2.0. Technical report, Technical Report CUCIS-2005-08-01, Center for Ultra-Scale Computing and Information Security, Northwestern University, August 2005. 9

[13] Andrew Corrigan, F.F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, 66(2):221–229, 2011. 10

[14] Wikipedia, 2011. http://en.wikipedia.org/wiki/LU_decomposition. 10

[15] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of molecular biology*, 48(3):443–453, March 1970. 11

[16] L.G. Szafaryn, Kevin Skadron, and J.J. Saucerman. Experiences Accelerating MATLAB Systems Biology Applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits, in conjunction with the 36th IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 1–4, June 2009. 13

[17] Christian Bienia, Sanjeev Kumar, J.P. Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008. 14

[18] M.A. Goodrum, M.J. Trotter, Alla Aksel, S.T. Acton, and K. Skadron. Parallelization of Particle Filter Algorithms. In *Proceedings of 3rd Workshop on Emerging Applications and Many-core Architecture (EAMA), in conjunction with the IEEE/ACM International Symposium on Computer Architecture (ISCA)*, June 2010. 15

[19] CMU. Neural Networks for Face Recognition. http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/faces.h 17

[20] Wikipedia, 2011. http://en.wikipedia.org/wiki/Shortest_path_problem. 18