Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Development of a workload set for multi-core architectures

Erick Martijn van Rijk

CE-MS-2009-05

Abstract

With the rise of multi-core chips in commodity hardware, the need for specialized workloads to evaluate the performance of multi-core systems has become apparent. The current generation of workloads used for evaluating multi-core systems often consist of sequential programs not capable of running on multiple processors and are therefore of limited use for evaluating multi-core hardware. Such sequential programs fail to show the benefits of adding additional cores to a system, since the programs are not capable of using all of the available resources concurrently.

Apple Inc. requested the development of a workload suite that would clearly show the benefits of increasing the number of cores, while stressing the main parts of the system. Key requirements for the workload are: Scalability, Reproducibility and Verifiability. In this thesis report, we present the whole process of workload development for multi-core systems, starting from selecting the programs to be included in the workload, till the application of the workload to actual hardware. In addition, this report discusses the classification of different programs based on fundamental algorithm classes called Dwarfs, which is the basis for selecting possible workload components. The thesis also presents the relevant technologies used for the parallelization of selected workload components. Finally, a case study is discussed showing how to use the developed workload in practice.

Faculty of Electrical Engineering, Mathematics and Computer Science

Development of a workload set for multi-core architectures

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Erick Martijn van Rijk
born in Utrecht, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Development of a workload set for multi-core architectures

---

by Erick Martijn van Rijk

Abstract

With the rise of multi-core chips in commodity hardware, the need for specialized workloads to evaluate the performance of multi-core systems has become apparent. The current generation of workloads used for evaluating multi-core systems often consist of sequential programs not capable of running on multiple processors and are therefore of limited use for evaluating multi-core hardware. Such sequential programs fail to show the benefits of adding additional cores to a system, since the programs are not capable of using all of the available resources concurrently.

Apple Inc. requested the development of a workload suite that would clearly show the benefits of increasing the number of cores, while stressing the main parts of the system. Key requirements for the workload are: Scalability, Reproducibility and Verifiability. In this thesis report, we present the whole process of workload development for multi-core systems, starting from selecting the programs to be included in the workload, till the application of the workload to actual hardware. In addition, this report discusses the classification of different programs based on fundamental algorithm classes called Dwarfs, which is the basis for selecting possible workload components. The thesis also presents the relevant technologies used for the parallelization of selected workload components. Finally, a case study is discussed showing how to use the developed workload in practice.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2009-05

Committee Members :

Advisor:	Zaid Al-Ars, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Georgi Gaydadjiev, CE, TU Delft
Member:	Henk Sips, PDS, TU Delft

To Connor,
your smile can brighten the darkest day

Contents

List of Figures

List of Tables

Acknowledgements

During the 18 months I was working on this MSc. thesis at Apple Inc. in Cupertino, I have met an incredible amount of bright engineers and friendly people and in this section I would like to thank them for their efforts and support.

I would like to thank Myke Smith from Apple Inc. for giving me the opportunity to do my Master thesis in his group. He went above and beyond what I would have expected from a manager to provide assistance and advice when I needed it most, providing encouragement and good company. I would have been lost without him.

I would like to thank the people in the Architecture and Performance Group at Apple Inc. for supporting me during the work on the thesis as well as proving a fun environment to work in. I would have never imagined hard work and fun could have been combined until I worked in this group.

It is difficult to put into writing the appreciation I have for my advisor Dr. Ir. Zaid Al-Ars, with his solid advise during and especially his help at the end of my thesis. Without him it would have been a greater challenge to put my thoughts into writing. His enthusiasm, good teaching and willingness to discuss for long periods of time makes him a great teacher and wonderful addition to the faculty.

I am grateful to my friends and student colleagues for their camaraderie, allowing me to learn and grow and become a better person.

I would like to thank Pamela Smith for keeping me sane during insane times, always willing to have a cheerful chat and allowing me to be part of her family during the time I was away from my own.

A special thanks to my extended family for providing me with a loving and caring environment. My parents who raised me, my sister who played with me and my in-laws who are always up for a laugh.

And finally, but most importantly, I would like to thank my wife Petra and my son Connor for their love and understanding during the periods I have spend separated from them. To them I dedicate this thesis.

Erick Martijn van Rijk
Delft, The Netherlands
July 27, 2009

1

Workload set development

With the arrival of more complex architectural designs for computer hardware, comparing them by just looking at specifications such as clock frequency and memory bandwidth is no longer feasible. Instead one must execute a variety of programs on the different hardware architectures, and then compare actual execution times for these programs on each architecture, a process called *workload testing*. The major challenge in workload development is choosing the selection of programs that must be:

Realistic Be representative of typical software run on the systems

Exhaustive Exercise all possible hardware bottlenecks

Efficient Execute in a reasonable amount of time

Verifiable Verify the computed results to ensure their correctness

Balancing these factors can be tricky. To date several groups have made audited and verifiable workloads for industry use. One of the most well known groups is the Standard Performance Evaluation Corporation (SPEC) founded in 1988 [31]. SPEC tries to use real world applications and intentionally emphasizes the second challenge over the first. The primary focus of workloads like SPEC CPU2006 [32] is exploring the corner cases of hardware architecture.

1.1 Problem description

The computational resources available in multi-core hardware are not fully utilized by current software applications. Special software is needed to fully stress current and future scalable multi-core architectures. This thesis looks into the process of finding and selecting suitable applications for inclusion into a multi-core workload.

Creating a suitable workload that is capable of fully stressing a multi-core system requires taking a look at the different kinds of software applications available. To fully utilize the available computational resources provided by multi-core architectures, use of all the available cores is necessary during computation. This means that the applications must be divided into sections that can be run in parallel. The algorithms used in these applications can be divided into three categories:

- Embarrassingly Parallel: The most suitable algorithms are the embarrassingly parallel algorithms, ones that can be easily divided into parallel regions without any adverse effects on the outcome of the algorithm [16]. Embarrassingly parallel algorithms are primarily found in scientific applications that work with large datasets and are specifically designed to be run on multi-core machines.

- Partially Parallel: The partially parallel algorithms can be suitable if the parallel parts of the algorithm are the most computationally intensive part and the serial parts have minimal computational requirements.
- Sequential: A sequential algorithm is often not suitable for parallelization, because each stage of computation may depend on the previous set of values [16].

To develop our scalable workload, we decided to look for real world applications in the domain of scientific computing, consisting of High Performance Computing (HPC), BioInformatics and an area what Intel calls "Recognition, Mining and Synthesis (RMS)" [12], as shown by Figure 1.1. We also decided to look at the media domain since modern media requires significant computation for content generation. During our initial study we decided that the Bioinformatics and Intel RMS applications are so complex that any proper evaluation falls outside the scope of this thesis. We will discuss the Scientific benchmarks in detail in chapter 4 and the Media benchmarks in chapter 5

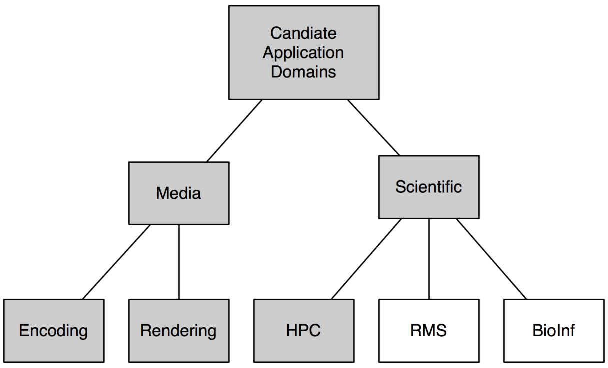


Figure 1.1: Candidate application domains for the workload

1.2 The project and the objectives

The project goal is to create a workload set for use by Apple Inc. It will be a verifiable workload that can be checked for correctness and will run in both 32 and 64-bit mode. The workload set should simulate high-end user behavior but be executable within a limited timeframe, such as 12-24 hours. This is much shorter than the a full run of the SPEC CPU2006 benchmark, which takes up to 3 days to complete. Because of our focus on multi-core architectures, specific attention should be given to how threading can be used to enhance performance of the applications. For each component in the workload, the following procedure will be used. First, we will look at several types of HPC, encoding and rendering benchmarks and select suitable programs from each to be analyzed. The analysis phase will then test each selected program and select the most interesting programs based on the presence of bottlenecks or high computational intensity, if possible. If the program is not suitable for workload testing, based on specific requirements (specified in section 2.3), the process of selecting begins again. When a component passes these requirements, it will be added to the workload set. Finally, a report generation and verification elements will be added to the component to ensure that it adheres to the workload set requirements.

1.3 Component selection procedure

For selecting components the following steps are taken to make sure the components were suitable for inclusion in the workload:

1. *Parallelization:* There are multiple ways to parallelize applications that are capable of using multiple cores on the host machine. So we need to select the most appropriate way to parallelize the application. There are several support libraries for multi-core systems available (Pthreads, OpenMP and MPI, among others). The preference goes to solutions that have shared memory support, so that threads all have access to common memory and there is no need for a communication layer (i.e. message passing, used with MPI). More information will be given in section 3.2.

2. *Building:* When a possible component has been selected, we first need to build it for the OSX platform to generate a working binary. This is heavily dependent on the application and source code type; some applications have been tested on OSX before or are already native to the OS.

3. *Scalability analysis:* Once the Application is multi-threaded, then we can start to look at the scalability of the application. We need to make sure that the application can always use all of the available cores. Since the workload needs to be usable for at least 2 years after completion, we need to make assumptions about future hardware components and speedup based on available roadmap information and current trends. Since our main target is consumer level machines ranging from simple laptop/desktop computers to workstation/server systems, and not enterprise-level mainframes, we can limit ourselves to machines that may contain 32 cores and can execute 64 threads concurrently. We expect that the increase in CPUs will not give a linear increase in performance. Various factors, like sharing of the memory subsystem and blocking while we are waiting for data from other threads, will usually limit us to sub-linear speedups. However, the application should not hit a ceiling where cores do not provide any additional speedup.

4. *Behavior profiling:* When we have a scalable parallelized or also called a multi-threaded application, we can start the in-depth profiling of the application and the data each application uses (also called the dataset), to look for instruction mix, branch and cache behavior, and bandwidth utilization of the application and our data set. The items we need to look at are: instruction mix, branch behavior, cache behavior, bandwidth utilization and memory latency. All of these items affect the overall performance of an application on an architecture. The tools used to look into these behavioral patterns are mentioned in section 3.3

1.4 Document overview

The layout of this thesis is as follows. Chapter 2 introduces the concept of Dwarfs which are common scientific algorithms. We will target these specific algorithmic methods specifically when selecting components for the workload. The relevant workload areas and the criteria for the workload are also discussed in this chapter. Following this , Chapter 3 will discuss the workload environment and parallelization methodologies. The application profiling tools that were used are also discussed in this chapter. Chapter 4 will discuss the various scientific applications we selected for the workload and an in depth analysis on each application. After this, Chapter 5 will discuss the two media applications that we selected for the workload and an in depth analysis on each application. In Chapter 6 a case study of the finished workload will be presented. Finally, Chapter 7 summarizes all the previous chapters and will present our conclusions and recommendations for further research.

Workload characteristics

2

So what is a workload? A workload is a group of programs that test various hardware parts of a computer architecture. The results of these programs (primarily runtime, but also sometimes throughput, bandwidth, etc.) can be analyzed to give the architecture a rating compared to other architectures that have run the same set of programs. One of the most important criteria of a good workload is that the results are verifiable and are not biased to a particular architecture.

2.1 Workload classification: the 13 Dwarfs

When looking for suitable applications to be included in a workload, we need to look at the kind of numerical and algorithmic methods used to implement the application. A good way to characterize an application is to subdivide the algorithms used in these applications into fundamental algorithm classes called *Dwarfs*. An example of an ideal workload would look like Figure 2.1, where all the types of Dwarfs are covered by the applications contained in the workload. The concept of Dwarfs as common building blocks of larger applications was originally proposed by P. Colella [10], who categorized common scientific algorithm implementations into seven Dwarfs. A working group in Berkeley [6] expanded this concept by adding an additional 6 Dwarfs to the original 7, making the total of the following 13 Dwarf categories:

	Dense Linear Algebra	Sparse Linear Algebra	Spectral Methods	N-Body Methods	Structured Grids	Unstructured Grids	Map Reduce	Combinational Logic	Graph Traversal	Dynamic Programming	Back-track & Branch-and-Bound	Graphical Models	Finite State Machine
Dwarfs:	1	2	3	4	5	6	7	8	9	10	11	12	13
Included in workload	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Figure 2.1: Ideal content of the workload covering all possible Dwarfs

1. *Dense Linear Algebra:* Matrix and vector operations on contiguous arrays of data elements.

2. *Sparse Linear Algebra:* Similar to Dense Linear Algebra, but with a large number of zero entries in the array. Compressed data structures are commonly used.

3. *Spectral Methods:* Operations performed in the spectral domain often get their input data from the temporal or spatial domains. Spectral algorithms usually use multiple stages to process the data.
4. *N-Body Methods:* Algorithms based on N-Body methods are calculations that are dependent on communication between many discrete points. Common algorithms are Barnes-Hutt and Fast Multipole.
5. *Structured Grids:* Data is stored in a multi-dimensional structured grid where grid-points are updated based on their neighboring values for each computational iteration.
6. *Unstructured Grids:* Data is stored in an irregular manner, where each data element is defined by the significance of its location within the global structure. Data elements can be a point, edge, facet or volume. The grid-points are updated based on their neighbors in the grid or mesh.
7. *MapReduce:* Originally called Monte Carlo, it is a repeated independent operation with little or no communication between the operations. The results are collected and stored or reduced at the end of each operation.
8. *Combinational Logic:* Performing simple operations on large volumes of data and aggregating the result into a final solution. Examples are hashing functions or on-the-fly processing of datastreams. High throughput is critical for this Dwarf.
9. *Graph Traversal:* Traversal of a collection of objects and analyzing the properties of these objects. Search algorithms often use graph traversal.
10. *Dynamic Programming:* Dynamically sub-dividing the larger problem into smaller sub-problems to find the optimal solution based on these smaller sub-problems. Used in DNA sequence matching.
11. *Back-track and Branch-and-Bound:* Uses the divide and conquer principle to subdivide a huge search space. These sub-spaces (branches) will be analyzed and processed independently; afterwards, they are processed by their higher level parents (roots).
12. *Graphical Models:* Sets of nodes with variables that are connected by edges representing the probabilities. Examples are Bayesian networks, hidden Markov models, and neural networks.
13. *Finite State Machine:* Comprised of interconnected sets of states, where a state can transition from one to another state when a specific condition has been met.

Matching applications to certain Dwarf characteristics will give an idea of the behavior of said applications. For example, Linpack, a benchmark that computes a dense linear system, would fit into Dwarf 1 (Dense Linear Algebra) with coarse grain parallelism. The implementation of all the above mentioned dwarfs into the workload is our goal, in this case our workload will provide a good insight on the behavioural patters of a particular architecture.

2.2 Targeted Hardware Parts

Since the hardware parts used by possible workloads include CPU, memory, graphics, I/O, memory interconnects we need to limit our selection of these hardware parts. Therefore we confine our research to workloads that stress the following parts of the hardware, and thereby limiting the scope of our research to the following targeted hardware parts:

- CPU
- Memory
- Memory interconnect (also referred to as North-Bridge or Front Side Bus (FSB))

We designed our workload so that the other hardware parts are not a bottleneck that can hamper the targeted hardware. For example, I/O will be minimized from the core of each benchmark.

Suitable areas where we can find applications that stress the CPU and/or the memory interconnect are found in the scientific community and media tasks. High Performance Computing (HPC) applications used by many areas of science push the computational limits of current hardware. Creating or modifying media requires vast computational power, and many algorithms are multi-threaded.

2.3 Criteria of the workload

The workload should be capable of fully stressing current and near-future high-end systems. Because the current roadmap of key hardware parts like the CPU and the memory interconnect call for increasing performance according to Moore's law [23], we can expect that in the next 18 months the performance of these sub-systems will approximately double. The performance of the CPU will most likely increase because of the increased number of cores on the same die, and not because of an increase in clock frequency [24]. With the numbers of cores increasing, parallelization of software is necessary to take full advantage of the architecture. Hence our first requirement: *The workload should have full support for multi-core systems, meaning it should be able to take advantage of all of the available cores.*

The state of operating systems is in constant flux, with minor revisions every few months and major revisions approximately every 2 years. Hence, a common use for the workload will be to ensure that there is no regression in performance between revisions. This brings us to our second requirement: *Full support of Mac OS X by the workload.*

Running workloads is a time consuming practice, with larger workloads running continuously for days. While the workload needs to run long enough to collect enough data to be analyzed, it still needs to be short enough to be practical to test with. That leads us to our third requirement: *The workload needs to complete and give results within a reasonable time of 12-24 hours, so we can get the results the next day.*

Getting results from the workload is a vital part of the performance analysis process. The timing and performance results produced by a particular workload should not vary when run repeatedly on the same platform (both hardware and software). When the results vary greatly, even with the same setup, the timing results become meaningless and the workload ceases to be useful, since one cannot distinguish between random workload variation and significant test measurement variation. This brings us to our fourth requirement: *Reproducible test results on the same platform.*

Finally, the output of the benchmarks/applications in the workload should be scientifically correct and should not differ between platforms. The results cannot vary when run on different platforms because of rounding errors or calculation errors. This brings us to our fifth and final requirement: *The workload needs to produce verified results that do not change when run on other platforms.*

To summarize these requirements:

Scalability The workload should have full support for multi-core systems, meaning it should take advantage of all of the available cores.

Compatibility Full support for Mac OS X systems

Duration The workload needs to complete and give results within a reasonable time of 12-24 hours, so you can start your test and get results the next day.

Reproducibility Reproducible test results on the same platform.

Verifiability The workload needs to produce verified results that do not change when run on other platforms

In the following sections the issues of scalability and reproducibility are discussed in more detail.

2.3.1 Scalability

An interesting issue is the scalability of the workload for the upcoming years. Any useful workload will need to take into account the rapid increase of computational power and threading capability in future hardware. It is a fair assumption that by 2010 we will have workstation-level machines that will be able to process up to 128 threads concurrently [9, 12, 18, 27, 28, 34]. To address this scaling problem there are 4 typical application scaling methods [16]:

1. Constant Dataset ("Problem Constrained"), where the dataset is fixed and the execution time decreases with more cores.

   pro: The dataset remains the same so calculating speedup is easier.

   con: The problem here is that the decomposition of the dataset used for the problem across parallel processors could heavily influence the performance. If each processor's data partition becomes too small, the communication overhead can become an issue. Processor caches can also have a dramatic effect on performance if they enlarge enough to hold a key working set of the application

2. Scaled Dataset ("Memory Constrained"), where the dataset increases by multiples of the number of cores.

   pro: Multiplying the dataset size is easy for some applications.

   con: Memory does not scale at the same rate as the number of processors in many systems of interest. Runtime can increase significantly. Speedup calculations will become more complex.

3. Constant Time ("Time Constrained"), where the execution time is constant and the dataset is scaled accordingly.

   pro: Computational runtime will be predefined.

   con: Only easy for streaming data (video encoding etc). Otherwise, dataset needs to be tweaked for each architecture, because aborting the computation midway will influence the results. Requires a lot of work.

4. Constant Efficiency ("Isoefficient"), where the computational efficiency is kept constant by increasing the dataset and the execution time to counter inefficiencies caused by added communication and synchronization time required by larger number of cores.

   pro: Efficiency can be decided by developer/user.

   con: Dataset needs to be tweaked for each architecture. Requires a lot of work.

Changing the dataset has too many downsides when considering that the results need to be compared between significantly different computational architectures. Therefore, we chose to use the Constant Dataset method because of its ease of implementation and simple comparison of results. We can define the dataset to be large enough so it will not be influenced by the communication overhead in the designed lifetime of this workload. When the dataset eventually becomes too small, it can be replaced with a new, larger one. This is a methodology used by SPEC to periodically update their benchmark suites with new datasets and applications [33].

Because we use the Constant Dataset method, we can calculate the total runtime T(p) of an application taking into consideration the serial portion of the application. This method is defined using Amdahl's law [4]:

$$T(p) = T_s + \frac{T_p}{p} \tag{2.1}$$

Where:

- *T(p)* is the total runtime required to execute on a p-processor system
- *Ts* is the time required for the serial part of the code to execute
- *Tp* is the time required for the parallel part of the code to execute on a single processor
- *p* is the number of processors

To calculate the speedup and efficiency, we use the Karp-Flatt metric [20] where we derive the serial fraction from the experimentally obtained runtimes by changing the numbers of active cores. Since the applications in the workload are too complex to implement an optimal serial algorithm, we define the serial runtime T_1 as the runtime with one core active with the knowledge that possible overhead for parallelization might be included in this T_1. Based on the reworked equations 2.2 and 2.3 we can estimate the serial fraction s that a particular application has. Based on this we can make an estimate on how an application might scale on machines that have more cores than our test system. This estimate we call the derived Amdahl's scalability.

$$\begin{aligned} T(p) &= T_1 * s + \frac{T_1 * (1-s)}{p} \\ T(p) &= s[T_1 - \frac{T_1}{p}] + \frac{T_1}{p} \end{aligned} \tag{2.2}$$

$$\begin{aligned} s &= \frac{T(p) - \frac{T_1}{p}}{T_1 - \frac{T_1}{p}} \\ &= \frac{\frac{T(p)}{T_1} - \frac{1}{p}}{1 - \frac{1}{p}} \\ &= \frac{p * (\frac{T(p)}{T_1} - \frac{1}{p})}{(p-1)} \end{aligned} \tag{2.3}$$

Another benefit of the Karp-Flatt metric is that the change in the serial fraction s can give us an insight on certain hardware limitations we might come across during scaling from 1 to 8 cores. Some examples are [20]:

- Irregularities in the serial fraction indicate load balancing issues and cache interference.
- Decreasing speedup with a smooth increase of the serial fraction can point to increasing overhead in synchronization.
- Constant serial fraction while the efficiency is dropping points to a limited parallel code portion.
- Decreasing serial fraction and the efficiency is dropping means the overhead is increasing.

2.3.2 Reproducibility

Another issue is the reproducibility of the tests done by the workload. So that small but real performance differences between systems can be measured, there can be only a minimal random performance variance. The industry considers a random difference between the min and max performance results below 5% among multiple iterations performed on the same system to be acceptable. However, this approach is rather ad hoc. A more systematic approach is given in section 2.3.4. The workload needs to include checks for any variations that may occur, to make sure they are within the specified tolerances. Variances in runtime and performance become more significant when we are completely saturating limited system resources such as CPU time and memory bandwidth. Since modern operating systems have multiple background processes for maintenance tasks, network services, etc., when background processes are scheduled instead of the target application they will affect the machine performance and produce variations in the final results. Even the scheduling of the processes on different CPUs can play an important role in performance, because cache sharing and resulting data locality can impact the performance significantly. To demonstrate the influences that the operating system and cache locality can have, the following section discusses variations we observed using Torque, a simple bandwidth-testing application.

2.3.3 Torque: example workload application

The application Torque tries to saturate the memory subsystems by loading or copying blocks of memory as fast as possible. Torque has an option to create multiple processes in order to exercise multiple processors simultaneously. Every process is capable of saturating the memory subsystem by itself, but when we run, for example, 8 processes concurrently on an 8 core machine, the effects of cache locality and scheduling become apparent. For this test, we ran 4 load processes and 4 copy processes in parallel. Torque was set to do 100 internal iterations of bandwidth usage and the average result was logged to an output file. This process was repeated 1000 times and these statistics were analyzed to detect variations caused by cache locality and scheduling.

Performance evaluation To examine the range of variation, a histogram of the performance statistics was made in Figure 2.2. The x-axis shows the throughput of the memory bus (Front Side Bus or FSB) in MB/second. The y-axis shows the frequency of occurrence of a particular throughput score over the course of the 100000 iterations run for the test. The histogram in Figure 2.2 shows that the majority of the performance is indeed centered around the mean of 5754 and the standard deviation goes from 5707MB/second to 5792MB/second, which represents a variation of 2%. The total difference between the lowest and highest bandwidth measured was 13%. The ones that are outside of the standard deviation perform either poorly or exceptionally well, as a result of behavior caused by cache locality and by the scheduler. In the worst case, the different processes are interfering with each other in the cache and on the FSB. They may also have been pre-empted by the scheduler for a background process for a time. In the optimal case, there was little to no interference between the different processes, and the processes stayed on the same core during the entire run. Other real world applications should have variances in performance similar to Torque's.

The second peak in the histogram plot at 5800 MB/Second is caused by grouping of specific processes on the cores. To find out what causes the better performance, further investigation was done and explained in Appendix A.

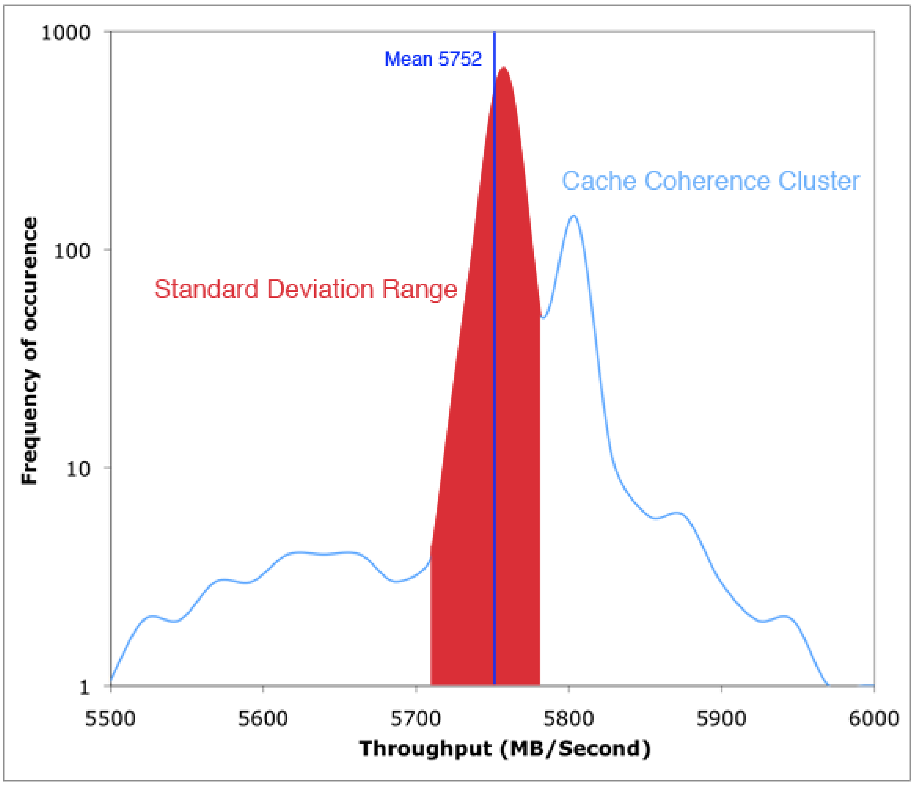


Figure 2.2: Torque histogram of memory access patterns

2.3.4 Reproducibility Analysis

To prevent possible outliers (a data point on a graph or in a set of results that is a lot bigger or smaller than the next nearest data point) affecting the final results, we will try and remove these outliers by analyzing the performance measurement results of several iterations. As discussed in the previous section these outliers can be caused by interruption of the application under test, due to the schedular or hardware behaviour such as cache coherence. To satisfy the reproducibility requirement, we use the concept of coefficient of variation (CV) which is defined as the ratio of the standard deviation σ and the mean μ:

$$CV = \frac{\sigma}{\mu} \tag{2.4}$$

The key benefits of CV is that it is a dimensionless number, so different datasets can be compared without problems. Based on the experimental data we have collected, we concluded that the CV needs to be smaller than 2% to be certain that the final results are valid. This means that the standard deviation is 2% of the mean of all the samples taken.

The steps for the reproducibility analysis, the procedure we use to check if the results are valid, are shown in Figure 2.3 and explained in the following paragraph.

Before we can begin the reproducibility analysis, we first need to run the iterations and generate the performance results. When the required number of iterations is com-

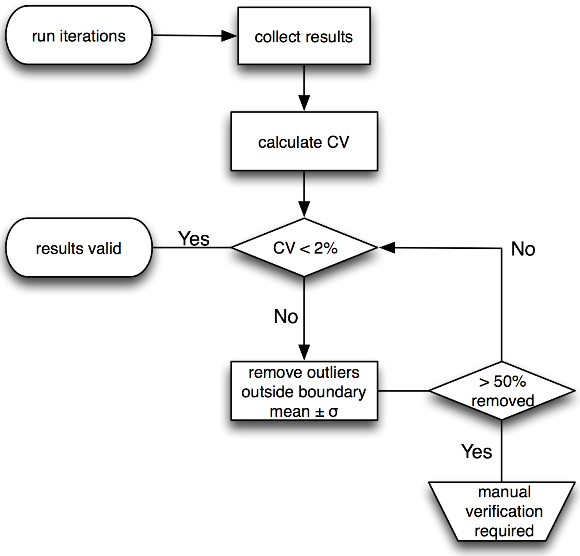


Figure 2.3: Flowchart of the reproducibility analysis

plete, we collect the performance results for processing and calculate the coefficient of variance (CV) for all collected performance number. Now we can start the reproducibility analysis by looking at the values of the CV. When the CV is not within the predefined threshold of 2%, we will remove outliers further than σ from the mean and recalculate the mean and standard deviation from the remaining values, and we will repeat the verification step. This process continues until the CV is below 2% or half of the returned values are removed. If we have removed more than half the results, the whole batch is considered bad and the user will need to manually check the state of the system. For all of the components in the workload the same reproducibility analysis will be applied and the final results will be based thereon.

The reproducibility analysis will be explained using one of our workload components NASA Parallel Benchmark CG (NPB CG), discussed in section 4.2.5, as an example.

CG produces similar results across multiple runs with only a small variance in the total runtime. However, CG has a large variation when comparing the minimum and the maximum for a 100 iteration test. The difference between the shortest and longest time was 16%, a considerable difference. To minimize the influence of these outliers we remove them as described in the flowchart shown in Figure 2.3.

Iteration	1	2	3	4	5	6	7	8	9	10	CV
Pass 1	180s	194s	183s	182s	182s	184s	187s	180s	183s	205s	4.29%
Pass 2	180s	Invalid	183s	182s	182s	184s	187s	180s	183s	Invalid	1.34%

Table 2.1: Datapoint validation of multiple iterations of NPB CG

After processing the initial collected performance results, we will proceed and calculate the CV of all the 10 iterations in our example. In our example the results, shown

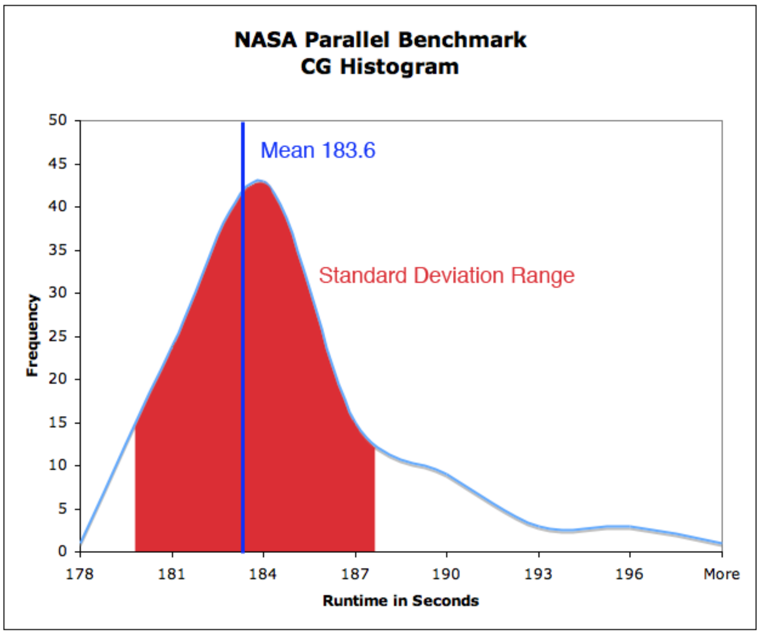


Figure 2.4: Histogram of CG runtime variance

in Table 2.1, the CV is 4.29%, too high to pass our reproducibility analysis. So the next step is to find the outliers that fall outside the boundary defined by *mean* $\mu \pm$ *standard deviation* σ, highlighted in the Figure 2.4 where the x-axis shows the runtime required to complete a single iteration of CG and the y-axis is the number of tests reporting a particular duration. When we have removed the outliers we recalculate the CV of the remaining datapoints and test if we pass the reproducibility requirement. In our example, shown in Table 2.1, the datapoints of iteration 2 and 10 (with a runtime of 194 seconds and 205 seconds respectively) fell outside the boundary and were invalidated. Due to the removal of these two datapoints the reproducibility requirement was passed at the second pass of the CV check and now the results are validated. During this process, every removal of additional datapoints will make the boundary smaller, resulting in a more strict validation process.

3

Workload evaluation environment

In this chapter we discuss the evaluation environment that we used for testing our applications. The profiling tools used to determine the suitably of the applications for the workload are also discussed in this chapter, including the impact profiling has on the actual metrics collected.

3.1 Baseline system setup

Our baseline system is a 3.0 GHz Mac Pro 8-core (with two Quad-Core Intel Xeon processors) machine, stocked with 8GB of main memory. The baseline system ran Mac OS X 10.5. This is a cache coherent Uniform Memory Access (ccUMA) architecture, with the 8GB spread over 4 separate memory access lanes that can be accessed independently, as is illustrated in Figure 3.1. A ccUMA architecture is an architecture where the main memory is shared among the processors with equal access time and any write operation to a processor's cache will invalidate the other processor's caches, keeping the caches coherent in the process.

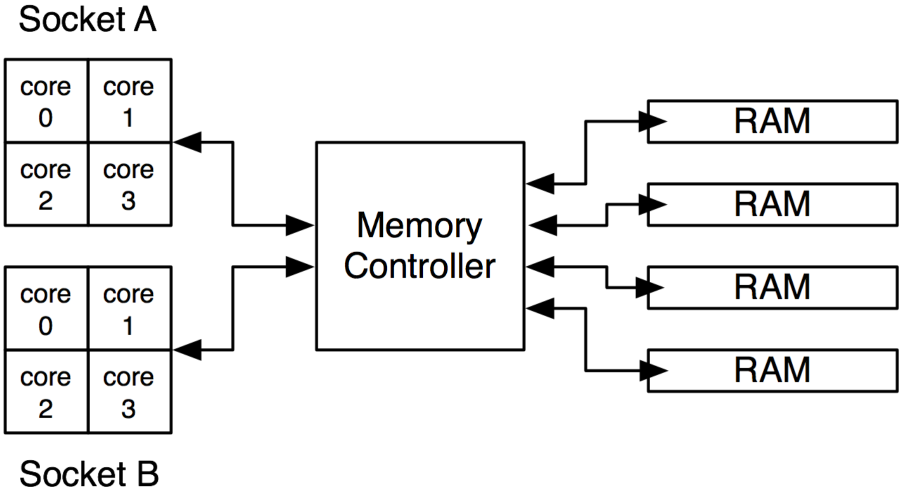


Figure 3.1: Mac Pro 8-core architecture layout

3.2 Parallelization methodologies

Because of the advent of multi-core processors, there needs to be a paradigm shift from the current serial programming methodology to a new parallel programming methodology. The current serial way to process work is to start a job in a single thread and wait for it to complete before doing something else with the data. This was the only way to do one's work in a uniprocessor world, but now, with multi-core chips available to the general public capable of running many threads concurrently, this method is a waste of the available resources. There are three ways to parallelize applications:

- Splitting the *data* in separate parts, generally referred to as the divide and conquer method. This is illustrated in Figure 3.2, where the dataset is chopped up into four similarly-sized parts to be processed independently.
- Splitting *program code* into separate parts and pipelining, as is illustrated in Figure 3.3. In this case, each core performs a dedicated computational stage, and all the data flows through all of the cores.
- Hybrid model where the process is pipelined, but data division is done in some pipeline stages.

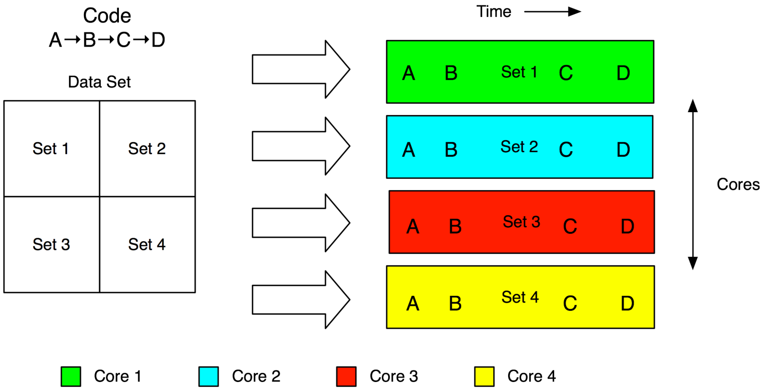


Figure 3.2: Divide and conquer

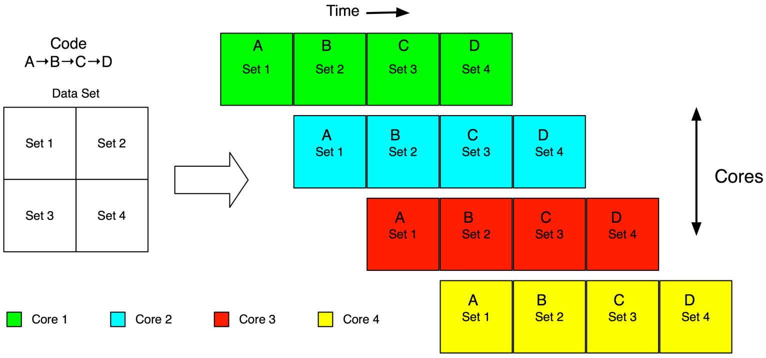


Figure 3.3: Pipelining

There are several libraries that supply a framework to control threading and communication in parallel applications: POSIX Threads (Pthreads), Message Passing Interface (MPI) and OpenMP.

During the process of selecting the applications to be included in the workload, we also need to take a look at the methods used to parallelize the applications. Some parallelization methods are more suitable than others, as we will explain in the following sections.

3.2.1 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a communication protocol library that enables the user to create scalable high performance parallel applications. The MPI library supplies an API for various programming languages including Fortran, C and C++. The way MPI works is that the application is written so that a new copy of the application is started on every core. The MPI system keeps track of each copy and gives each copy a tracking number. These marked applications are then placed on different cores to be run in parallel without working in the same memory space, and explicitly package up data into *messages* when communication among the parallel processes is required. The memory footprint will increase with each process added with the rate of $(datasetsize + application\ footprint) \times nproc$. The complex MPI API has a steep learning curve, and is therefore mostly implemented for clusters, where separate memory spaces and applications are necessary.

3.2.2 POSIX Threads (Pthreads)

POSIX Threads (Pthreads) is the standard programming interface for threads in UNIX systems using the C language (IEEE standard 1003.1 [25]). By using threads, the program can occupy multiple cores at the same time and sub-divide its work. Depending on the data dependency of the chosen algorithm, the threads can be used for the divide-and-conquer method or the pipeline method. Pthreads is considered a medium skill level programming framework, where the developer needs to learn the basic concept of parallel computation, but doesn't need to explicitly manage all the communication. It is possible to mix Pthreads with MPI to create a hybrid applications that can work in a shared memory mode locally on a multi-core machine, and also use MPI for communication with other machines in a cluster.

3.2.3 OpenMP

OpenMP is a framework for shared memory systems where the source code is marked with compiler directives that allow the compiler and a run-time library to parallelize the application. Because only a few keywords need to be added to parallelize loops, OpenMP is easier to learn and to implement when going from serial code to parallel code. Using environment variables in the run-time environment, the user can easily specify how many threads are generated and how they are mapped to their system. Another advantage OpenMP has is the capability to dynamically change the number of parallel tasks based on the underlying hardware, to make optimal use of the system. Like Pthreads, it is also possible to mix OpenMP code with MPI so that a single machine can run multiple threads using OpenMP while executing an application across a cluster of machines using MPI.

When looking at these methodologies, our preference is towards applications using OpenMP, because the implementation has support for dynamic scaling and does not have the memory usage overhead that MPI has. Also, our evaluation environment is a ccUMA architecture, as described in section 3.1, so we can use shared memory parallelization techniques.

3.3 Application profiling tools

During the characterization phase of the workload selection process, we looked at multiple segments of both the hardware behavior and software behavior of each application. By using the following applications during profiling, we got a good overview of the behavior and possible bottlenecks generated by the target applications in the workload.

Apple Shark Software and hardware profiling tool made by Apple Inc.

Intel Pin Tools Software profiling tool made by Intel Inc.

3.3.1 Apple Shark

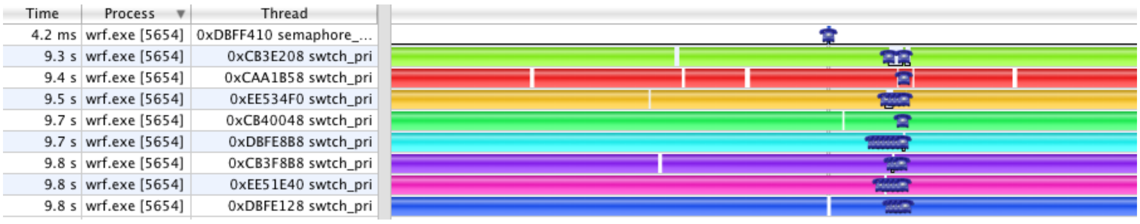
Shark [5] is Apple's primary performance analysis tool. Shark is designed to make it easy for a developer to profile applications to better understand their behavior. To do so, Shark has various configurations to show performance characteristics of applications during runtime both from a hardware and software point of view. Shark also has the ability to monitor hardware performance monitor counters (PMCs) of the underlying architectures, providing detailed information about the state of the system, such as monitoring the L2 cache miss ratio or the memory bandwidth usage between the memory controller and the processors. The following subsections describe the various configurations of Shark.

3.3.1.1 Time Profile

We utilize Shark's "Time Profile" to do statistical sampling of the target application to see which functions the application executes most often. The application is halted at evenly spaced time intervals and Shark records the function the CPU is currently executing. Shark then gives an overview of where most of the computational time is spent. A secondary configuration, "Time Profile (All Thread States)," profiles even blocked or sleeping threads. Information about hot loops, blocking and spinlocks can be determined from these profiles. These events tell you how an application performs on a system, and where possible problems are located.

3.3.1.2 System Trace

The System Trace configuration records all user/kernel transitions that happen on the entire system during the time the trace is recorded. Shark logs system calls, interrupts, virtual memory faults, and thread scheduling decisions. By looking at this information, we can understand how the target application interacts with OSX and see inter-thread synchronization behavior for multi-threaded applications. Understanding where blocking and synchronization occur can be done by looking in the timeline view of System Trace, as shown in Figure 3.4. The phone icons are system calls that indicate special events, in this case synchronization with the other threads. The example shows a barrier synchronization event, where all the threads are waiting for the red colored thread to complete before continuing.



Time	Process	Thread
4.2 ms	wrf.exe [5654]	0xDBFF410 semaphore_...
9.3 s	wrf.exe [5654]	0xCB3E208 swtch_pri
9.4 s	wrf.exe [5654]	0xCAA1B58 swtch_pri
9.5 s	wrf.exe [5654]	0xEE534F0 swtch_pri
9.7 s	wrf.exe [5654]	0xCB40048 swtch_pri
9.7 s	wrf.exe [5654]	0xDBFE8B8 swtch_pri
9.8 s	wrf.exe [5654]	0xCB3F8B8 swtch_pri
9.8 s	wrf.exe [5654]	0xEE51E40 swtch_pri
9.8 s	wrf.exe [5654]	0xDBFE128 swtch_pri

Figure 3.4: Synchronization behavior viewed in System Trace

3.3.1.3 Processor bandwidth profile

With the Processor bandwidth profile, we can get an overview of how applications communicate between the CPUs and use main memory. When we know the practical bandwidth limitations of the hardware, we can see if the application is regularly hitting this hardware bandwidth ceiling, as is shown by Figure 3.5. This ceiling is mostly displayed as occasional peaks if the application does burst accesses with high bandwidth requirements, or as plateaus if there is a continuous access pattern to the main memory or between the CPUs. Since there are different types of memory operations, they may also have different bandwidth ceilings. Typically a store operation has a higher bandwidth ceiling because it does two memory operations, first a load of the old cache line and then a store of the data, which can be done in parallel with subsequent CPU operations. A load by itself has a lower bandwidth ceiling because it only fetches data from memory and often has dependent operations immediately following that must be performed sequentially. In Figure 3.5, we show the bandwidth limits for the different memory access patterns used by Torque (see section 2.3.3). The Load (reading a continuous portion memory) test tops out around 5693 MB/second, while the Store (reading a continuous portion of memory and writing it to another location) test has a higher bandwidth ceiling of 7347 MB/second. The other two memory patterns, Bcopy (copying bytes from one memory location to another) and Bzero (clearing a continuous portion of memory with zeros), have a bandwidth limit similar to the Load test, 5702 MB/second and 5141 MB/second, respectively.

3.3.1.4 L2 cache miss profile

We used a special Shark configuration to profile the ratio of level 2 cache misses to the number of level 2 cache requests to see if the application generates a lot of cache misses. We set up the hardware performance counters in the Core 2 architecture to trigger (generate a sample point) every 100 L2 cache misses. When the trigger is fired, that

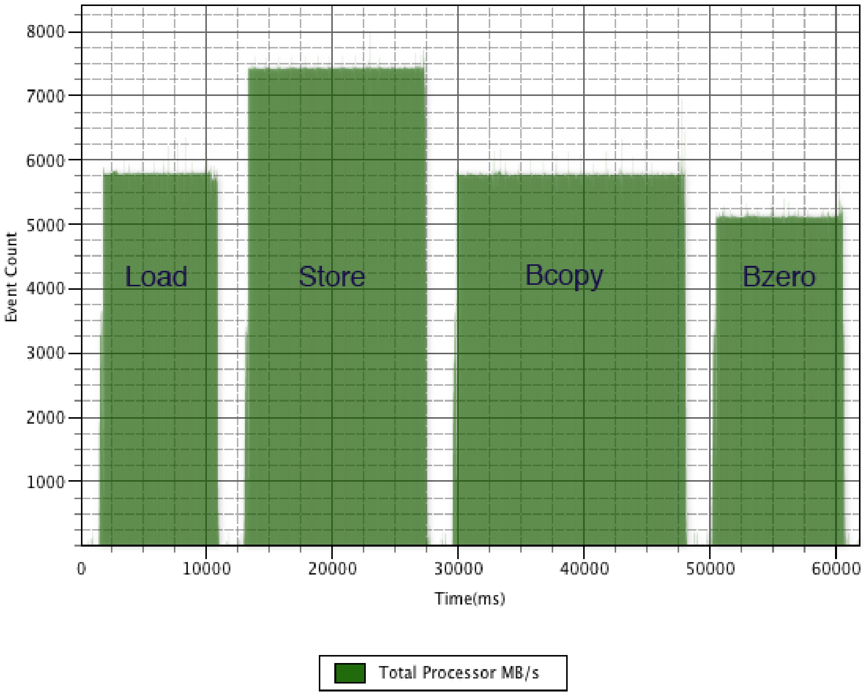


Figure 3.5: Bandwidth profile with different Torque memory access patterns

processor halts and logs the current count of the total number of L2 requests between the previous trigger and the current one. The trigger is then reset and that processor continues executing the application again. Because the sampling is trigger based, the various cores log datapoints at different times. The ratio of L2 cache misses / L2 cache requests is the systemwide cache miss ratio. Unfortunately, these event counters are not thread-based but core-based, so they count everything that the core does, whether or not it is running our workload. Shark's sampling interferes with the state of the L2 cache and flushes the L2 Cache when it needs to write blocks of data out of the kernel. The influence of this limitation on the statistics is discussed in section 3.4

When we combine the profiles collected using Shark, we are able to categorize the applications to see if they will fulfill the criteria set for the workload. For example, *Time Profile* and *Time Profile (All thread states)* allow us to see the behavior of the applications during runtime and to see if we are actually stressing all the available cores. Idle time cause by (spin)locks, blocking or sleeping threads will be visible in these profiles, along with a percentage of the total runtime that this happens. The hardware performance profiles, L2 cache miss profile and processor bandwidth profile, give a low level overview of the hardware bottlenecks. With an L2 cache miss profile, we can spot how the caches are being utilized. From the processor bandwidth profile, we get a nice overview of the bandwidth requirements for the memory subsystem caused by the target application.

3.3.2 Intel Pin Tools

Intel's Pin Tools [21] allow developers to dynamically instrument their applications, modifying the actual behavior without modifying their source code. The Pin Tool does not rewrite the original source code to enable profiling; instead it dynamically injects instrumentation code at run time. This allows Pin Tools to profile already executing applications. Pin Tools has several supplied configurations to give the user basic information about an executing process. An example is where the instruction count is recorded during execution via tracing. While this will give an exact trace of the execution behavior of the target application, it is extremely slow compared to uninstrumented execution. For small applications this is not a real problem, but for large workload sets, running a full trace is not feasible.

Issues There are several issues with the Intel Pin Tools. The Intel Pin Tools currently do not support 64-bit binaries for OS X. Additionally, several bugs have been found with the way pin tools handle double precision floating point numbers. Possible workarounds for 64-bit applications are to compile them for the Linux OS with the same compiler (Intel ifort/icc) and see if there is a large difference between the two assembly traces. If there is little difference, we can assume that the execution of the application code that does not rely on OS support is roughly the same on both operating systems.

3.3.3 Summary

We chose to use the Shark toolset to do the profiling of the target applications because of the ease of use and extensive flexibility of the tools. Along with in-house support from the developers, we were able to profile all the needed information. The issues with the Intel Pin Tools prevented us from fully using the capabilities of that tool set. Another downside of the Intel Pin Tools is that the dynamic instrumentation is too slow to do full workload analysis, while in Shark the support for hardware performance monitor counters (PMCs) enabled us to do performance data collection at native execution speed.

3.4 Influence of analysis on results

When profiling an application with hardware performance monitor counters (PMCs), we affect the performance of the application because we are interrupting the application to gather information. This phenomenon is similar to the Heisenberg uncertainty principle for quantum physics [17], where probing particles to take measurements changes what you are measuring. To calculate the error created by sampling the application, we look at the overhead the profiling causes and possible alterations in the environment on a hardware level. To demonstrate possible worst case effects from overhead, we use the Linpack benchmark discussed in section 4.1 to show the performance effects caused by the different configurations that we use in Shark.

3.4.1 L2 cache miss profile

When profiling Linpack for L2 cache misses, we interrupt the application after every 100 cache misses to gather statistics on the total number of cache misses and requests. Because Shark stores its data in the kernel and needs to write the data to user space after every 2MB of samples, we come across the problem that every time the data needs to be written from the kernel space to the user space, it flushes the entire 2MB L2 cache. This fully flushed cache will cause otherwise unnecessary cache misses until the cache is repopulated again with part of the application's dataset. The frequency this will happen is dependent on the fill-rate of the Shark's 2MB buffer. To calculate the frequency of a full cache flush we need to find out how many samples of statistical information will fit into the 2MB kernel buffer. For Linpack we have the following equation (3.1) to calculate the number of samples that will fit in the 2MB kernel buffer:

$$\begin{aligned} \frac{2MB}{internal\ storage\ element\ size} &= \\ \frac{2MB}{chud\ header\ 56\ byte + 8\ Byte * PMCs + 8\ Byte * average\ call\ depth} &= \\ \frac{2MB}{56\ byte + 8\ byte * 2 + 8 * 1} &= \\ \frac{2MB}{80\ Bytes} &= \\ 26214\ samples\ per\ dump & \end{aligned} \tag{3.1}$$

The chud header in this equation is added to each sample to describe the contents of the sample. The PMCs can return up to 64-bit numbers and therefore are stored in 8 Byte segments. The call stack contains an actual backtrace of the functions called to get to the currently executing call. In our case the call stack depth for Linpack is 1 because of a single call to the Intel Math Library MKL to do the processing. This means that for every 26214 samples the cache will be fully flushed, resulting in false cache miss counts during this refill time. The number of 64 byte cache lines to be refilled per 2MB cache:

$$\frac{2MB}{64\ Byte\ cache\ line\ width} = 32768\ lines \tag{3.2}$$

Since we generate a sample for every 100 cache misses, this results into 328 false samples per dump. Hence, the total number of faulty cache misses is calculated in the following way:

$$\frac{(\frac{total\ number\ of\ samples}{29127\ samples\ per\ dump}) * 328}{total\ number\ of\ samples} \tag{3.3}$$

For our Linpack example we collected 735401 L2 cache miss samples during the run of our experiment, this will give us the following equation:

$$\frac{(\frac{735401\ samples}{29127\ samples\ per\ dump}) * 328}{735401\ samples} = 9184\ false\ samples \tag{3.4}$$

This results into faulty miss rate of 1.25%, as shown by equation 3.5

$$\frac{9184\ false\ samples}{735401\ samples} = 1.25\% \tag{3.5}$$

3.4.2 Time Profile

Time Profile overhead is calculated using a methodology similar to the L2 cache miss profile calculation. The main difference is that Time Profile does not use PMCs for measurements, so that value in the formula is 0. Another important difference is that the cache misses we cause because of the flushing of the kernel space buffer do not affect the metric we are looking for (in this case for Time Profile total run time) in the direct way that L2 cache misses did for L2 cache miss profile. For Time Profile, we need to calculate the overhead of the extra latency caused by access to main memory compared to L2 cache. Equation (3.6) is similar to equation (3.1) for the same linpack application, resulting in the following:

$$\begin{aligned} \frac{2MB}{internal\ storage\ element\ size} &= \\ \frac{2MB}{chudheader\ 56\ byte + 8\ Byte * PMCs + 8\ Byte * average\ call\ depth} &= \\ \frac{2MB}{56\ byte + 8\ byte * 0 + 8 * 1} &= \\ \frac{2MB}{64\ Bytes} &= \\ 32768\ samples\ per\ dump & \end{aligned} \tag{3.6}$$

It will take 32768 samples before the buffer needs to be flushed, resulting in a cleared cache. Assuming the worst case for linpack, with the entire cache filled without any prefetching by the memory controller, this will result in an increase of memory latency from 5ns L2 cache accesses to 100ns main memory accesses. Knowing that we sample every 1ms, it will take 32.8 seconds to fill up the kernel buffer before a flush. Also, to fill the cache again after a flush we need to fill the 32768 cache lines with accesses from main memory resulting in 32768 * 100ns = 3.2768ms overhead per flush. This results into a total overhead of 3.2768 ms / 32.768 seconds = 0.01%, well into the background noise caused by other effects. The actual time spent in processing inside the kernel and Shark is subtracted from the profile, so this overhead is not visible.

3.4.3 Time Profile (All Thread States)

Time Profile (All Thread States) uses a different information gathering technique. It requests information from the kernel about the current available threads (running, waiting, sleeping etc.). This will cause the kernel to suspend all the threads from the target application and log the call stacks for each thread out to user-space. For every sample interval (1ms) the number of threads which are part of the target application times the sample size are flushed. For our Linpack example, we can use the same formula as before: 64Byte sample size * 9 threads = 576 Bytes per ms. This translates to 9 cache lines to be flushed out per sample when running on 8 cores, well within the noise of normal computation. Like Time Profile, the overhead caused by running the kernel processing and Shark is subtracted from the total time and does not show up in the final performance analysis profile.

3.4.4 Processor Bandwidth

Processor Bandwidth uses the same system as the L2 cache miss profile to calculate the memory traffic generated by the processors. It uses two PMCs to gather the data, resulting in 26214 samples per dump of the kernel buffer. This will happen every 26214 * 1ms samples = 26.214 seconds. The difference is that for the Processor Bandwidth profile, the *total* amount of traffic is important, and not just cache misses caused by the flushing of the cache when the kernel buffer needs to be emptied. The resulting overhead in traffic caused by this flushing is the number of cores * 2 MB cache per 26.214 seconds = 8 * 2MB = 16MB of traffic overhead. We know that Linpack memory traffic runs into the thousands of MB/second, as shown in figure 4.2, so this overhead is negligible.

3.4.5 System Trace

Since System Trace provides an exact trace of the events that happen in the entire system, its buffer is limited to a fixed number of events. These events have no specific time-based interval, so the sampling is considered non-deterministic and will occur randomly. The sampling of events is handled by the kernel and when call stack tracing is disabled, no execution overhead is present for the target application.

4 Scientific Benchmarks

One of the domains we chose to look at for finding suitable applications was the High Performance computing domain. High Performance Computing applications are primarily executed on supercomputers or clusters. The target applications for these large systems are primarily scientific and make considerable use of large matrices. These matrices tend to be very amenable to being sub-divided into smaller datasets that can be distributed to different cores. Since these systems are built with multiple CPUs in one or multiple systems, the codes used to run on these systems are generally designed to be highly parallel.

4.1 Linpack

Linpack [11] is one of the most well-known floating-point benchmarks for high performance systems. The application is designed to solve dense systems of linear algebra problems either in single precision or double precision floating-point using various Basic Linear Algebra Subprogram (BLAS) 1, 2 and 3 routines. This makes the Linpack a perfect example of the Dense Linear Algebra Dwarf, Dwarf 1, explained in section 2.1. Linpack is useful to calculate the practical peak floating-point performance of a specific architecture. The main algorithm used is Gaussian Elimination with partial pivoting.

4.1.1 Scalability

Linpack was designed to be a computationally intense application. From the original serial application, multi-core capable versions were created using both MPI and OpenMP. Because the test machine has a cache coherent Uniform Memory Access (ccUMA) memory system, we chose to use the OpenMP version to take advantage of the shared memory address space. Using OpenMP, we were easily able to scale from a single core to 8 cores, as is shown by Figure 4.1. The time required for calculating the matrix is significantly reduced by doubling the number of cores (respectively 186% faster for 2 cores, 329% for 4 cores and 540% for 8 cores compared to the use of a single core), while the average GFLOPS/Second increased (respectively 193% more for 2 cores, 364% for 4 cores and 664% for 8 cores compared to the use of a single core).

When looking at the Shark Time Profile of our run on a 23k by 23k matrix, 73% of the total wall clock time is spent in a single computational loop function called *N4_M4_LOOPgas_1*. The Time Profile showed that there is an average of 15% overhead in thread synchronization. This is due to the barriers implemented in the code to periodically synchronize between the cores. Synchronization barriers can cause threads to stop executing while waiting for all the other threads that are part of the barrier to complete. In contrast synchronization does not exist in sequential code. This synchro-

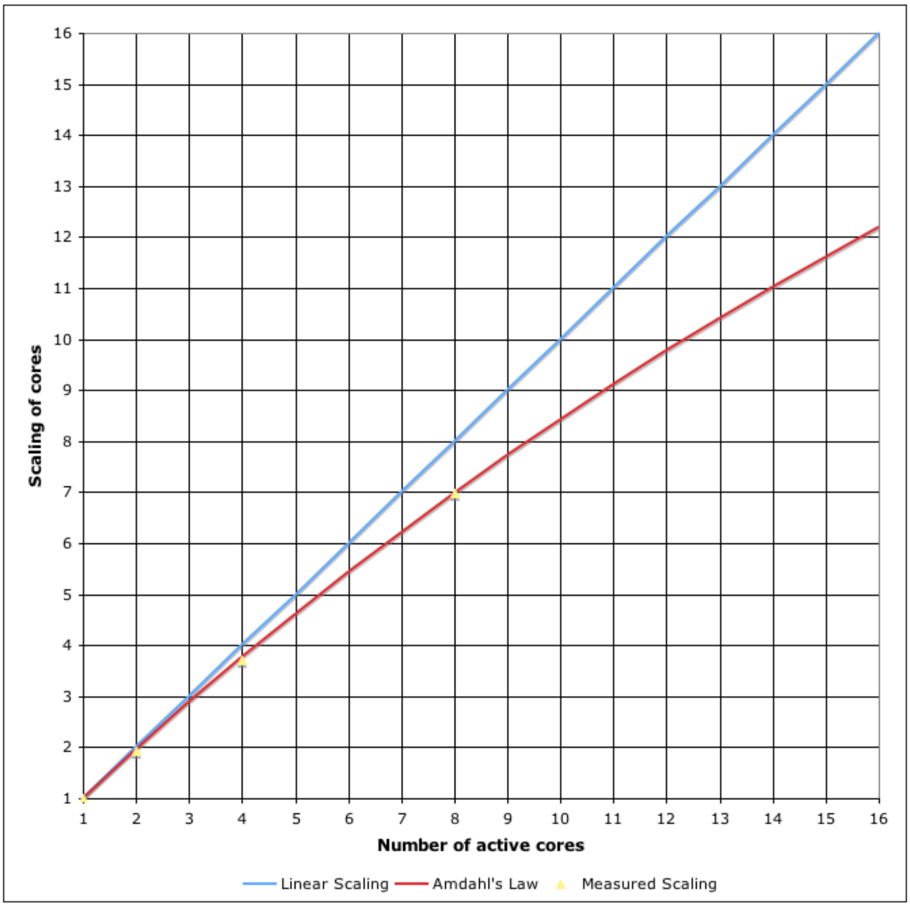


Figure 4.1: Scaling of Linpack depending on number of cores

nization time adds to the execution overhead that parallel code has over sequential code. Synchronization overhead normally increases monotonically as a function on the number of processors p.

The processor bandwidth profile showed that there were bursts of memory activity reaching the bandwidth ceiling of the architecture and several distinct execution phases that were limited by the CPU. Overall, the application is heavy on Front Side Bus bandwidth, due to its large and hard-to-cache memory footprint. Using the Processor Bandwidth profile, we can determine the separate phases of the application based on the bandwidth required and the type of functions being executed at that time. These phases are obvious in Figure 4.2.

Figure 4.2 identifies the following computational phases in the linpack benchmark:

1. *Setup:* During the first 8.9 seconds, the application only runs with the master thread,where it generates the matrix.

2. *(8.9 seconds to 21.5 seconds):* The master thread spawns multiple worker threads where about 88% of the time is spent doing computation and 9% of the time is spent synchronizing between the threads.

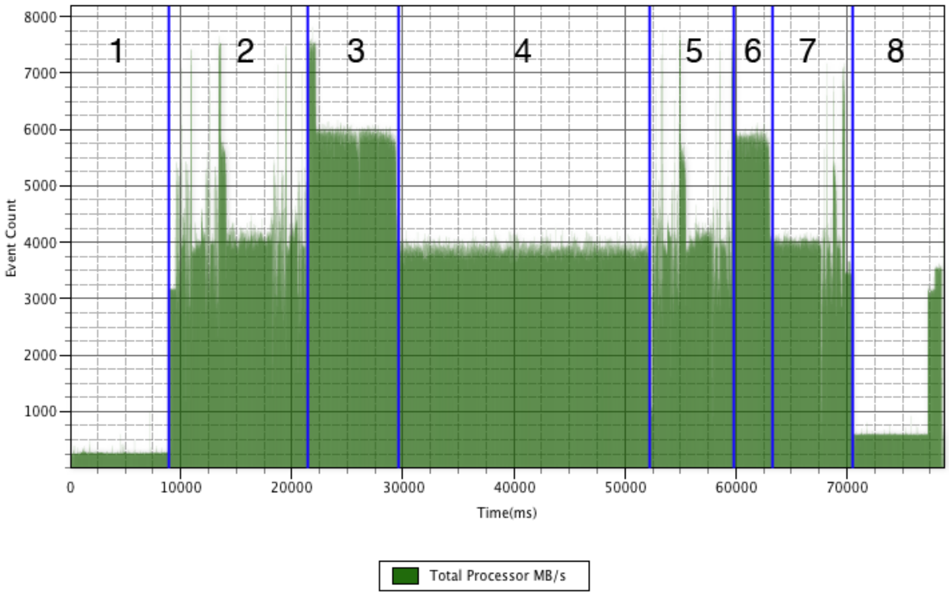


Figure 4.2: Processor Bandwidth profile with a sample rate of 10ms

3. *(21.5 seconds to 29.4 seconds):* There is a large increase in bandwidth usage and very little synchronization. 99% of the time is spent doing computation with large dedicated data moves in the *Y_ADDR_SET_Agas_1* function. This function causes a significant increase in bandwidth usage because it mostly just moves data around and does little computation.

4. *(from 29.4 seconds to 52.3 seconds):* This is computation intensive (99.6%) and does not have the dedicated data transfer requirements of *Y_ADDR_SET_Agas_1*. Therefore, the bandwidth usage is limited by time spent in computation functions. At the end of this function, there is a sudden drop in bandwidth, which can be explained by the presence of a barrier to synchronize the execution at the end of this computational segment.

5. *(from 52.3 seconds to 59.6 seconds):* This is similar to the second segment. About 84% of the time spent is in computation, but the *Y_ADDR_SET_Agas_1* function causes data bursts for 6% of the time and the drops in bandwidth are caused by synchronization stalls for 8.4% of the total wall time in this segment.

6. *(59.6 seconds to 63.1 seconds):* This is like the third segment, with a peak caused by use of *Y_ADDR_SET_Agas_1*. There is little synchronization between the threads in this segment and 97.5% of the time is spent in computational functions.

7. *(63.1 seconds to 70.1 seconds):* This is again similar to section four, but has peaks and drops caused by 4.7% of the time spent in calls to *Y_ADDR_SET_Agas_1* for data transfers and 5% synchronization overhead, while 87.6% of the time is spend in computational functions.

8. *Reduction of results:* Finally, this is where the master thread collects all the data and generates the final result matrix.

4.1.2 Validation

4.1.2.1 Input validation

The matrix is pseudo randomly generated when the application runs, based on the requested matrix size. The random numbers are generated by the *DLARAN* function, which is part of the LAPACK package.

4.1.2.2 Output validation

The Linpack application has its own internal verification process based on computing the residuals of its randomly generated matrixes. When there is a computational error, the application will terminate and the results will be marked as invalid.

4.1.3 Criteria overview

- Multi-core: Because we use OpenMP to supply the multi-threading support, the application is capable of using all the available cores.
- Scalable: Section 4.1.1 described that there is an average overhead of 15% due to thread synchronization. Figure 4.1 also showed that Linpack is capable of scaling when you increase the number of cores. Although the scaling is sub-linear, the rate of decline is not significant enough to suspect poor scalability when used on systems with up to about 32 cores.
- Reproducible: There is a variation in the results reported by Linpack. We remove possible outliers using the technique discussed in Section 2.3.4. We make sure that the coefficient of variation (CV) is minimal. A CV of 2% is acceptable to ensure reproducible results.
- Verifiable: As discussed in Sections 2.3.4, the input is generated pseudo randomly and should not differ when we change platforms. The output is verified by the application itself and will produce an error if the result is faulty.
- Runtime: The runtime of this application will differ based on the speed of the underlying machine. On our test 3.0 Ghz Mac Pro, the application ran for 83 seconds for single precision and 143 seconds for double precision floating point. To get reproducible results, we run 10 iterations each to remove the outliers, resulting in a reasonable runtime of under 40 minutes for both.

4.2 NASA Parallel Benchmark

The NASA Parallel Benchmark (NPB) is a collection of programs extracted from important aerospace applications and designed to mimic a class of computational fluid dynamics (CFD) applications. The NPB is based around 5 core CFD kernels and 3 simulated CFD applications commonly used by NASA to solve aero-physics problems. We selected the NPB 3.3-OMP [19] with OpenMP support to run on our test system, since OpenMP can use shared memory to communicate with other threads while the MPI version cannot. NASA modified the original serial implementations to include compiler directives around the data structures and loops so that the run-time library can easily parallelize the applications while executing. For more information about the implementations and the algorithms used in these benchmarks we refer you to “The NAS Parallel Benchmark Overview” [7].

4.2.1 Simulated Computational Fluid Dynamic applications: BT, SP and LU

- BT Block-Tridiagonal: Simulated CFD application that solves 3D Navier-Stokes equations, resulting in Block-Tridiagonal 5x5 blocks. Applicable Dwarf: 1.
- SP Scalar Pentadiagonal: Simulated CFD application that solves the finite differences solution based on Beam-Warming approximate factorization, resulting in Scalar Pentadiagonal bands of linear equations. Applicable Dwarf: 5.
- LU Lower Upper: Simulated CFD application that solves 3D Navier-Stokes equations using symmetric successive over-relaxation (SSOR), resulting in block Lower and Upper triangular systems. Applicable Dwarf: 1.

4.2.1.1 Scalability

BT is a straightforward application that consists of 5 main functions: *compute_rhs* calculates the Right-Hand-Side of the equations and takes 49.6% of the total processing time, while block-tridiagonal systems are solved for each direction, requiring 42.8% of the total processing time (respectively *x_solve*(13.5%), *y_solve*(14.6%) and *z_solve*(14.7%)). Finally, the solution is updated by the *add* function, taking 4.2% of the total processing time. We spend 2.6% of our time in the OpenMP synchronization methods and the remaining time is spent in various system functions (0.8%). During the experiment BT occupied 97.9% of all the available cpu time, with 1.6% spend it the system idle thread mach_idle and the remaining 0.5% distributed among the various background daemons.

When looking at the Processor Bandwidth Profile, Figure 4.3, we can see that there is a bursty pattern throughout the entire duration of the application. This consists of the 5 different stages of the application that constantly repeat. Figure 4.4 shows a enlarged portion of the Processor Bandwidth profile for a single iteration. The main functions are labeled in the following order:

1. *compute_rhs*
2. *x_solve*
3. *y_solve*
4. *z_solve*

5. *add*

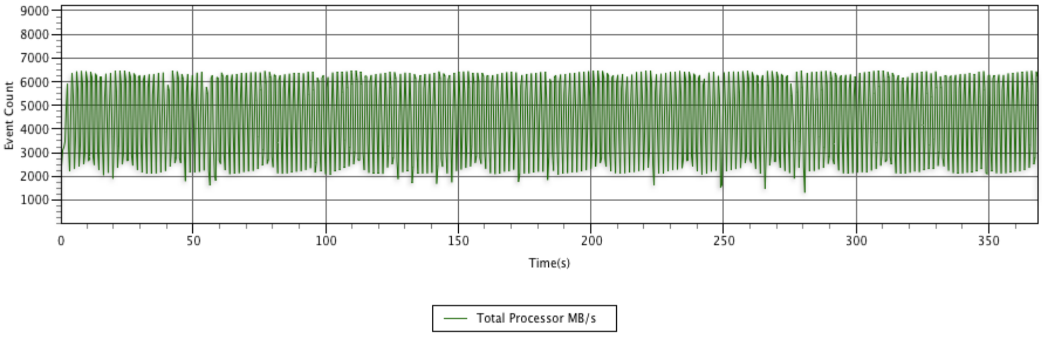


Figure 4.3: Processor Bandwidth profile of NPB BT

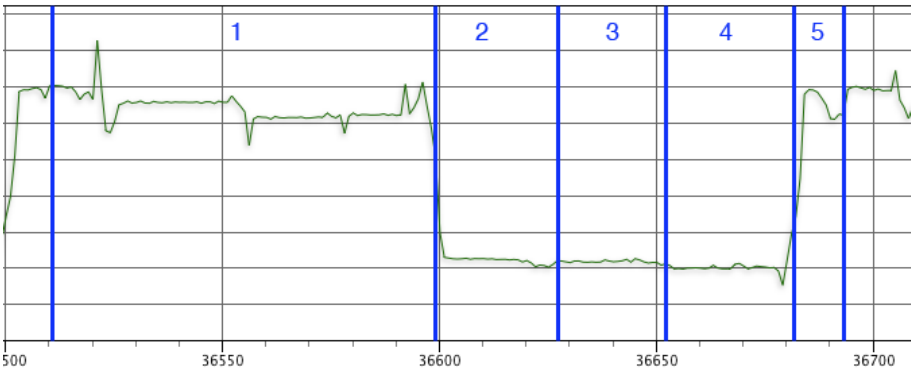
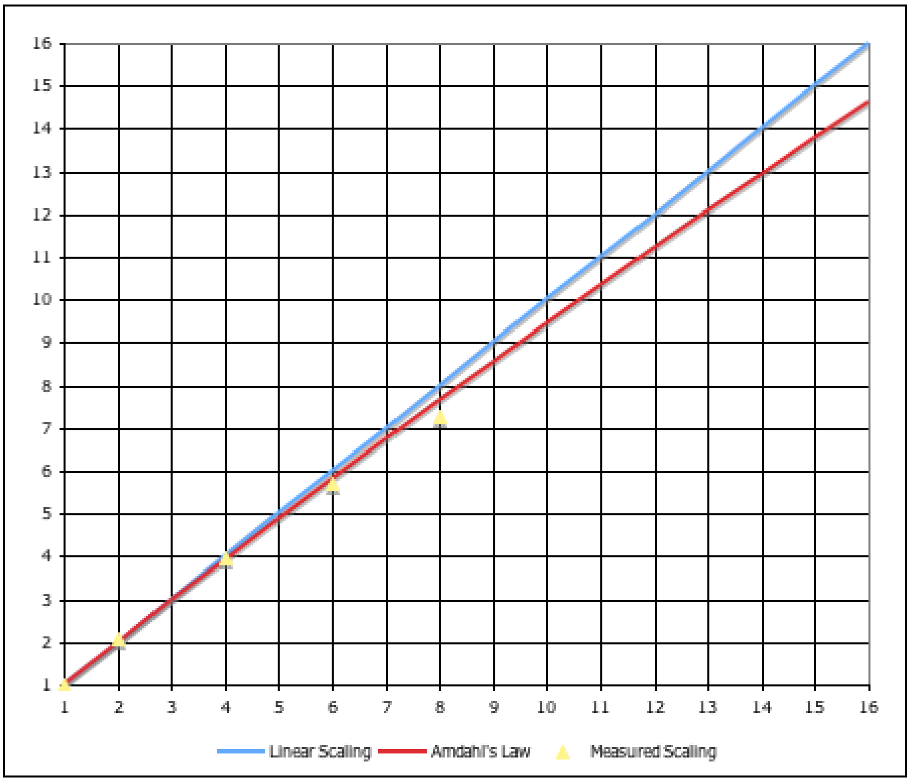


Figure 4.4: Zoomed in Processor Bandwidth profile of BT

When looking at the derived Amdahl scaling, shown in Figure 4.5, we can see that the scaling efficiency drops off when we increase the number of cores. When looking at the Karp-Flatt metric in Table 4.1, we can determine that the decrease in efficiency is due to increasing synchronization overhead.

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	1336			1.00	
2		687	1.95	0.97	0.02789
4		369	3.62	0.91	0.03471
8		192	6.97	0.87	0.02111

Table 4.1: Calculating the scalability of NPB BT



*

Figure 4.5: Scaling of BT depending on number of cores

SP performs a computation very similar tor BP, but the method of approximate factorization is different. SP has 9 main computational phases that make up the approximate factorization algorithm: *compute_rhs* calculates the Right-Hand-Side of the equations and takes 33.9% of the total processing time, an approximate factorization solver for all directions that requires 49.7% of the total processing time (respectively *x_solve*(16.9%), *y_solve*(16.2%) and *z_solve*(16.5%)), and the block diagonal matrix vector multiplication stages for each direction (10.4% of the total computational time; (*txinvr*(3.0%), *pinvr*(1.7%), *ninvr*(1.7%) and *tzetar*(3.9%) respectively). The solution is updated by the *add* function, which takes 2.8% of the total computational time, at the end of the each cycle. The synchronization overhead for running with 8 cores is 2.5%, with the remaining 0.7% used for system functions. During the experiment SP occupied 97.8% of all the available cpu time, with 2.1% spend it the system idle thread mach_idle and the remaining 0.1% distributed among the various background daemons.

Looking at the Processor Bandwidth profile, Figure 4.6, we can see that SP manages to saturate the available memory bandwidth completely over the entire duration of the application. Figure 4.7 shows a enlarged portion of the Processor Bandwidth profile for a single iteration. The main functions are labeled in the following order, with the block diagonal stages left out because they are to small to plot separately:

1. *compute_rhs*

2. *x_solve*
3. *y_solve*
4. *z_solve*
5. *add*

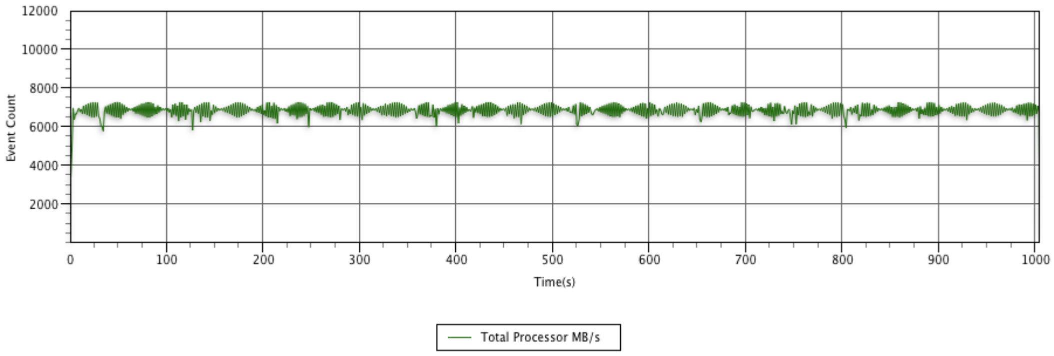


Figure 4.6: Processor Bandwidth profile of NPB SP

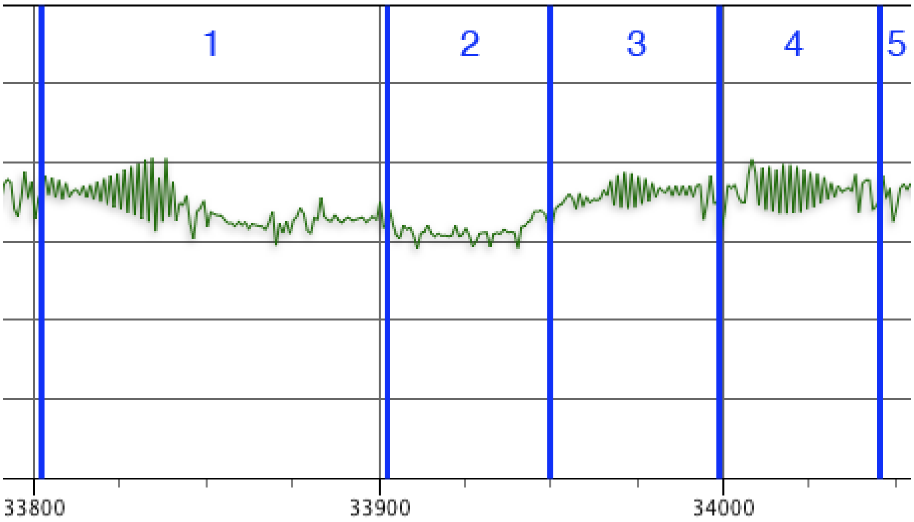
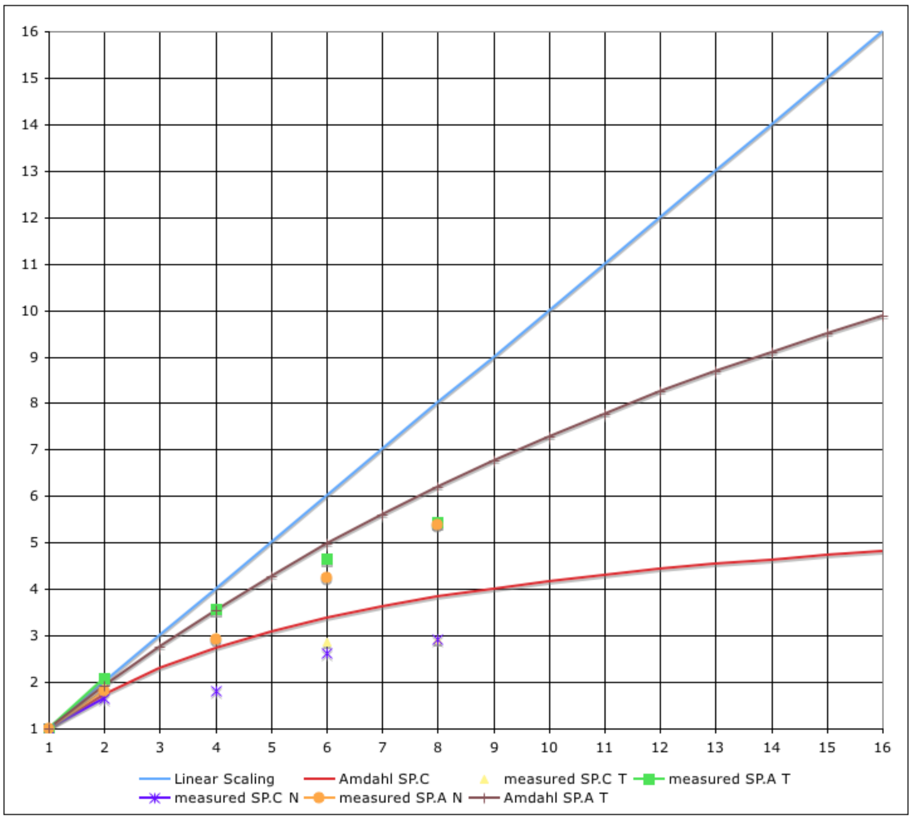


Figure 4.7: Zoomed in Processor Bandwidth profile of SP

When looking at the Amdahl scaling, shown in Figure 4.8, we can see that during measurements going over 4 cores we reach a bottleneck and scaling beyond that does not return significant performance enhancements. Time Profile did not show us any significant increase in synchronization overhead for scaling to more cores. The scaling numbers for 4 to 8 cores point to a bottleneck in either the memory bandwidth being saturated at 4 cores or cache coherency issues. To verify this we ran the SP with a significantly smaller dataset, and the performance did increase while the bandwidth still was being saturated pointing towards cache interference caused by the larger dataset. The Karp-Flatt metric, Table 4.2, the rapidly increasing serial fraction agrees with what we have seen for the scaling efficiency pointing to significant overhead, in this case due to the cache interference.



*

Figure 4.8: Scaling of SP depending on number of cores

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	845			1.00	
2		433	1.95	0.98	0.02555
4		272	3.10	0.78	0.09618
6		272	3.10	0.52	0.18677
8		266	3.17	0.40	0.21722

Table 4.2: Calculating the scalability of NPB SP

LU has 5 main computational phases that make up the SSOR algorithm: *compute_rhs* calculates the Right-Hand-Side of the equations and takes 14.8% of the total processing time, the lower-triangular and diagonal systems are formed in the *JACLD* phase taking 20.1% of the total computational time, with the solver in the *BLTS* phase requires 13.2% of the time. The upper-triangular system is created and solved by the *JUCA* and *BUTS* phases, taking 25.4% and 13.7% of the total computational time respectively. The result is finally updated burning 8.5% of the time. Thread synchronization requires 1.1% of the computation time, and the remaining 0.3% is used for various system functions. During the experiment LU occupied 98.7% of all the available cpu time, with 1.2% spend it the system idle thread mach_idle and the remaining 0.1% distributed among the various background daemons.

The Processor Bandwidth Profile, Figure 4.9, shows us that there is a repetitive high bandwidth pattern throughout the duration of the application. This matches the computational phases the application goes through in an iterative manner. LU is capable of pushing the memory traffic to the bandwidth ceiling of our experiment system. An enlarged Processor Bandwidth profile is shown in Figure 4.10, where it shows a single iteration of the main computational phases and labels in the following order:

1. *compute_rhs*
2. *JACLD / BLTS*
3. *JUCA / BUTS*
4. *add*

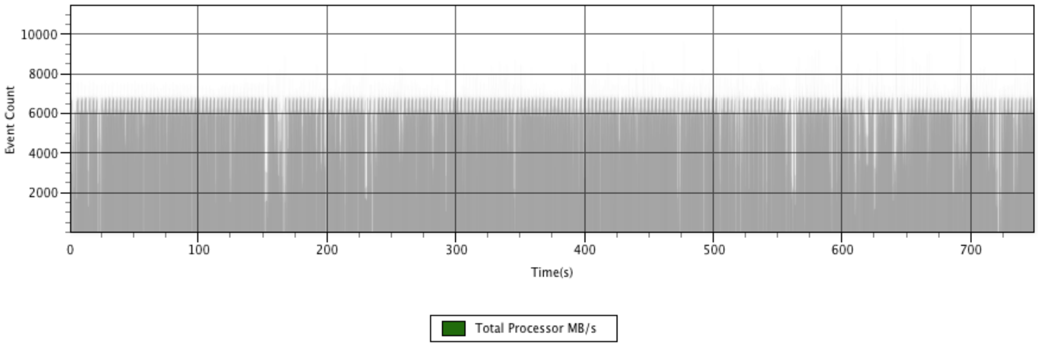


Figure 4.9: Processor Bandwidth profile of NPB LU

The derived Amdahl scaling, shown in Figure 4.11, tells us that the efficiency declines with more cores and the average efficiency is 89% per core added. The Karp-Flatt metric, Table 4.3, reveals that we have a super-linear speedup going from 1 to two cores primarily cause by the extra cache added when we went form 1 to 2 cores. The serial fraction is increasing when we add more cores to the system while the efficiency is dropping, this points to an increase in synchronization overhead.

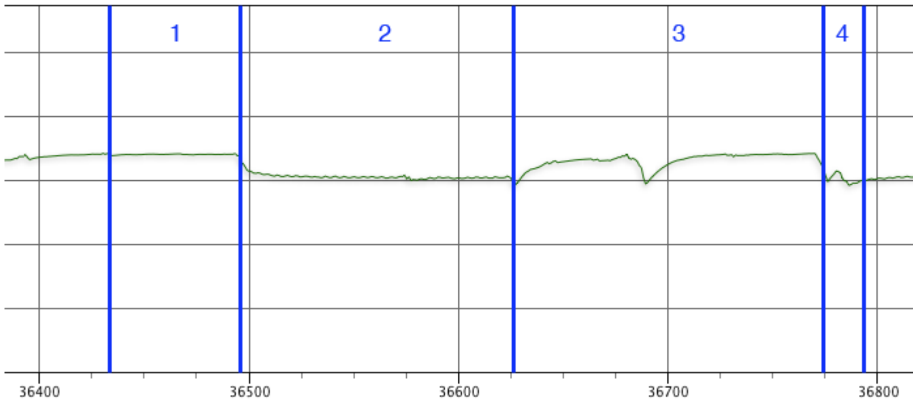
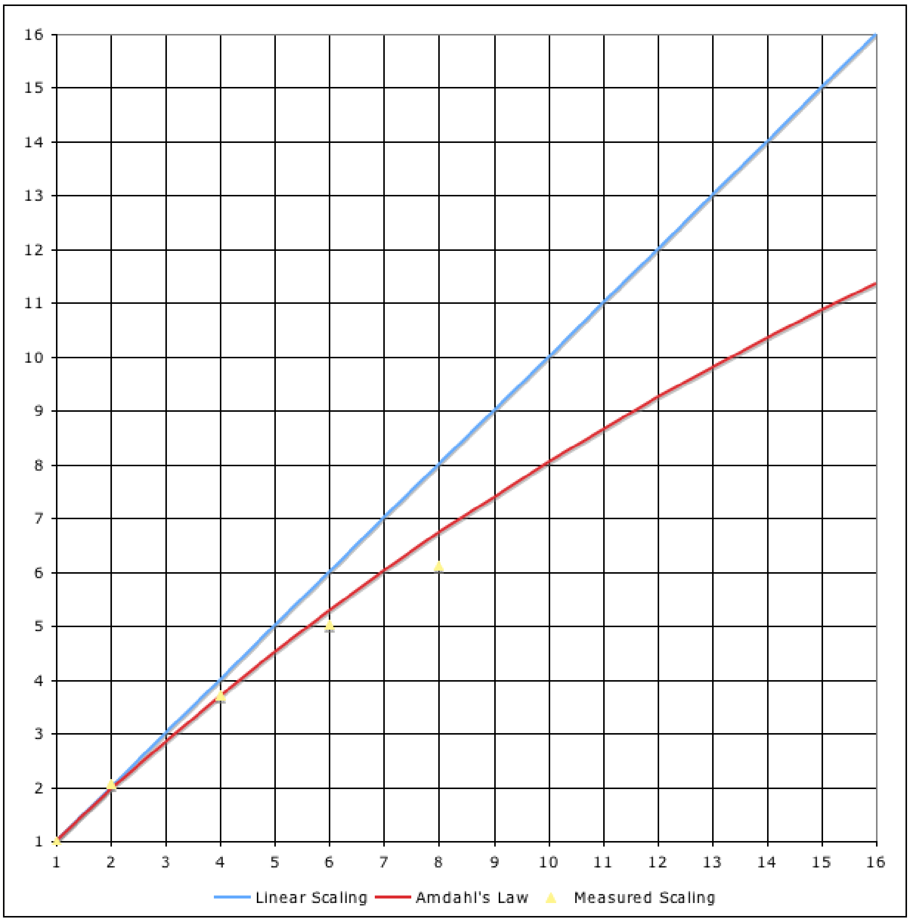


Figure 4.10: Zoomed in Processor Bandwidth profile of LU



*

Figure 4.11: Scaling of LU depending on number of cores

	Speedup and Efficiency				Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	1355			1.00	
2		656	2.06	1.03	-0.03098
4		368	3.68	0.92	0.02873
6		270	5.03	0.84	0.03874
8		221	6.14	0.77	0.04325

Table 4.3: Calculating the scalability of NPB LU

4.2.1.2 Validation

Input validation: All of the benchmarks mentioned above (BT, SP and LU) use different numerical approaches to calculate the solution of the system of differential equations shown in Equation 4.1:

$$\begin{aligned}\{I - \Delta\tau[\frac{\partial(A)^n}{\partial\xi} + \frac{\partial^2(N)^n}{\partial\xi^2} + \frac{\partial(B)^n}{\partial\eta} + \frac{\partial^2(Q)^n}{\partial\eta^2} + \frac{\partial(C)^n}{\partial\zeta} + \frac{\partial^2(S)^n}{\partial\zeta^2}]\}\Delta U^n \\ = \Delta\tau[\frac{\partial(E+T)^n}{\partial\xi} + \frac{\partial(F+V)^n}{\partial\eta} + \frac{\partial(G+W)^n}{\partial\zeta}] \\ -\Delta\tau\varepsilon[h_\xi^4\frac{\partial^4U^n}{\partial\xi^4} + h_\eta^4\frac{\partial^4U^n}{\partial\eta^4} + h_\zeta^4\frac{\partial^4U^n}{\partial\zeta^4}] + \Delta\tau H^* \qquad (4.1)\end{aligned}$$

For more information about how this partial differential equation was derived we refer you to “The NAS Parallel Benchmark Overview” [7].

Output validation: For all of the simulated CFD applications mentioned above, the internal verification test calculates the Root Mean Square norms RMSR(m) of the residual vectors and the Root Mean Square norms RMSE(m) of the error vectors. If the ratio between the computed value X_c and the reference value X_r is smaller then the maximum allowable relative error ϵ, the results are valid.

4.2.2 FT: Fourier Transform

Computational kernel of a 3-D partial differential equation (PDE) using forward and inverse Fast Fourier Transform (FFT) techniques known as Swarztrauber’s vectorization of Stockham’s auto sorting algorithm. Applicable Dwarf: 3.

4.2.2.1 Scalability

FT has 3 main execution phases, the first phase is the setup where the pseudo-random number generator generates the required 64-bit floating point numbers needed for the initial input data. During the setup phase a warmup iteration is done to make sure all the data is touched, reducing variations in the startup cost. The setup stage takes 12.8% of the total execution time. In this stage, the *init_ui* function touches all the arrays, while the *compute_initial_conditions* function fills the primary array with the pseudo-random numbers in parallel. The second stage is the actual computational phase and takes up the majority of the computational time where the original 3D-array is passed through the initial Fast Fourier Transform, afterwards the result is passed to an iterative loop. In this loop the exponent factors are calculated for the inverse Fast Fourier Transform and finally checked by a checksum. This procedure takes up 87.1% of the total computational time, divided into 57.5% for FFT computations and 23.3% for the calculation of the exponent factors in the *evolve* function. The computational time for the checksum is small enough to be discarded in the overal calculations. During the iteration loops the synchronization overhead accounts for 6.3% of the total computational time caused by barriers in the OpenMP library. The final phase is the verification stage where the computed checksums are compared to the pre-calculated checksums. This verification step is very short and requires little time to complete. The verification and remaining system functions take up 0.01% of the computational time. During the experiment FT occupied 92.4% of all the available cpu time, with 6.9% spend it the system idle thread mach_idle and the remaining 0.7% distributed among the various background daemons.

The Processor Bandwidth profile, Figure 4.12, shows us the three separate phases outlined in the previous paragraph. The initial setup phase generates a steady amount of memory traffic due to the generation of the pseudo-random numbers, with a warmup routine at the end of this phase. The second phase shows a repetitive memory pattern that conforms with the iterative nature of the Fast Fourier Transforms. The third and final phase is shows a sharp drop in memory traffic due to that only the verification of the pre-calculated checksums has to be done here. Measurements for scaling were only done on the iteration loop of the actual FFT routine. This was done to exclude the initialization phase and verification phase that are not required by real FFT implementations. An enlarged Processor Bandwidth profile is shown in Figure 4.13, where it shows a multiple iterations of the main computational phases and labels in the following order:

1. *evolve*
2. *fft*

The derived Amdahl scaling, shown in Figure 4.14, tells us that the measured datapoints follow our predicted curve but deviate when we reach 8 cores. When going form 6 to 8 cores the memory bandwidth ceiling is limiting us to reach optimal performance for burst transfers, and therefore decreasing the efficiency for 8 cores. The overall efficiency of FT is 71% when running our 8-core test machine. This efficiency tells us that there is still room for adding more cores before the overhead becomes to great and adding more cores will not increase our computational output.

4.2.2.2 Validation

Input validation: The FT benchmark dynamically generates a 3-D matrix with a pseudo-random number-generator that uses π as initial seed value. Because of the

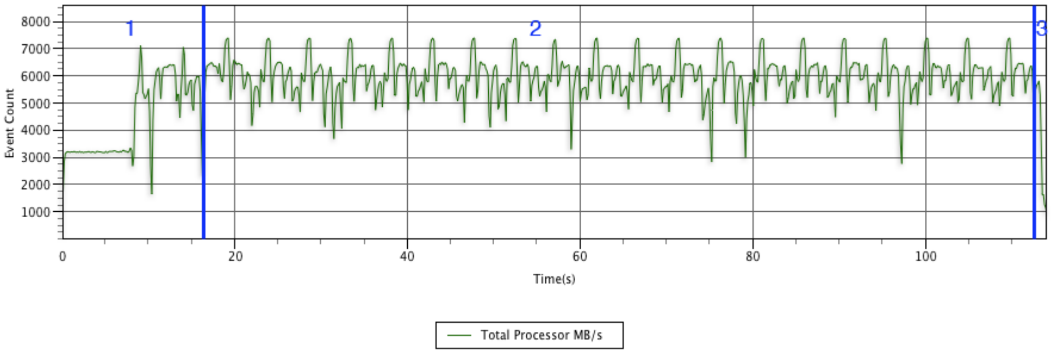


Figure 4.12: Processor Bandwidth profile of NPB FT

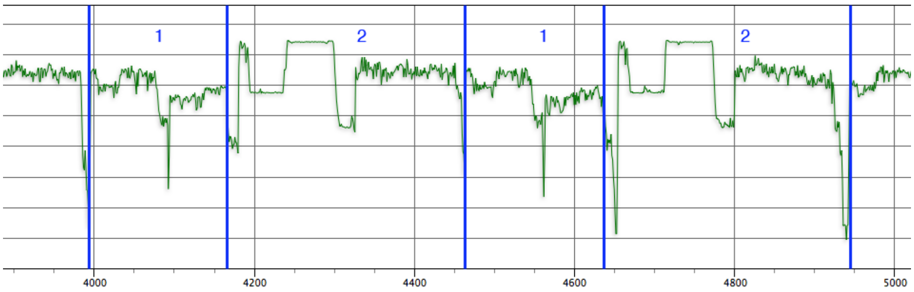


Figure 4.13: Zoomed in Processor Bandwidth profile of FT

pseudo-random number generator, the matrix should always have the same arrangement of values.

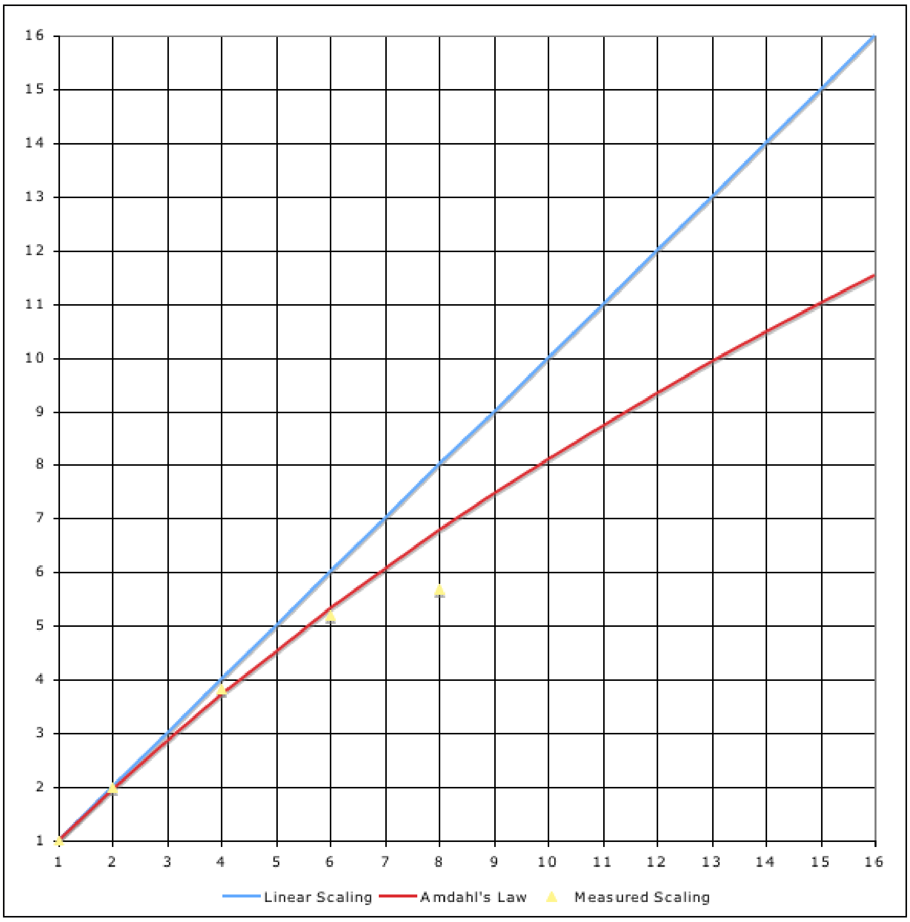
Output validation: The FT benchmark has an internal verification system that will check if the complex checksums are the same as the reference values stored in the application, within a margin of $10^{-10}\%$.

4.2.3 IS: Integer Sort

Computational kernel that does a parallel sort over small integer keys based on the bucket sort algorithm [30]. Applicable Dwarf: 11

4.2.3.1 Scalability

IS has 3 main execution phases. The first is the initial setup, where the integers are generated by the key generation algorithm. This takes up 23.4% of the total execution time. In this stage, the *create_seq* function generates the keys and puts them in one large array. The second phase is the actual sorting stage, and takes up the majority of the computational time. The *rank* function takes up 67.4% of the total computational time. In this *rank* function, there is a barrier to obtain the rank for each key, where the application must add the individual keys before continuing. The synchronization



*

Figure 4.14: Scaling of FT depending on number of cores

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	302			1.00	
2		151	2.00	1.00	0.00002
4		79	3.83	0.96	0.01458
6		58	5.20	0.87	0.03071
8		53	5.69	0.71	0.05801

Table 4.4: Calculating the scalability of NPB FT

overhead for these barriers is 2.8% of the total computation time. The final phase is the verification, where the *full_verify* function goes over the final sorted array and verifies that the keys are sorted correctly. This final verification requires 6.2% of the computation time. The remaining 0.2% is spend in various system functions. During the experiment IS occupied 89.0% of all the available cpu time, with 10.2% spend it the system idle thread mach_idle and the remaining 0.8% distributed among the various background daemons.

The Processor Bandwidth profile, Figure 4.15, clearly shows us the separate phases outlined in the previous paragraph. The initial setup (marked as 1) has lower bandwidth requirements than the sorting stage, with an increase in memory traffic in the second portion of the initialization. This increase in traffic is caused by the kernel failing to allocate a page during a VM fault when its allotted time slice expires, due to locking. This is shown in Figure 4.16, a System Profile of the second part of the initialization phase. The sorting stage starts out with a warmup sort (marked as 2) by running rank once, and then it loops through the rank routine (marked as 3) 10 more times before going to the final verification phase (marked as 4). Measurements for scaling were only done on the iterations of the actual sorting algorithm. This was done to exclude the serial initialization and verification phase from the measurements. They are not scalable, but are also not required by real sorting applications.

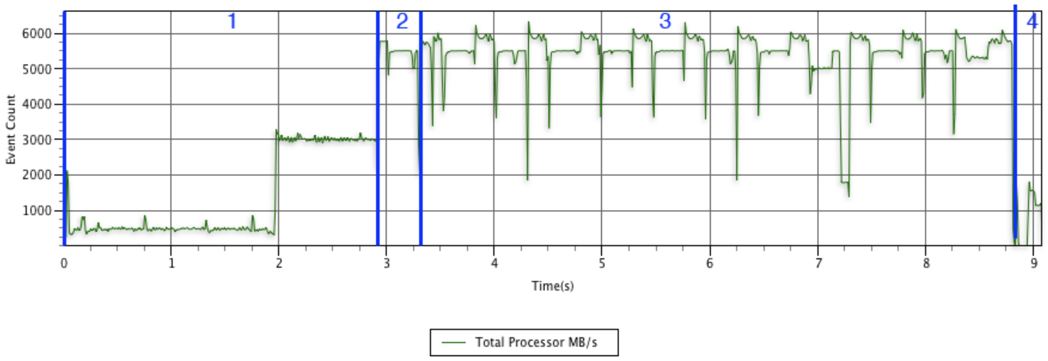
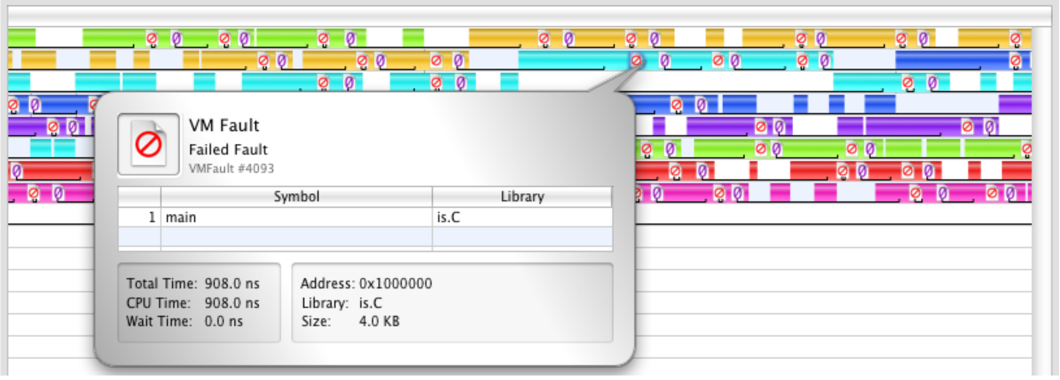


Figure 4.15: Processor Bandwidth profile of NPB IS

The derived Amdahl scaling, shown in Figure 4.17, shows us that IS has a nearly perfectly linear scaling efficiency. The Karp-Flatt metric, Table 4.5, reveals that we have a super-linear speedup for 1 to 4 cores, indicating that additional shared cache, gives us a significant performance boost. The serial fraction is slowly increasing with more cores, pointing to an increase in synchronization overhead at the barriers, but the overall efficiency of IS is still 98.8% when running on our 8-core test machine. We expect IS to perform exceptionally well with the addition of extra cores, with the assumption that the relative cache size scales at the same rate as the number of cores.

4.2.3.2 Validation

Input Validation: IS generates the keys by using a key generation algorithm that uniformly distributes the keys in memory. The ranking is determined by the index location of the key inside the list. The distribution is dependent on the memory model used. Since we use a shared memory model, the keys are stored in a continuous block of addresses.

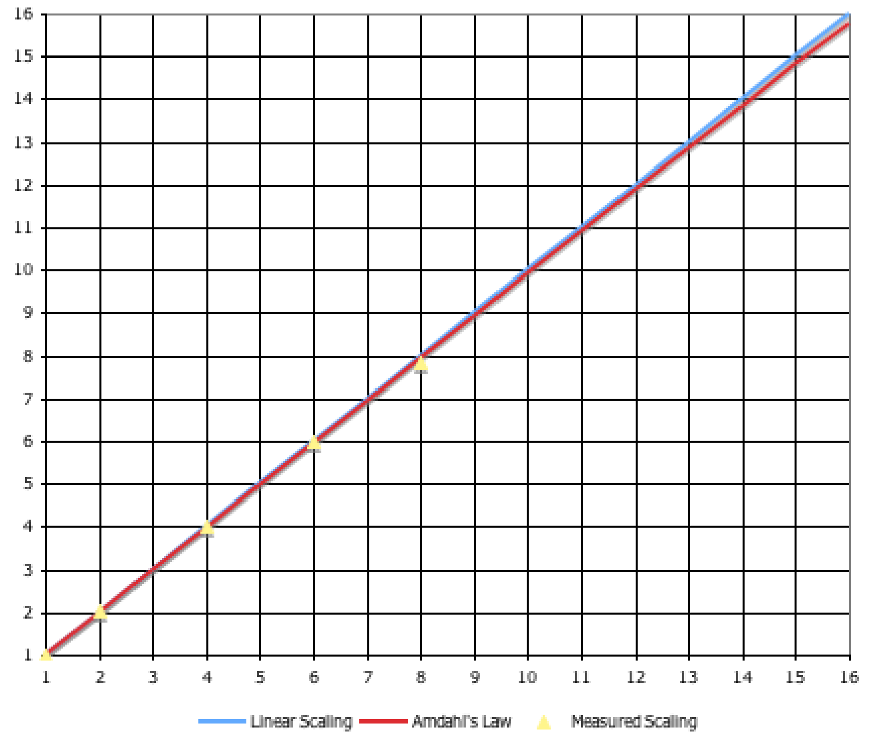

VM Fault
Failed Fault
VMFault #4093

	Symbol	Library
1	main	is.C

Total Time: 908.0 ns
CPU Time: 908.0 ns
Wait Time: 0.0 ns

Address: 0x1000000
Library: is.C
Size: 4.0 KB

Figure 4.16: System Trace profile of NPB IS



*

Figure 4.17: Scaling of IS depending on number of cores

Output validation: The IS benchmark has two internal testing systems. A partial verification is performed after each ranking. This partial verification checks if a subset of ranking is the same as the reference values known by the application. For the final and full verification, we run through the list and make sure they have been sorted correctly.

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	17.44			1.00	
2		8.65	2.02	1.01	-0.00822
4		4.33	4.02	1.01	-0.00197
6		2.92	5.98	1.00	0.00072
8		2.23	7.83	0.98	0.00308

Table 4.5: Calculating the scalability of NPB IS

4.2.4 MG: MultiGrid

Computational kernel of a 3-D scalar Poisson equation using a V-cycle multigrid method. Applicable Dwarf: 5. Parallelization is done by using OpenMP directives on the outer most loops of every step in the V-Cycle algorithm.

4.2.4.1 Scalability

MG consists of 3 main phases during its full execution: The setup, consisting of 3 steps, is where the work arrays are created. For the first step, the work arrays are zeroed and filled with +1 and -1 at ten randomly chosen points. The second step calculates the residual and evaluates the norm of the array. The third and final step in the setup is to do a warmup iteration of the V-cycle. This process takes around 13 seconds on our test machine and is not included in the scaling study. The Second phase is the actual V-Cycle benchmark, where we measure how long it takes to do 20 iterations. The V-cycle algorithm consists of 5 steps that are executed in sequence. The first step is the function *rprj3*, that restricts the residual from a fine grid to a coarse and takes 8.3% of the total computational time. The second step is the function *psinv*, that calculates an approximate solution for the coarse grid, taking up 22.9% of the total computational time. The third step is the function *iterp*, that extends the approximate solution from the coarse grid to the fine grid and takes up 8.9% of the total computational time. The fourth step is the where the residual is calculated, function *resid*, taking up 53.3% of the total computational time. The final step is again the function *psinv*, used in this case to smooth out the arrays after residual calculations. The last phase is the internal verification, where the computed residual is compared to a reference residual matching the class of the dataset computed. The synchronization overhead is 4.7% of the total computational time. During the experiment MG occupied 93.7% of all the available cpu time, with 5.7% spend it the system idle thread mach_idle and the remaining 0.6% distributed among the various background daemons.

The Processor Bandwidth profile, Figure 4.18, shows us the separate phases, The setup phases (marked as 1), the warmup phase (marked as 2), the V-Cycle benchmark (marked as 3) and the verification phase (marked as 4). It also tells us that MG is operating at the bandwidth limit of our test machine. An enlarged Processor Bandwidth profile is shown in Figure 4.19, where it shows a single iteration of the main computational phases and labels in the following order:

1. *rprj3*
2. *psinv*
3. *iterp*
4. *resid*
5. *psinv*

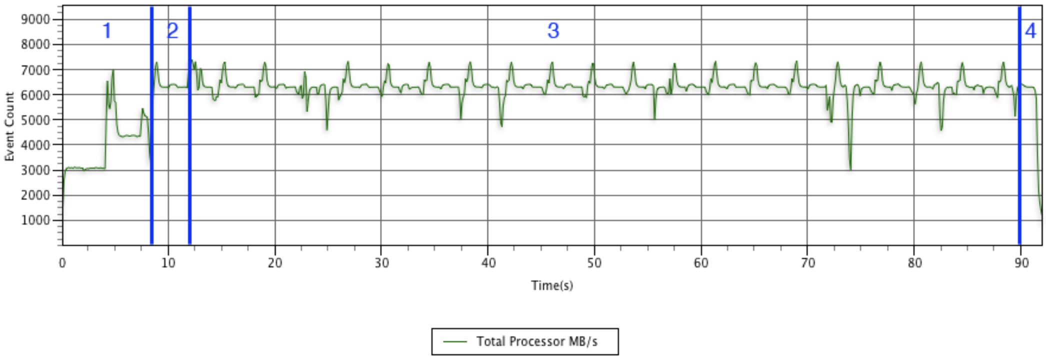


Figure 4.18: Processor Bandwidth profile of NPB MG

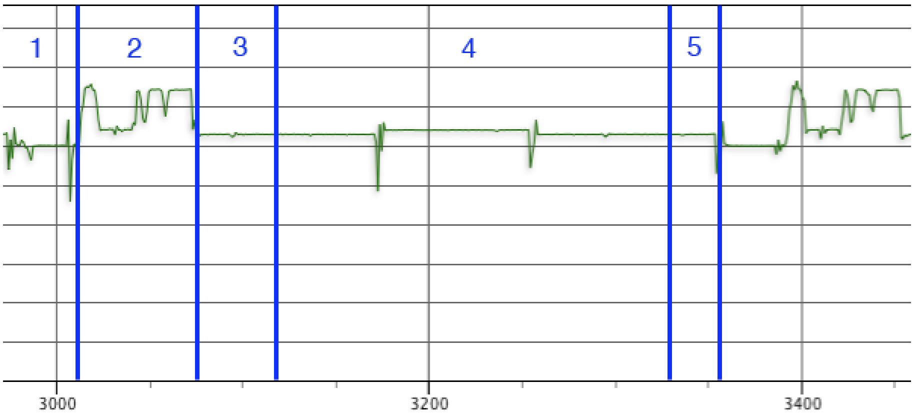


Figure 4.19: Zoomed in Processor Bandwidth profile of MG

When looking at the derived Amdahl scaling, shown in Figure 4.20, we can see that our measured datapoints follow our plotted curve at first, but differentiate further after 4 cores. Running MG with only 2 cores required 4.2GB of bandwidth on the memory bus. Increasing the number of active cores pushed these requirements even higher, eventually reaching the bandwidth limit of the system. The Karp-Flatt metric, Table 4.6, has an increasing serial fraction, pointing to an increase in synchronization overhead and bandwidth requirements as we add more cores. The overall efficiency of MG is only 56% when running our 8-core test machine.

*

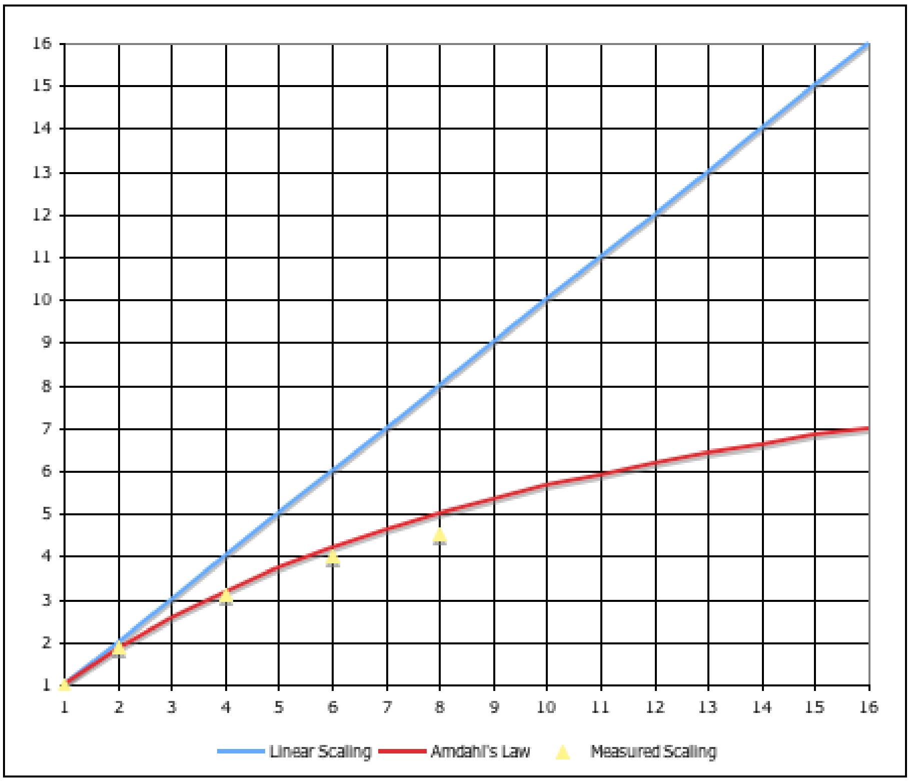


Figure 4.20: Scaling of MG depending on number of cores

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	79			1.00	
2		41	1.92	0.96	0.04292
4		25	3.14	0.78	0.09161
6		20	4.04	0.67	0.09739
8		18	4.52	0.56	0.11020

Table 4.6: Calculating the scalability of NPB MG

4.2.4.2 Validation

Input validation: The input for the MG benchmark is defined for the discrete Poisson problem $\nabla^2 u = v$ for which $v = 0$ except for twenty points predefined where $v = \pm 1$. These matrices will be used as initial input for the application.

Output validation: Depending on the class size used, internal verification is done by looking at the residuals of the V-cycle multi-grid algorithm and comparing them to the reference value of that class type. If the residuals are within the tolerance defined for that class type, the verification passes.

4.2.5 CG: Conjugate Gradient

Computational kernel that calculates the largest eigenvalue of a large sparse unstructured matrix. Applicable Dwarf: 2.

4.2.5.1 Scalability

The CG benchmark has two main stages, the setup, where the generation of the sparse matrix takes place, and the main conjugate gradient (CG) loop. The setup takes only 1.3% of the total runtime, while the CG loop takes up 94.1% of the total runtime. There are several synchronization barriers in place to make sure the phases of do not overlap. The first barrier is at the end of the setup, before the conjugate gradient loop iterations can start. There are several barriers within the CG loop, around specific computation blocks such as the sparse matrix vector multiplication and the two reduction sums. The final barrier is where the norms are calculated via reduction. These barriers account for 3.6% of the total runtime. The remaining 1% is distributed among various system functions. During the experiment CG occupied 97.0% of all the available cpu time, with 2.1% spent in the system idle thread mach_idle and the remaining 0.9% distributed among the various background daemons.

The Processor Bandwidth Profile, Figure 4.21, shows us the first phase (the setup) for the duration of 4 seconds. For the second phase, the CG loop, we see that the CG saturates the available bandwidth during the iterative procedure. The small drops in bandwidth utilization are caused by the various barriers in the code.

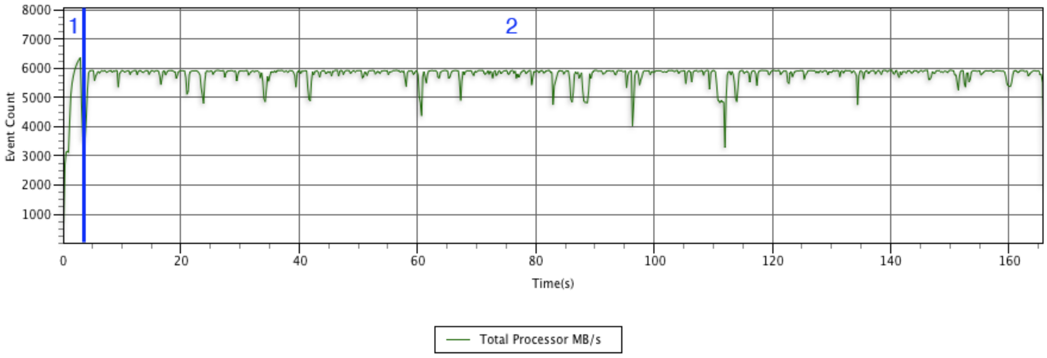


Figure 4.21: Processor Bandwidth profile of NPB CG

When looking at the derived Amdahl scaling, shown in Figure 4.22, we can see that our measured datapoints follow the optimal linear scaling up to 4 cores, after that point the overhead of synchronization increases. The Karp-Flatt metric, shown in Table 4.7, shows a super-linear speedup for 2 cores, caused by the addition of extra cache associated with the second core. With more cores, we can see the serial fraction increasing, pointing towards extra overhead caused by the communication and synchronization, as well as increased pressure on the shared caches. The overall efficiency of CG is 81% when running on our 8-core test machine. We expect CG to perform well with the addition of extra cores, assuming that the cache size scales with the number of cores.

*

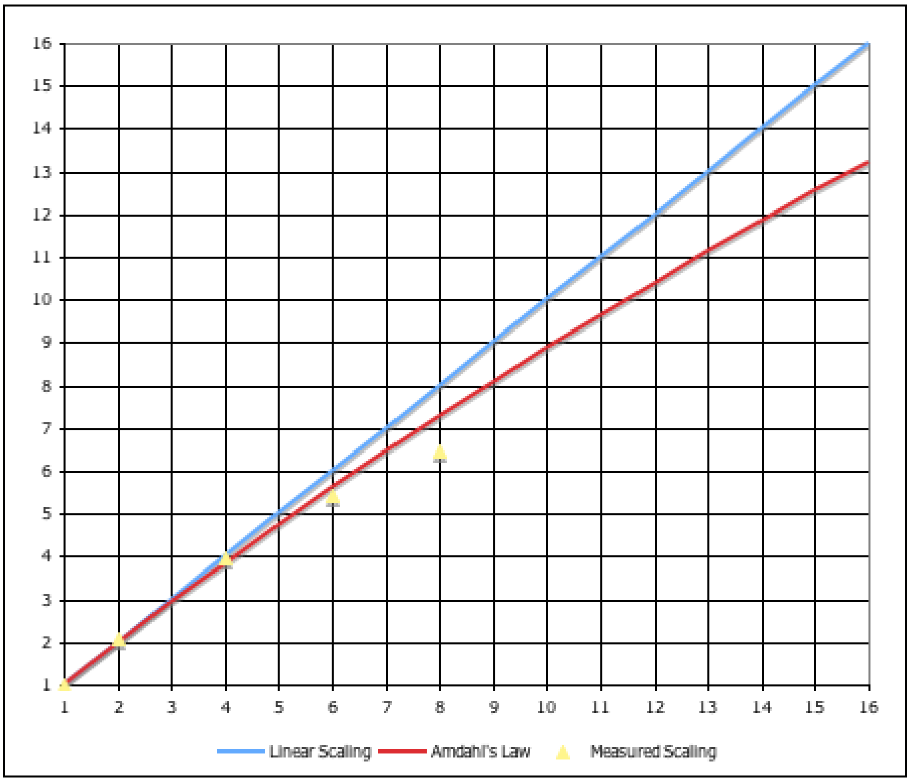


Figure 4.22: Scaling of CG depending on number of cores

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	342			1.00	
2		166	2.06	1.03	-0.03125
4		86	3.99	1.00	0.00099
6		63	5.43	0.91	0.02084
8		53	6.47	0.81	0.03388

Table 4.7: Calculating the scalability of NPB CG

4.2.5.2 Validation

Input validation: The CG benchmark will generate real numbers for the symmetric positive definite sparse matrix with a random pattern of non-zeros.

Output validation: The CG benchmark has an internal verification system that checks if the zeta (ζ) that was calculated is the same as the reference value ζ_{REF}.

4.2.6 EP: Embarrassingly Parallel

An Embarrassingly Parallel Benchmark. By generating pairs of Gaussian random deviates, this benchmark attempts to calculate peak performance of a platform. This is typical behavior for Monte Carlo/Map Reduce simulation applications. Applicable Dwarf: 7.

4.2.6.1 Scalability

The benchmark EP has only a few phases that are fully parallel. First it generates an array of pseudo random numbers using the *vranlc* Fortran call, using 9.8% of the total processing time. The second phase is the generation of Gaussian pairs based on the previously generated pseudo random numbers, taking up 90.1% of the total computational time. This phase computes the Gaussian deviates $X_k = x_j \sqrt{(-2 \log t_j)/t_j}$ and $Y_k = y_j \sqrt{(-2 \log t_j)/t_j}$ [19], with a mean of 0 and a variance of 1. A whopping 24.1% of the total computing time is required just for the logarithmic calculations. The only synchronization barriers are at the end, where EP sums the gaussian pairs in parallel. This gather operation is a fraction of the total runtime and does not account for any influence in the scalability of EP. During the experiment EP occupied 97.9% of all the available cpu time, with 1.9% spent in the system idle thread mach_idle and the remaining 0.2% distributed among the various background daemons.

When looking at the Processor Bandwidth profile, Figure 4.23, we see that there is a minimal amount of traffic on the memory bus, pointing towards all local memory accesses and no communication between the threads. This agrees with the type of work EP is doing, calculating pseudo-random numbers and forming Gaussian pairs.

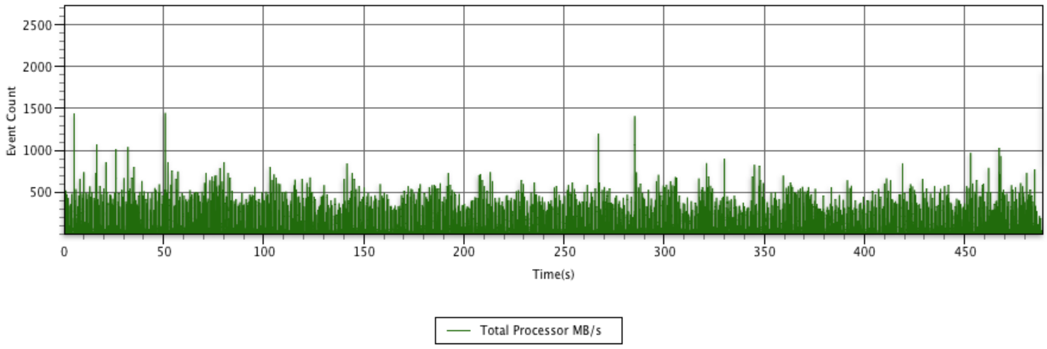


Figure 4.23: Processor Bandwidth profile of NPB EP

The derived Amdahl scaling, Figure 4.24, confirms our theory by showing perfect linear scaling for this benchmark. The Karp-Flatt metric, Table 4.3, reveals that we have a small super-linear speedup for every increase in the number of cores. This tells us that the entire EP benchmark runs out of the caches and the little communication required for the Gaussian pairing is fetched from the neighboring caches. The slow decrease in the Karp-Flatt metric does point to an decrease in efficiency concerning cache coherency when increasing the number of cores, but EP is still expected to give perfect linear scaling for high numbers of cores.

*

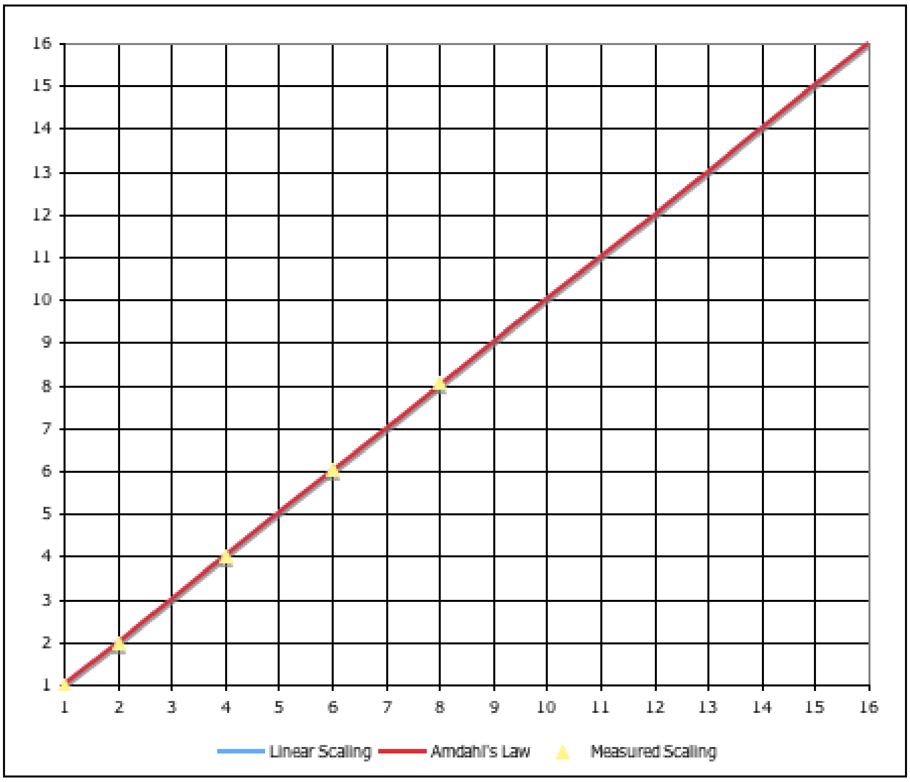


Figure 4.24: Scaling of EP depending on number of cores

	Speedup and Efficiency				Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	2540			1.00	
2		1262	2.01	1.01	-0.00631
4		629	4.04	1.01	-0.00298
6		419	6.06	1.01	-0.00195
8		314	8.08	1.01	-0.00143

Table 4.8: Calculating the scalability of NPB EP

4.2.6.2 Validation

Input validation: The application will generate floating point numbers pseudorandomly, therefore there is not an input requirement for this case.

Output validation: The EP benchmark has an internal verification system that checks if the sums of the pairs are identical with certain reference values. If these results differ, then the application will report an error.

4.2.7 UA: Unstructured Adaptive

Computational kernel that calculates solutions to unsteady partial differential equations (PDEs) using Spectral Element Method with adaptive non-conforming meshes [13, 14]. Applicable Dwarf: 3,6.

4.2.7.1 Scalability

The UA benchmark has three main phases, the setup, where the initial grid is created based on the input criteria, the main computational part where the adaptive mesh is calculated and solved using a Conjugate Gradient loop and the final verification phase. The setup and verification is not included in the time calculations since the internal timing mechanisms only time the main computational part. The main computational part is built around the grid calculations, transferring data between shared points and computation of the conjugate gradient. This process is done repeatedly while the algorithm refines or coarsens the grid up to a predefined maximum level of refinement. This computational portion takes up a total of 65.5% of the total runtime. There is extensive use of locks to transfer the data between adjacent elements in the unstructured grid. These locks do cause significant overhead and consume 33.5% of the total runtime. The remaining 1% is distributed among various system functions. During the experiment UA occupied 98.3% of all the available cpu time, with 1.6% spent in the system idle thread mach_idle and the remaining 0.1% distributed among the various background daemons.

The Processor Bandwidth profile, Figure 4.25, shows us that the UA application is pushing the memory bus to its limits even with the locks. The Setup phase is not visible in the profile since it is relatively short compared to the duration of the application. The iterative nature of this benchmark is shown in the enlarged Processor Bandwidth profile, shown in Figure 4.26. The first segment, where the convection term (gathering the grid points to the central collocation points) shows a drop in bandwidth traffic due to the locking. The second stage the refinement steps are repeated until the maximum allowed refinement is met.

When looking at the derived Amdahl scaling, shown in Figure 4.27, we can see that our measured datapoints reveal the problems caused by the locking overhead. The Karp-Flatt metric, shown in Table 4.9, shows an irregular serial fraction, caused by load-balancing issues. Another determining factor of limited scalability is that from 4 active cores on the bandwidth of our test machine is saturated. The overall efficiency of UA is 61% when running on our 8-core test machine. We expect UA to have limited scalability beyond 8-cores if the locking and memory bandwidth bottlenecks are not solved.

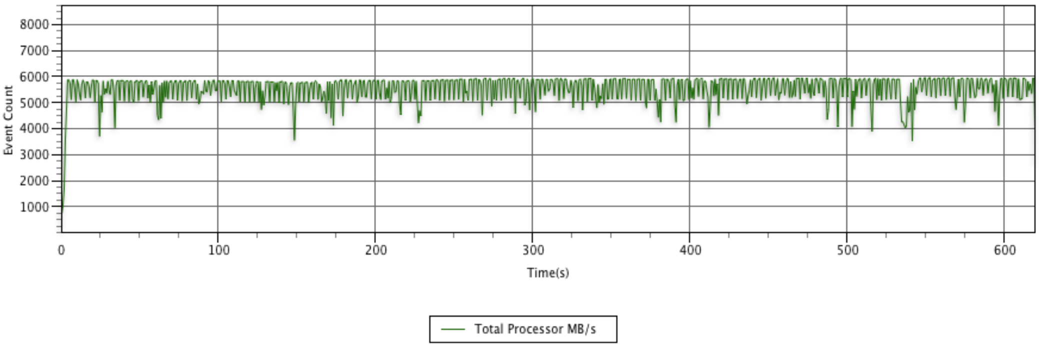


Figure 4.25: Processor Bandwidth profile of NPB UA

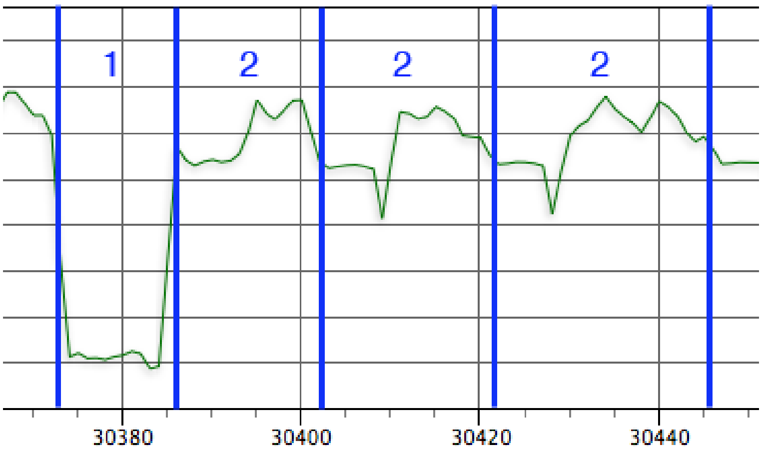


Figure 4.26: Zoomed in Processor Bandwidth profile of UA

4.2.7.2 Validation

Input validation: The application has one element $[0,1]^3$ that makes the entire domain with a predefined heat source, therefore there are no input requirements for this case.

Output validation: The UA benchmark has an internal verification system that checks the result of the integral with the verification value. If the difference between the computed integral and the verification value, divided by the verification value, exceeds the preset threshold of 10^{-8}, the application will return an error.

4.2.8 Dwarf overview for NPB

The NASA Parallel Benchmark (NPB) has a total of eight benchmarks, that fall under a total of six dwarfs as shown in Table 4.10.

When looking at the collected data we can say the following about the NPB components:

*

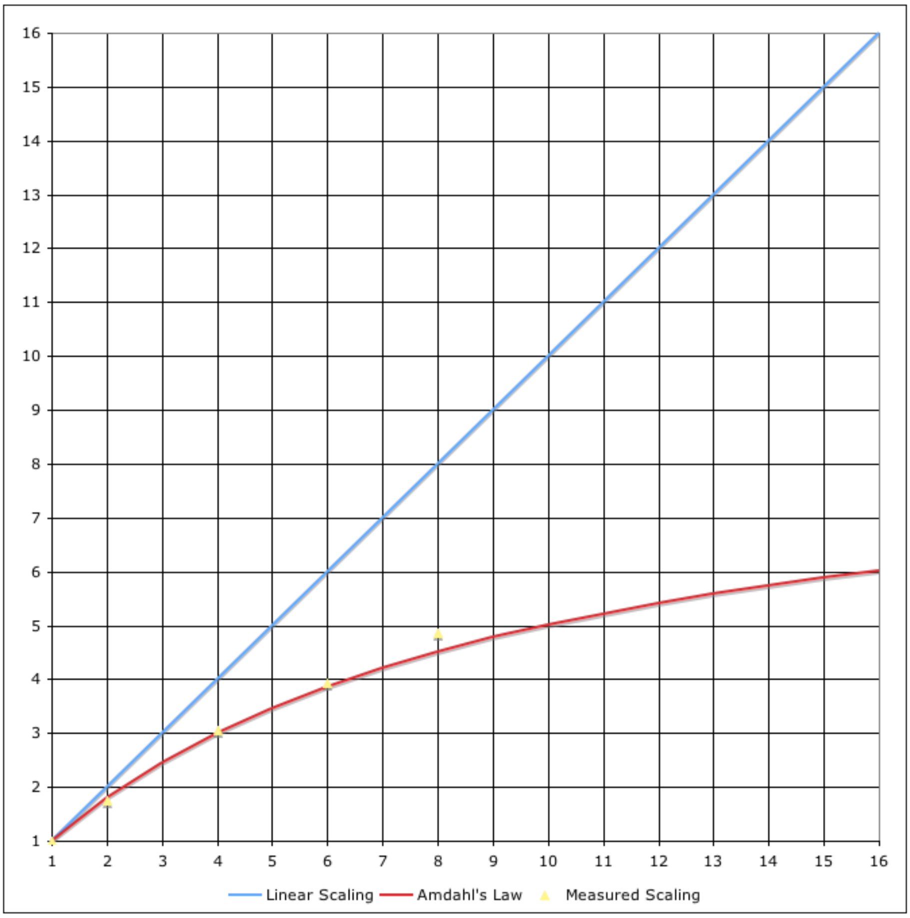


Figure 4.27: Scaling of UA depending on number of cores

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	1256			1.00	
2		718	1.75	0.87	0.14386
4		410	3.07	0.77	0.10147
6		319	3.94	0.66	0.10449
8		258	4.87	0.61	0.09205

Table 4.9: Calculating the scalability of NPB UA

Dwarf:	1	2	3	4	5	6	7	8	9	10	11	12	13
NPB	BT LU	CG	FT		MG SP	UA	EP						

Table 4.10: Dwarf distribution in NPB

- BT has better scaling properties than LU, running LU is not necessary in this case.
- MG has better scaling properties than SP, running SP is not necessary in this case

We can safely discard LU and SP from the workload if we choose to do so. LU is also covered primarily by the linpack component since both are in dense linear algebra applications.

4.3 WRF

The Weather Research and Forecasting (WRF) Model is a mesoscale numerical weather prediction system. Mesoscale weather prediction systems work on areas ranging from a 5 km to several hundred km. In our workload, we use a model of the continental US (CONUS) as our input. WRF uses a 2nd and 3rd order Runge-Kutta method for time integration and the approximation of the differential equations required for the computation of the weather models. The work is distributed by either generating multiple threads using OpenMP or creation of processes with the MPI framework. The WRF system partitions the input data into a structured grid that is distributed to the nodes. The nodes then perform calculations on the smaller datasets using spectral algorithms and the results are shared between neighboring nodes.

4.3.1 Scalability

The implementation of WRF we chose to use for our workload uses OpenMP for the parallelization. OpenMP works in the same memory space so sharing of the dataset is possible, unlike MPI where every process has its own memory space and duplicates the dataset for every process created.

The Processor Bandwidth profile, Figure 4.18, shows us that WRF is capable of saturating the available bandwidth of our test system, where each drop in bandwidth, at time index 90/175/260/340/425/510, is a write of the processed data to disk. Since WRF walks through the dataset on a per time step base, there are no distinct phases to be detected from the Processor Bandwidth profile. The main computational function is the *solve_em* that controls the sub-functions that compute every time-step. The main components of the *solve_em* function are the main Runge-Kutta loop, with physics calculations and environment variables (e.g. temperature/moist/pressure) are updated within that loop, and time-split physics where the calculation of the environment variables are post processed my the micro-physics driver. The *solve_em* and its sub-functions take up 88.9% of the total computational time. Other functions take up 6.4% of the computational time. The synchronization overhead is 4.3% of the total computation time. The remaining 0.4% of the time is spend doing system calls.

When looking at the derived Amdahl scaling, shown in Figure 4.29, it becomes clear that the scaling efficiency drops off when adding more cores. The cause of this limited scalability is the requirement of cache coherence between the cores. Cache coherence makes sure the integrity of the local caches is guaranteed. By invalidating its own cache when the another shared memory block is being written by another cache (this process is called cache snooping) it can make sure that the local value is always in sync with the rest of the system. This reasoning is reinforced by the Karp-Flatt metric shown in Table 4.11, where the serial fraction is large and has a non-linear stepping when increasing the numbers of cores. The efficiency of WRF when running 8-cores drops down to 53%. We don't expect the efficiency to improve when we add more cores to the system when if the memory bandwidth available stays the same. During the experiment WRF occupied 91.3% of all the available cpu time, with 8.4% spend it the system idle thread mach_idle and the remaining 0.3% distributed among the various background daemons.

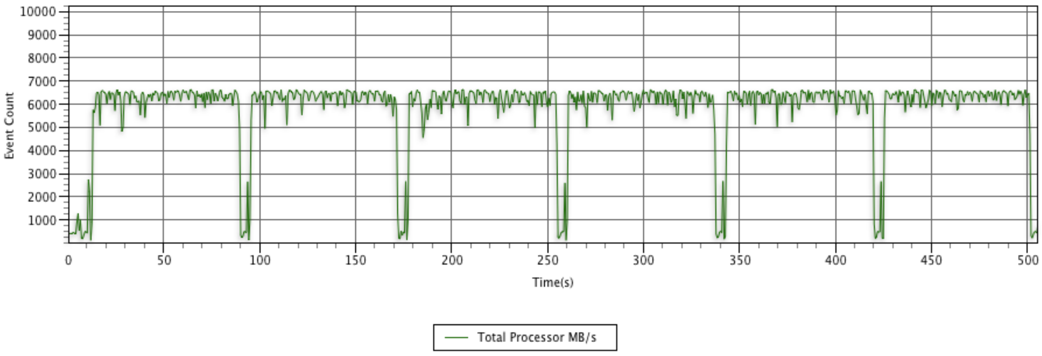


Figure 4.28: Processor Bandwidth profile of WRF

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	698			1.00	
2		401	1.74	0.87	0.14790
4		271	2.58	0.64	0.18369
8		166	4.21	0.53	0.12834

Table 4.11: Calculating the scalability of WRF

4.3.2 Validation

4.3.2.1 Input validation

The weather model data used in this test was provided by Prof. R. Fovell from the Department of Atmospheric and Oceanic Sciences, University of California, Los Angeles [15]. It is a simulation for a Continental U.S. (CONUS) weather model, with 15km data point resolution.

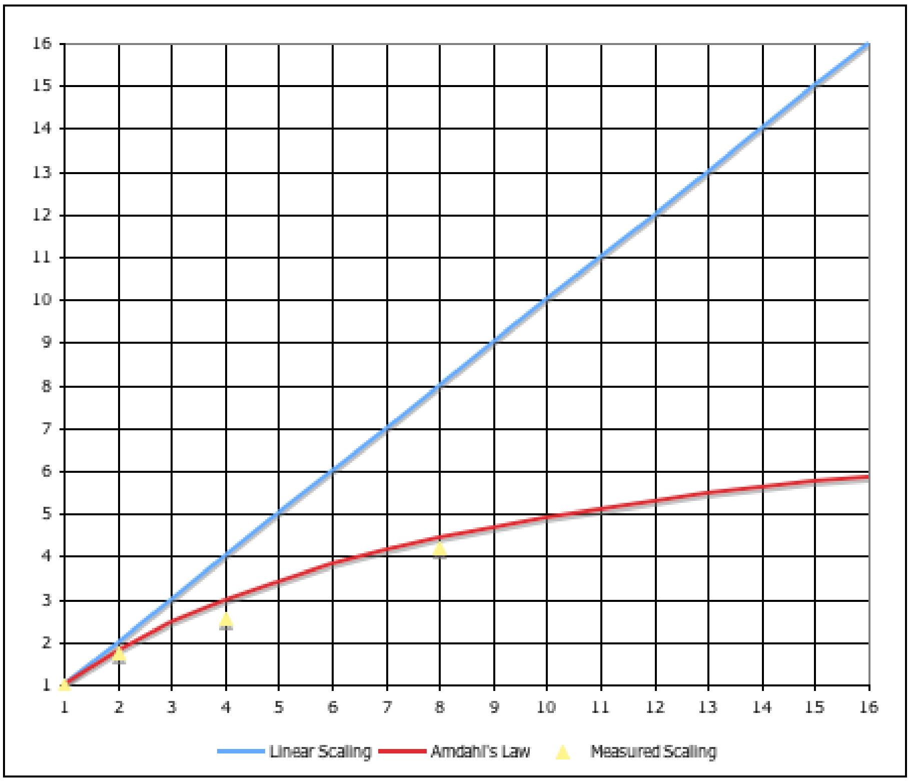


Figure 4.29: Scaling of WRF depending on number of cores

4.3.2.2 Output validation

The output produced by WRF can be verified by comparing it with a reference output. The application diffwrf provides an overview of the variation between the reference output and the computed output. Since weather modeling is not an exact science, the results are expected to vary between the runs, but the WRF user community will only accept results that are the same to within the 3rd correct decimal [15].

Media Benchmarks

5

Another domain we chose to look at for finding suitable applications was content creation. Content creation has ever increasing computational requirements, ranging from creating high quality digital 3D renders that are almost impossible to differentiate between an actual photo or converting raw High Definition video footage to high compression video playback formats, all demand a lot from modern hardware. Various applications in this domain are multi-core aware and tend to be highly parallel in their computational tasks.

5.1 Yaf(a)Ray Raytracer

Yaf(a)Ray is an open-source raytracer that is capable of rendering complex models without the scaling limitations of the original YafRay raytracer. YafRay is part of the popular Blender3D [2] open-source rendering/modeling package. Yaf(a)Ray renders with multiple threads, with each thread rendering a 64x64 pixel block. Because Yaf(a)Ray is a raytracer, it uses light particles to calculate the visual representation of the models. As every light particle travels through an imaginary path from the point of the camera it will bounce off objects, creating soft shadows and reflections. This process is computationally expensive, since a complex scene can have millions of particles bouncing around. The first modern raytracing algorithm was designed by Steven Rubin and Turner Whitted in 1979 at Bell Laboratories [29].

5.1.1 Scalability

Yaf(a)ray is designed to be scalable, with each thread capable of rendering raytraced blocks independently of the other threads. Yaf(a)ray uses pthreads, discussed in section 3.2.2, to partition the work among the available cores. Because the threads are internally independent, there is no communication between the threads and no synchronization barriers, resulting in excellent scalability when we introduce more processing units. When looking at the derived Amdahl scaling, shown in Figure 5.1, we can see that the scaling of Yaf(a)ray is close to linear and the measured datapoints follow the plotted Amdahl's law curve. There is no optimized serial version of the Yaf(a)ray code, so we run Yaf(a)ray with 1 processor to determine the Serial runtime T_s resulting in $T_{(1)}$. Table 5.1 shows us that the efficiency from 1 core to 8 cores is close to linear. However, we can see that there is a minute drop in efficiency going from 4 to 8 cores. This drop is explained by looking at the Karp-Flatt metric [20] where a steady increase in the serial fraction $Serial_{frac}$ points towards increasing overhead. This extra overhead is caused by the creation and destruction of extra threads for each block that is rendered. It is interesting to note that going from 1 core to 2 cores, we get a super-linear speedup primarily due to the extra cache that becomes available when adding a second core.

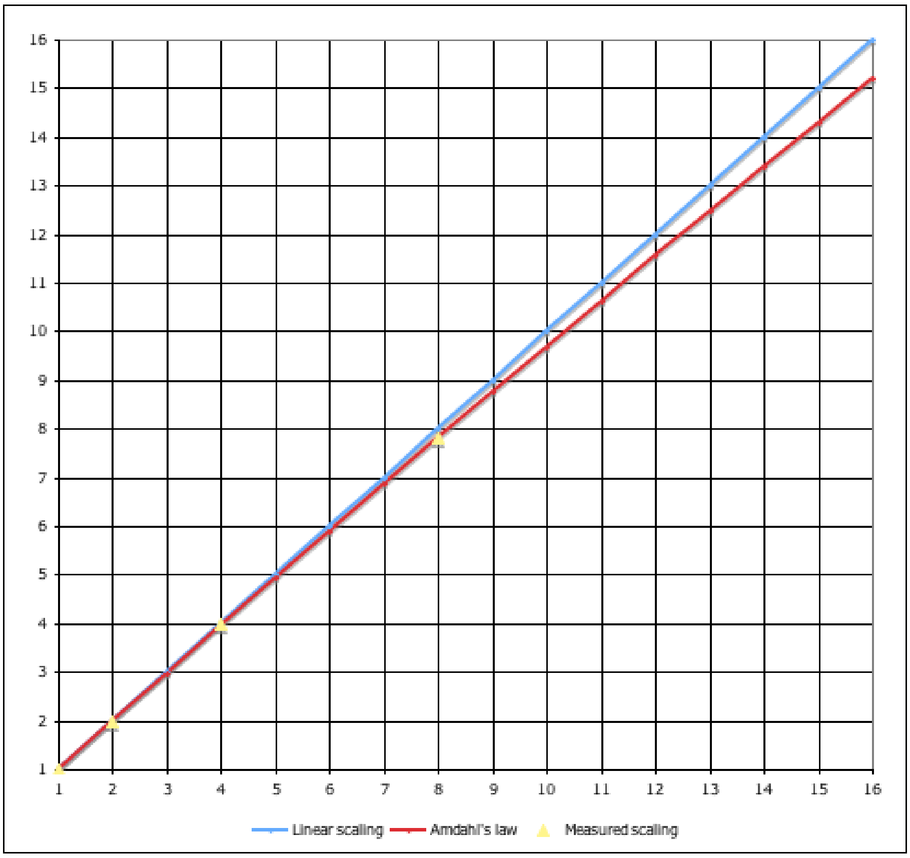


Figure 5.1: Scaling of Yaf(a)ray depending on number of cores

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	1732			1.00	
2		865	2.00	1.00	-0.00144
4		434	3.99	1.00	0.00068
8		222	7.81	0.98	0.00354

Table 5.1: Calculating the scalability of Yaf(a)ray

By analyzing the Time Profile of Yaf(a)ray, we confirm that there is no interprocess communication and no synchronization between the threads. There are 4 primary methods that are called by Yaf(a)ray during raytracing: $triangle\_t :: intersect$, $triKdTree\_t :: intersect$, $triKdTree\_t :: intersectTS$ and $pathIntegrator\_t :: integrate$. The first three methods are primarily used to compute the crosspoints of the objects and the light particles for rendering. $PathIntegrator\_t :: integrate$ calculates the path that the light particles take when they bounce off of the objects to light other portions of the scene. These 4 methods take 77% of the execution time.

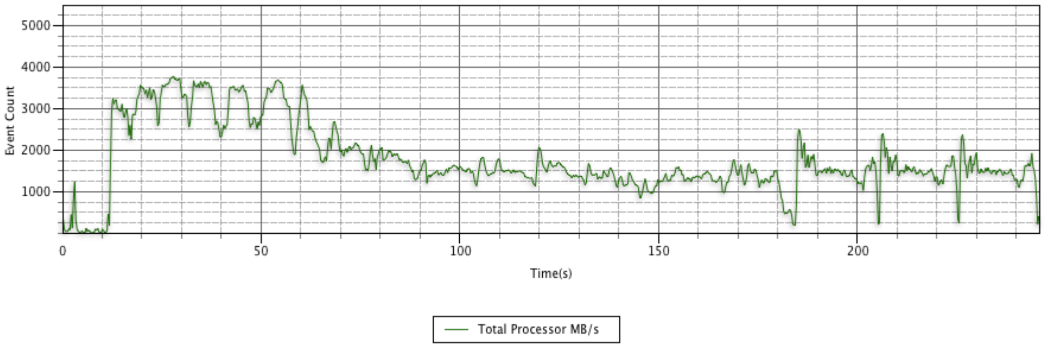


Figure 5.2: Processor Bandwidth profile of Yaf(a)ray

The Processor Bandwidth profile showed that there was considerable traffic on the Front Side Bus, but not enough to fully saturate the bus continuously. Over the duration of the application, fewer image blocks need to be updated, and therefore the amount of memory traffic also decreases over time. Using the processor bandwidth profile, we can determine 5 separate stages of the application based on the bandwidth required and the type of functions being executed at that time.

Figure 5.2 identifies the following computational stages in the Yaf(a)ray application:

1. *Setup:* During the first 12 seconds, the application only runs with the master thread, where it will do initial setup of the environment and load the plugins needed for the render.

2. *(12 seconds to 186 seconds):* The master thread spawns multiple worker threads which will calculate separate image blocks. The calculation is primarily done using the 4 main functions mentioned before, using about 79% of the time. The remaining time is is spread across support functions. There is no inter-thread synchronization.

3. *(186 seconds to 208 seconds):* Similarly to the second stage, this stage spends 77% of its execution time in the 4 main functions. The duration in this case is shorter because we do not have to calculate the entire block from scratch. This stage refines the image blocks and updates the photon maps so that the lighting and reflections can be calculated more precisely during the next pass.

4. *(from 208 seconds to 228 seconds):* This pass is identical to the previous pass, spending 77% of its time in the 4 main functions as it refines the final image and updates the photon maps.

5. *(from 228 seconds to 248 seconds):* This final stage, where the image is updated for the last time, is identical to the previous 2 stages except that it produces the final render.

We can see that the initial setup generates almost no memory bandwidth. The first render runs for 68% of the total runtime and generates the most memory traffic due to the generation of the base image, where each image block needs to be rendered. The remaining “refining” render passes require a total of 26% of the total render time. The resulting image is a render with soft shadows and proper light refraction of transparent objects. The final render of the test image is shown in Figure 5.3.

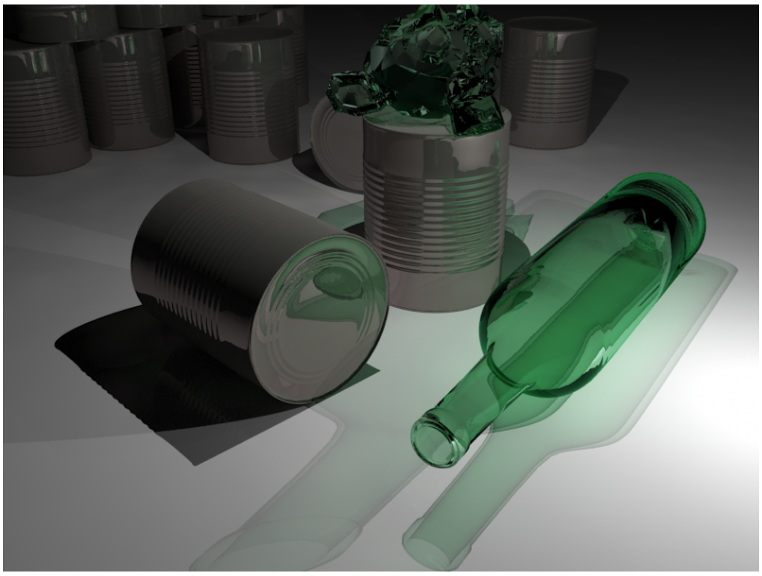


Figure 5.3: Final raytraced image produced by Yaf(a)ray

5.1.2 Validation

5.1.2.1 Input validation

As an input we use an xml file containing a model description provided by a third party [35] that is processed by the raytrace renderer. The xml file is pre-processed every time to correctly set the thread count to the number of available cores in the system.

5.1.2.2 Output validation

We collect every render for each iteration we run and calculate the Peak Signal to Noise Ratio (PSNR) on the difference between the output file and the reference render image. The higher the PSNR number, the closer the images are. Any variations in the output image and the reference render will be spotted by the PSNR calculations, and when the result falls below the PSNR value of 40 dB, artifacts will probably be visible to the human eye [26]. If this occurs, the result is declared invalid and discarded.

5.1.3 Types of dwarfs used

Yaf(a)ray contains a variety of algorithms that match our list of dwarfs.

1. Graph traversal: Graph traversal is used to traverse the kd-tree data structure.

2. MapReduce: Monte Carlo is used for sampling in the light particle path-tracing algorithm.

5.2 x264 encoder

The x264 encoder [3] is a open source library that implements the H.264/MPEG-4 part 10 (AVC) codec. H.264 is the current generation block-oriented codec with support for the following features:

- Motion estimation between frames
  This computes the change of location of objects within a frame compared to another frame and calculates the represents the difference in vector format.
- Intra frame coding (16x16, 8x8, and 4x4 macro blocks with predictions)
  Also called I-frames, these are frames that have no reference to any other frame except themselves and are used as a base reference point for P- and B-frames. These base frames are commonly called key-frames because they provide the key information to recreate the original images.
- Predicted frame coding (16x16, 8x8 and 4x4 macro blocks)
  Also called P-frames, are frames that contain references (motion vectors and macro blocks) to the previous I-frames. P-frames require less space then I-frames because part of the data is referenced from the previous reference frames.
- Bi-directional frame coding (16x16 and 8x8 macro blocks, including skip/direct)
  Also called B-frames, are frames that contain references (motion vectors and macro blocks) to other encoded frames either before or after the current frame location in the movie stream. B-frames require less space than either I- or P-frames.
- Context-adaptive variable-length coding (CAVLC)
  Is a lossless data compression scheme used in the H.264 codec.
- Context-adaptive binary arithmetic coding (CABAC)
  Is a lossless data compression scheme used in the H.264 codec with a higher compression ratio than CAVLC, but requires more processing to decode a frame.
- Adaptive B-frame placement
  Allows variation in the number of B-frames that can follow other B-frames. This improves encoding efficiency.

5.2.1 x264 encoder

The x264 encoder processes individual frames in parallel. When encoding a source movie, each thread processes one frame from the beginning to the end, while calculating the data (macro blocks and motion vectors) for the P-, B- and I-frames required to encode its frame. Because the contents of some frames are recreated using the encoded data from other frames, some dependancies between frames will occur, as we illustrate in Figure 5.4. I-frames, such as N and N_{+8}, are not dependant upon any other frame. P-frames are dependant on the previous I-frame, and B-frames are dependant upon previous and subsequent I- and B-frames. This has the effect of creating dependancy groups of each I-frame and all P- and B-frames after it, like N to N_{+7} and N_{+8} to N to N_{+15} in Figure 5.4.

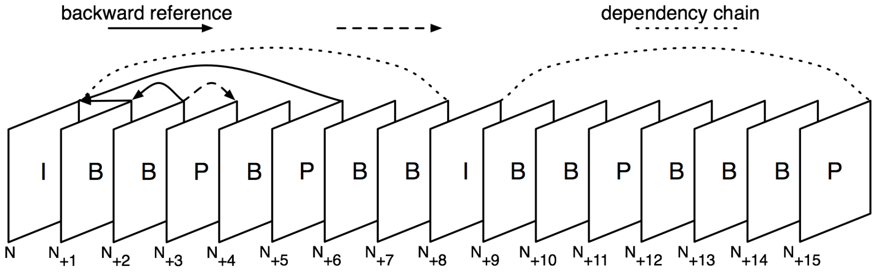


Figure 5.4: Camera and display order

To encode the B-frames with their forward and backward references, x264 reorders the frames and their corresponding threads so that the P-frames are encoded before the B-frames that precedes them, as shown in Figure 5.5. Because each thread works on one frame, encoding the frames using this method does cause some synchronization to occur due to the fact that the newer threads can only encode up to the point, on a macroblock row by macroblock row basis, where the previous thread has completed encoding. For example, if thread N is 75% done, then the following threads N_{+1} to N_{+7} (assuming no keyframes) can only encode their frame up to this point. At the beginning of each row in the frame, the thread checks the progress of all its threads it depends upon. If one of the preceding threads has not yet completed that macroblock row, the thread sleeps and waits to be woken up by the thread it is dependant upon. Figure 5.6 show the row dependency for frames N to N_{+2}.

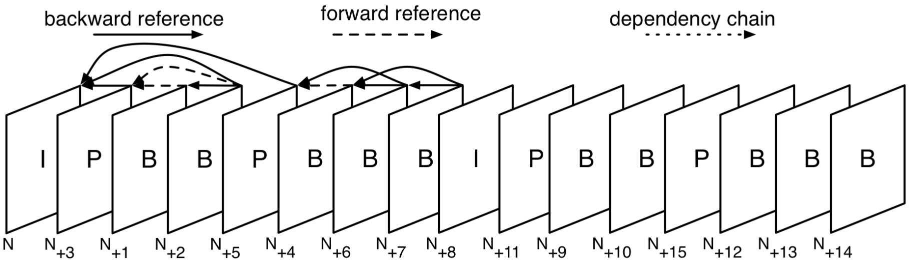


Figure 5.5: Reordering the frames for encoding

When thread N completes its encoding, it terminates and N_{+1} becomes the oldest thread. A new thread is then created for the next frame in the sequence. A sliding window of frames, are "active" on the system at once and can be processed in parallel. Because one new thread is created to replace every one that completes, the "window" of active frames always stays the same size for the duration of the encoding. Choosing the size of this window is important, because a large window can expose more parallelism,

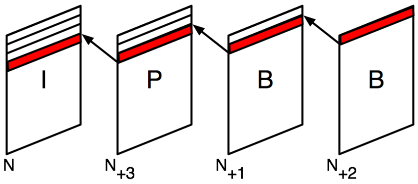


Figure 5.6: Row dependency for frames N to N_{+2}.

but requires more memory to hold the key frame data structures.

In our workload the x264 encoder processes the input 1080p HD movie on our baseline system in 200 seconds; with 10 iterations the total runtime will be roughly 33 minutes.

5.2.2 Scalability

The x264 application uses pthreads, discussed in section 3.2.2, to coordinate multithreading. As we explained in the previous section, a thread is created and destroyed for each frame encoded, and each thread stays alive until it has completed and is the oldest thread in the reference dependency chain. Thread creation and destruction has minimal overhead because of the low frame rate, under 25 frames per second for an encode of an HD source. Due to the synchronization overhead, we can see drops in performance when many dependent threads are waiting for older threads to complete encoding, primarily cause of the delays are caused by fast upward motion [8]. One way to minimize the performance hit of sleeping threads is to increase the number of spawned threads to an amount greater than then number of available cores, a process called over-subscribing, because this creates a larger "window" of available threads and hence increases the amount of exploitable parallelism. When setting the threading model to automatic, the number of threads is set to 1.5 times the number of the available cores. We discovered that for an 8-core machine setting the number of threads to 2 times the number of cores performs better and there is less system idle time, so we set our thread count to 16 for our experiments.

The application x264 has 4 distinct stages [22] when processing a frame:

1. *Frame type decision:* The main thread decides what kind (I/P/B) frame the next thread will be encoding. This is done by a fast heuristic motion estimation pass on the next two frames. It compares different encoding strategies for the next two frames. First, it determines if the second frame contains more than 50% of the previous I-frame, which prevents the use of B-frames. The second step is to see what order is more efficient, PP-frames or BP-frames. In the case where the P-frame following the B-frame is more efficient, another B-frame will be added before the last P-frame until a certain intra block threshold is reached in the P-frame. When this process is complete, worker threads will be created to do the next 3 stages

2. *Macroblock analyze:* Macroblock analyze is the second stage in the encoding process. The worker thread will do motion estimation (in our example we use transformed exhaustive search algorithm or TESA). This is the complex and time consuming step. Next come the intra-predictions that allow different macroblock sizes to be encoded, improving the efficiency. When the macroblock type and motion vectors are selected, we proceed to the encoding stage.

3. *Macroblock encode:* The Macroblock encoding we selected is context-based adaptive binary arithmetic coding (CABAC) with Trellis quantization. The trellis algorithm searches for the lowest cost for encoding a macroblock, by looking at several encoding paths and choosing the one with the shortest path/cost.

4. *Macroblock write:* The final stage is collecting the encoded frames and converting them to a bitstream that can be written to file.

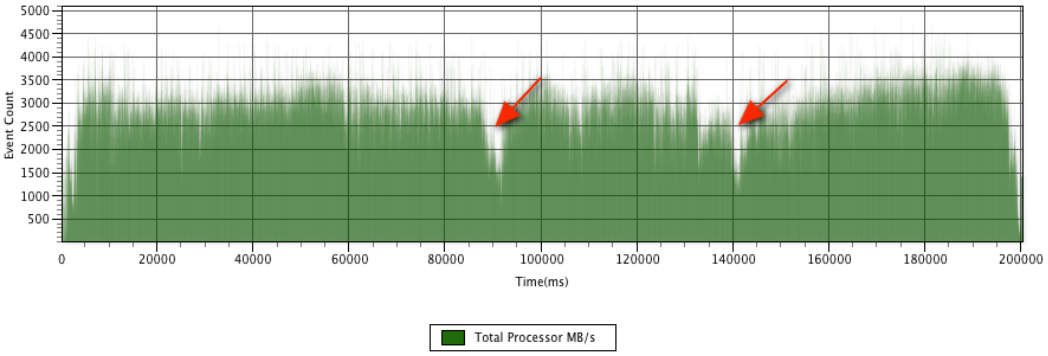


Figure 5.7: Processor Bandwidth profile of x264 encoder

The Processor Bandwidth profile, Figure 5.7, shows us that there are fluctuations in the memory bandwidth, with arrows pointing to a drop in memory traffic in some points. These drops are caused by fast vertical motion in parts of the stream. A fast motion vector requires more calculations for the frame being encoded. Due to the way the dependency chain is built, the following frames have to wait until the first frame has encoded a row before they can encode the same row or reference the previous frame. This in turn causes the waiting thread(s) to go to sleep, and in the worse case all the threads are waiting on the oldest thread, assuming there is no keyframe to break the dependency chain. Other fluctuations in bandwidth utilization are areas in the stream where the scene has minimal changes and B-frames can be used to reference previous or future frames without encoding much data, which results in less memory traffic as well.

Using the Processor Bandwidth profile, we can determine where these drops in performance occur and then correlate them with events in the movie stream. The sample movie we selected consists of a side scrolling scene from the Creative Commons movie Big Buck Bunny [1]. At the time index of 90 seconds, we can see a drop in bandwidth caused by the thread synchronization required for the large motion vectors starting at frame 290 of our sample movie. The initial thread will spend more time calculating the frame and will cause the following threads to sleep while they are waiting on the initial thread to finish encoding a sufficient number of macroblock rows. Another point in the chart is around the time index of 140 seconds, where a similar situation occurs, this time caused by the large motion vectors starting at frame 370.

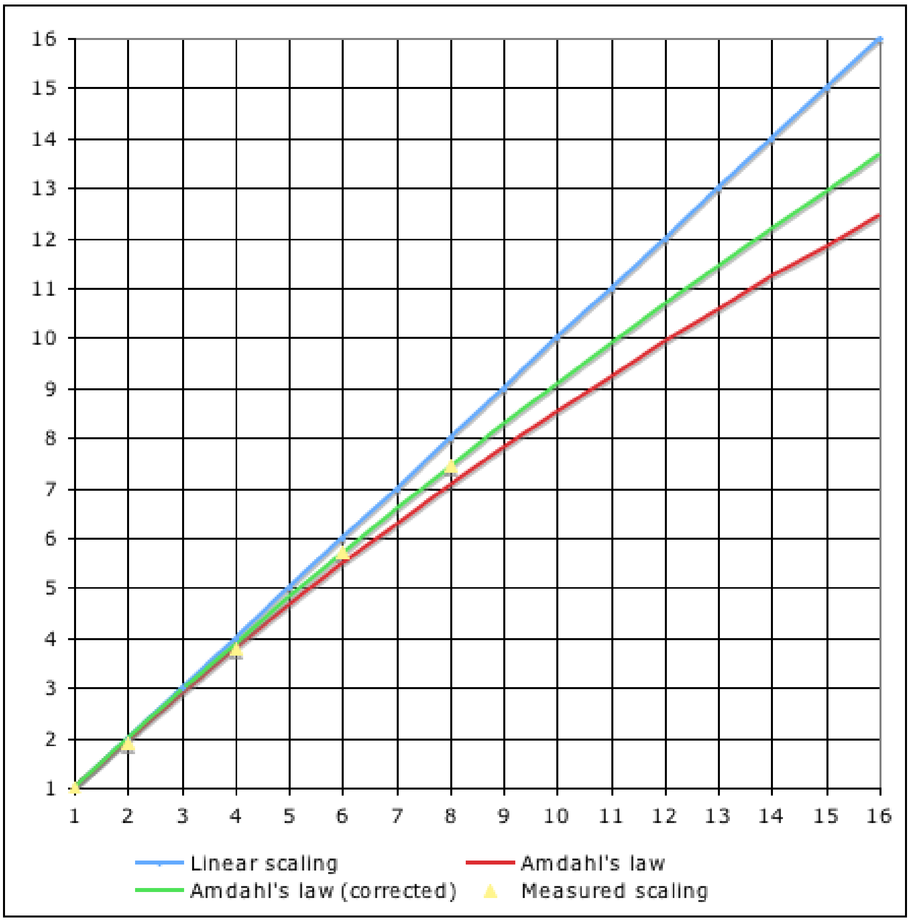
When looking at the derived Amdahl scaling, shown in Figure 5.8, we can see that the measurements begin close to the linear line but start to drop off when we add more cores. When running the x264 encoder with a single thread, it does not do any synchronization blocking and does not need over-subscription. When we do use multiple cores, the synchronization and over-subscribing functions are enabled. As shown by the Karp-Flatt metric in Table 5.2, the serial-fraction for 2 cores is 4%, a high number compared to the other values. These irregularities in the serial fraction point towards load-balancing problems, something to expect when parallelizing by over-subscribing the cores. The more cores we enable on our test system, the higher the number of "extra" inflight threads will be. Since these "extra" threads give the scheduler more choices when trying to allocate waiting threads to an idle core, they are an effective means of addressing load imbalance. When we correct the Amdahl scaling graph, by removing the obvious irregularity of 4%, we can see that the x264 encoder is expected to scale very well with a core count greater than 8. We can also see that the average efficiency is around 95% per core added, which is considerable for this type of application.

Speedup and Efficiency					Karp-Flatt metric
Cores	$T_{(1)}$	$T_{(p)}$	$S = \frac{T_{(1)}}{T_{(p)}}$	$E = \frac{S}{p}$	$Serial_{frac}$
1	1564			1.00	
2		813	1.92	0.96	0.04035
4		410	3.81	0.95	0.01623
6		273	5.73	0.96	0.00937
8		209	7.47	0.93	0.01010

Table 5.2: Calculating the scalability of the x264 codec

By analyzing the Time Profile, we can see that the x264 encoder spends its time primarily (68%) calculating the motion estimation in the *macroblock_analyze* function. The remaining time is spent on encoding the frames (24%) in the *macroblock_encode* function and creating the bitstream (4%) in the *macroblock_write* and other various functions. Even with over-subscription of the cores, 6.2% of the time is still spent in *mach_idle*, the system idle thread. This is primarily caused by the cost of creation and destruction of the threads, which is an expensive operation in OS X.

Due to the way x264 divides its work, there are no specific changes in the computational stages over time and the statistical average collected by Time Profile over the duration of the entire application gives a good overview of what happens for each frame.



*

Figure 5.8: Scaling of x264 encoder depending on number of cores

Figure 5.9: Frame from Big Buck Bunny

5.2.3 Validation

5.2.3.1 Input validation

For the input reference we use an uncompressed Y4M HDTV (1920x1080) movie clip of 512 frames from the Creative Commons project Big Buck Bunny [1]. We encode this with a complex encoding profile that will result in a highly compressed file with near lossless video quality. For more information about the encoding profile see Appendix B.3

5.2.3.2 Output validation

The x264 encoding has internal mechanisms to verify the output quality after encoding. It uses the peak signal-to-noise ratio (PSNR) and structural similarity index (SSIM) to verify the encoded image against the source frame after encoding, to see if the degradation is within acceptable limits. These two values are tracked during the entire encoding process and averaged at the end. There are small non-determinisms when encoding with multiple threads, but these are not significant enough to alter the values of PSNR and SSIM to an serious extent. Like in the Yaf(a)ray section, we will accept a PSNR [26] with a value of 40 dB or higher, since the human eye cannot determine a difference between the source and result frame below that threshold. For SSIM [36], a value of 0.90 or higher is acceptable for near lossless encoding algorithms.

5.2.4 Types of dwarfs used

The x264 encoded contains a variety of algorithms that match our list of dwarfs.

1. Finite State Machines: The trellis and CABAC residual encoding components of the x264 encoder use transition tables to calculate their target values.

2. Graph traversal: Graph traversal is used for trellis quantization to optimize the cost of each block encoded [22].

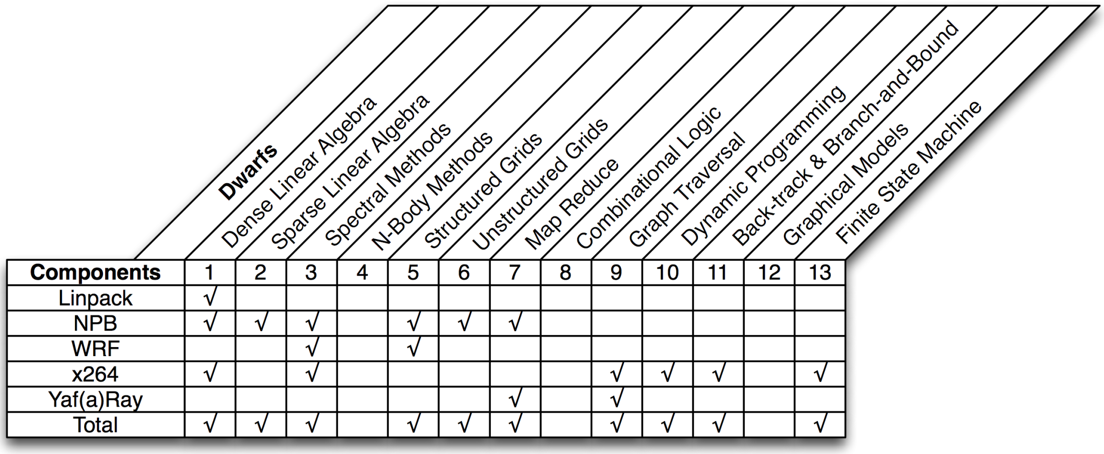
3. Branch-and-Bound: The TESA motion search algorithm uses branch-and-bound for the best encoding value.

4. Dynamic programming: Viterbi is used to choose the absolute optimal distribution of B-frames for the adaptive B-frame placement.

5.3 Selection and characterization of workload

Figure 5.10 shows the Dwarf components of the profiled applications. Because there are overlapping dwarfs, we can discard certain applications from the final workload. Selection will be based on the profile information collected in the previous stages regarding the requirements of scaling and reproducibility.

As mentioned before in section 4.2.8, we can safely discard the LU and SP benchmarks. WRF also was a possible candidate since it has considerable overlap with NPB



Dwarfs	Dense Linear Algebra	Sparse Linear Algebra	Spectral Methods	N-Body Methods	Structured Grids	Unstructured Grids	Map Reduce	Combinational Logic	Graph Traversal	Dynamic Programming	Back-track & Branch-and-Bound	Graphical Models	Finite State Machine
Components	1	2	3	4	5	6	7	8	9	10	11	12	13
Linpack	√												
NPB	√	√	√		√	√	√						
WRF			√		√								
x264	√		√						√	√	√		√
Yaf(a)Ray							√		√				
Total	√	√	√		√	√	√		√	√	√		√

Figure 5.10: Total number of Dwarfs in workload

UA and MG but we choose not do discard this application because it performs the structured gird and spectral methods within the same application, therefore it is an interesting test case for mixed algorithmic patterns.

We are still missing three essential dwarfs: N-Body Methods, Combinational Logic and Graphical Models. For future work these dwarfs are ideal candidates to be added to the workload.

Case study

6

In this chapter we provide a case study on how to compare multiple hardware configurations and we take a look at how to interpret these results. For our case study we run the workload on different hardware configurations and compare the results to see what kind of scaling occurs. We will take a 2007 4-core Mac Pro as baseline and compare it to two 2007/2008 8-core Mac Pro machines running at the same clock frequency. The major differences between the 2007 and 2008 models are the increase in cache size (from 4MB per socket to 12MB per processor), a faster memory interconnect (from 1.3GHz to 1.6GHz) and the memory speed (667MHz to 800MHz).

6.1 Performing the workload runs

We ran the default workload configuration by calling the workload using the following command: *./leviathan -l modelname -a* where modelname was the configuration of the machine we were testing. This resulted into 3 complete runs of the workload each with a different configuration. The total runtime of the entire workload is shown in Table 6.1, where we can see that there is indeed a significant decrease in the total time required to run the workload when doubling the number of cores. We can also see that the 2008 model performs its tasks faster, presumably because of the more modern hardware.

Model	Total Runtime
2007 4-core	22h20m
2007 8-core	12h51m
2008 8-core	9h23m

Table 6.1: Reported runtimes for different processor models

In the next section we will take a look at the collected data and see how the different workload runs compare to each other.

6.2 Analyzing the results

The output of each workload run is put into a comma separated value list (csv list) for easy post processing. After gathering the three output files and after post processing them, based on the reproducibility requirements specified in section 2.3.4, we can get an insight on the behaviour of each separate hardware configuration as well as comparing the separate runs with each other.

In Table 6.2 we can see a typical output of the workload, in this case running the 2007 model with 8 active cores. Each workload component returns its total runtime in seconds for each iteration. There are 10 iterations for every workload component and the result of the reproducibility analysis can be seen. We marked the CV scores in bold face(shown in column CV 1) that are higher than 2% as well as the outliers that caused the CV check to fail. By removing these marked iterations and recalculating the CV values (shown in column CV 2), the reproducibility analysis will pass the verification step. When we take all three post processed output files that complete the reproducibility analysis, we can take the average scores of each workload component and plot them in a chart.

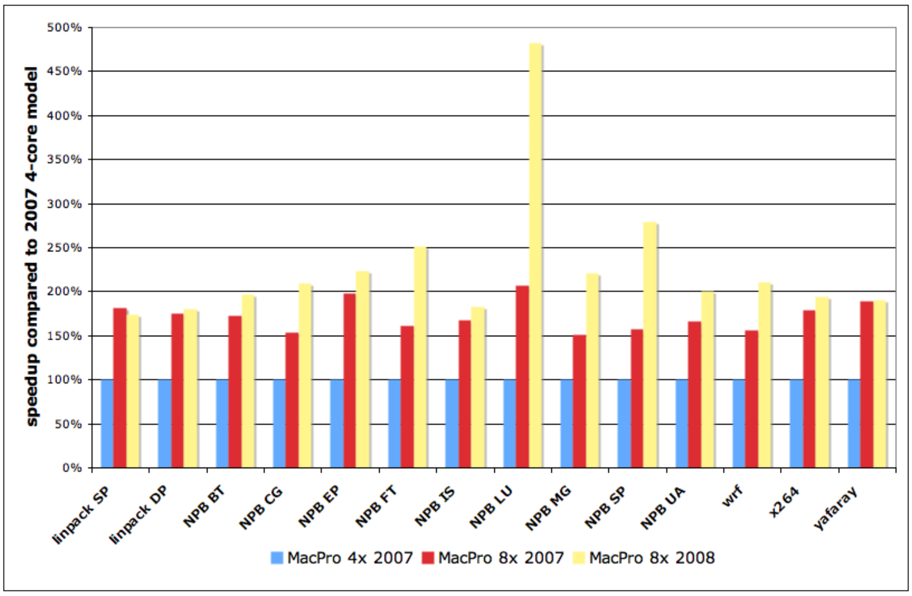


Figure 6.1: Comparing different processor models using the workload results

Figure 6.1 shows an example of a chart that plots the scalability of each configuration compared to a baseline configuration (in this case the 2007 4-core model). By looking at the chart we can see that some applications scale better than others going from 4 to 8-cores. A nice example would be NPB EP, where we have a close to linear scaling making it ideally suitable to measure scalability. As discussed in section 4.2.6, NPB EP runs almost completely locally in the caches and therefor will have little to no bandwidth bottlenecks, making the scalability dependant only on the amount of active cores. We do see that the newer 2008 model is faster than the 2007 8-core model, most likely due to the larger cache and possibly some optimizations in the processing hardware. However NPB EP does not tell us anything about the memory interconnects, since it barely uses them. So we will need to look at the other benchmarks to get an idea what will happen if we saturate another hardware bottleneck. NPB SP, discussed in section 4.2.1, showed that it is capable of saturating the available memory interconnect bandwidth. However, the chart does reveal that there is an increase in speedup when we go from 4 to 8-cores when comparing the two 2007 models with each other. While we double the amount of processing power by adding the additional cores the speedup does not double but stops

	runtime in seconds										avr	CV 1	CV 2
Iteration	1	2	3	4	5	6	7	8	9	10			
linpack SP	153.61	153.82	153.36	155.57	154.29	153.56	153.18	153.06	152.79	153.84	153.71	0.51%	0.51%
linpack DP	92.26	92.47	92.24	92.64	92.41	92.31	92.45	91.94	92.38	92.52	92.36	0.21%	0.21%
NPB BT	361.05	358.5	359.51	358.88	360.09	359.96	358.93	359.92	361.62	358.79	359.73	0.28%	0.28%
NPB CG	**167.78**	158.53	155.55	155.5	155.45	159.31	159.31	157.7	158.34	**171.98**	159.95	**3.46%**	**1.08%**
NPB EP	476.35	484.29	479.3	481.51	481.34	479.04	487.99	488.49	479.31	477.68	481.53	0.86%	0.86%
NPB FT	108.04	107.92	106.15	107.81	109.46	108.88	107.6	109.1	107.59	107.65	108.02	0.87%	0.87%
NPB IS	4.93	4.72	**5.12**	4.81	4.7	4.71	4.71	4.71	4.86	4.72	4.80	**2.86%**	**1.75%**
NPB LU	672.34	666.65	669.68	665.52	673.67	671.23	668.68	671.2	669.92	669.14	669.80	0.37%	0.37%
NPB MG	77.08	77.46	77.29	78.2	78.28	77.03	77.32	77.21	77.99	77.52	77.54	0.59%	0.59%
NPB SP	991.23	991.22	990.2	990.52	992.01	991.59	993.5	990.7	993.15	991.19	991.53	0.11%	0.11%
NPB UA	608.34	608.19	610.24	610.39	608.5	609.89	608.72	609.53	611.84	609.75	609.54	0.19%	0.19%
WRF	495.04	492.15	493.63	493	494.36	495.59	494.63	494.27	494.64	494.14	494.15	0.20%	0.20%
X264	228.02	227.05	227.37	226.91	227.41	226.79	227.13	226.03	226.43	227.34	227.05	0.24%	0.24%
Yaf(a)ray	232.691	231.236	232.322	231.41	231.092	232.353	231.378	230.933	232.02	231.565	231.70	0.26%	0.26%

Table 6.2: Reported runtimes for the 2007 model with 8 active cores

at 158% compared to the 4-core configuration. The limiting factor in this case is the memory interconnect being saturated completely and the processors are data starved. This suspicion is confirmed by looking at the results for the 2008 8-core model, which performs considerably faster. Here the increase in cache size and faster memory interconnect allows the processors to do more calculations without becoming data starved, giving a 279% improvement compared to the baseline configuration. Another interesting point in the chart is NPB LU, that shows a nice scaling from 4 to 8 cores but jumps significantly when comparing the 2007 and 2008 8-core models. The 2008 model more than doubles the performance for this application, likely due to the increase of available bandwidth by the better memory interconnect but more importantly the cache size plays an important role here. Large improvements like these, when running at the same clock frequency, often depend on the way the caches are being used. For the 2008 model the caches were likely large enough to fit the entire dataset while the 2007 models were not. When the entire dataset fits into the cache all the required data is local and little communication is needed to be fetched from memory (similar to NPB EP) allowing the processors to calculate at maximum efficiency. Some applications, like Yaf(a)ray, do not show improvements when comparing the 2007 and 2008 8-core models. What this tells us is that the execution times of these applications are not limited by either the cache size or memory(interconnect) speed, but only by frequency of computations done by the processor (since the clock frequency is the same for our test, these computations would complete at a same rate).

By interpreting these kind of charts, we can determine how a particular configuration change affects the overall performance, telling us the benefit of having an 8-core machine versus 4-core machine as well as how different hardware components influence the results.

6.3 Case study results

By repeatedly running the workload during a design process of either the system hardware or the operating system, we can find out if any changes positively/negatively influence the performance of the machine under test. We selected the workload components in such a way that various kinds of algorithmic stresses will be placed on the machine generating a broader testing field than one would get with only a single synthetic test.

We have shown that the workload can produce valid performance data even with outliers, by removing these from the scoring process, making it more robust.

Summary, Conclusions and Recommendations

7

This thesis project discussed the creation of a workload capable of evaluating multi-core systems. This workload is being used by Apple Inc. to examine the capabilities of current and next generation multi-core systems. The first chapter gives some background information about workload set development and the application domains suitable for multi-core workload components. The component selection procedure is also explained in this chapter. The second chapter introduces the concept of Dwarfs which are common scientific algorithms. We will target these specific algorithmic methods specifically when selecting components for the workload. The relevant workload areas and the criteria for the workload are also discussed in this chapter. The third chapter will discuss the workload environment and parallelization methodologies. The application profiling tools that were used are also discussed in this chapter. The fourth chapter will discuss the various scientific applications we selected for the workload and an in depth analysis on each application. The fifth chapter will discuss the two media applications that we selected for the workload and an in depth analysis on each application. In the following sections we will give a summary of these chapters and our conclusion and recommendation for future work.

7.1 Summary

In this section we present a summary of our findings for each chapter in this thesis.

7.1.1 Workload set development summary

The objective of this project is to create a workload that is capable of giving the user insight into the capabilities of a particular multi-core system. Creating a workload set is a multifaceted task where the designer has to define what the workload will test and what kind of criteria it needs to follow. A multi-core capable workload requires applications that are parallel and are capable of scaling. To select these real world applications we looked at several candidate domains, consisting of scientific and media applications. We then narrowed our domain to the high performance computing sub-domain that is part of scientific domain and the rendering and encoding sub-domains of the media domain.

7.1.2 Workload characteristics

The workload components are selected based on the concept of Dwarfs discussed in chapter 2. These Dwarfs are types of algorithms that are common in science and engineering. All selected application will contain one or more of these Dwarfs and will exercise the main hardware elements of the underlying hardware, the CPU, main memory and the memory interconnect (NorthBridge). The 5 criteria for the workload are defined as follows:

1. The workload should have full support for multi-core systems, meaning it should take advantage of all of the available cores.
2. Full support for Mac OS X systems
3. The workload needs to complete and give results within a reasonable time of 12-24 hours, so you can start your test and get results the next day.
4. Reproducible test results on the same platform.
5. The workload needs to produce verified results that do not change when run on other platforms

The workload developed in this thesis met al of these criteria.

Criterion 1 Full multi-core support is implemented by only using applications that are capable of running in parallel and can scale to multiple cores.

Criterion 2 Full Mac OS X support has been met by developing and running the workload on the Mac OS X platform.

Criterion 3 Workload completion time has been met due to the fact that the workload will complete a full run in 14 hours on our test system using 8 cores.

Criterion 4 Reproducibility is assured by selecting applications that have consistent output and runtimes and the minor variances in the runtime can be detected when post-processing the results as described in section 2.3.4.

Criterion 5 Verification of results is implemented by every component in the workload by either internal verification of the computed results or comparison of the output based on a reference file. When either one of these verification steps fails, it will be reported.

For the workload we will use a constant dataset, so we can calculate the total time an application requires to complete and we can use Amdahl's law and the Karp-Flatt metric to estimate the scalability of these applications beyond the actual available core count. The reproducibility is another major issue we discussed in chapter 2, were we discuss the methodology to cope with variance in the results and possible outliers. By using this methodology we can be satisfied that the coefficient of variation will be below 2% for each application when the final results are processed.

7.1.3 Workload evaluation environment summary

The various parallelization methodologies we looked at were MPI, Pthreads and OpenMP. We chose to use OpenMP extensively in our project since it has the capability of scaling the number of threads dynamically based on the underlying hardware and the benefit of a shared memory model. For the application profiling tools we looked at two toolsets: Apple Shark and Intel Pin Tools. We decided to continue with Shark since it has the capability of profiling both the hardware and the software. Intel Pin Tools is only a software profiling tool with the additional problem that the 64-bit binaries were not supported. Also several bugs in the software prevented us from reliably profile double precision floating point applications. Another benefit of Shark over Pin Tools is that Shark is capable of profiling low level hardware calls in real time whereas Pin Tools is only capable of this in emulation mode that is significantly slower. We also looked into the influence our analysis might have on the actual performance of the application while profiling and came to the conclusion that using Shark with its hardware assisted sampling methods the overhead is less than 1.5% for the most demanding profile setup.

7.1.4 Scientific benchmarks summary

We selected several scientific benchmarks to be included in our workload. We selected Linpack, the NASA Parallel Benchmark Suite (NPB) and Weather Research and Forecasting (WRF) applications.

Linpack was ran in both single precision and double precision with a 30.000x30.000 and 20.000x20.000 matrix respectively. Linpack scales exceptionally well using these large matrices, reaching the computational and bandwidth limits of our test machine.

NASA Parallel Benchmark consists of 8 applications that mimic a class of computational fluid dynamics. For each of these applications a detailed study of their behaviour has been done as well as a scaling study. The following scaling behaviour was noticed on our test system:

- **Scales exceptionally well:** BT IS EP CG
- **Scales well:** FT LU
- **Scales moderately:** UA MG
- **Scales poorly:** SP

The applications SP, UA and MG have poor or moderate scalability because they reach a bottleneck in the current system. For SP, UA and MG it reaches the maximum available memory bus bandwidth at 4 active cores. All the applications have internal verification steps to check the computed results.

Weather Research and Forecasting is a weather modeling application that in our model computes the weather for the entire continental US. The scalability of WRF is limited due to the cache coherency requirement when running OpenMP, when scaling to 8-cores the overhead and the resulting bandwidth pressure limits scaling. Since weather modeling is not an exact science, the results are expected to vary between the runs, but we will only accept results that are the same to within the 3rd correct decimal.

7.1.5 Media benchmarks summary

We selected two media benchmarks one from each sub-domain, encoding and rendering.

Yaf(a)ray is a raytracer designed to render image blocks independently of its neighboring blocks and has therefore close to linear scalability. There is a significant amount of memory traffic on the memory bus but not enough to saturate the Front Side Bus. We collect every render for each iteration we run and calculate the Peak Signal to Noise Ratio (PSNR) on the difference between the output file and the reference render image.

x264 is a video encoder that is capable of encoding individual frames in parallel. It uses over-subscription to sidestep the issues with fast vertical motion vectors that create dependency chains in the encoding order. x264 scales exceptionally well, but has some problems with load-balancing, something that can be expected when oversubscribing the actual number of available cores. The x264 encoding has internal mechanisms to verify the output quality after encoding. It uses the peak signal-to-noise ratio (PSNR) and structural similarity index (SSIM) to verify the encoded image against the source frame after encoding, to see if the degradation is within acceptable limits.

7.2 Conclusions

The completed workload provides a good basis covering the Dwarfs mentioned in section 2.1. We successfully completed all the requirements specified at the start of this project. The workload can and is used in the computer industry for testing multi-core systems. As we have shown in the case study, the workload can be used to generate performance data on several machines for comparison. Important information regarding the effects of architectural changes can be derived by comparing the results of different machines, such as:

- The effects of cache size, determines if a particular dataset can fit into the cache so the computational elements can be fed with instructions without interruption.
- The effects of memory interconnect bandwidth, if the bandwidth is limited the computational elements will be starved for data and overall performance will suffer.
- The effects of functional unit improvements, if the clock rate and the memory interconnect bandwidth remains the same significant improvements in the computational elements can be spotted.

The workload in its current form, gives the user significant coverage of various computational areas while giving additional information about architectural limitations/improvements. At the same time the workload fulfills the requirements set at the start of the thesis project, in the sense that it is scalable, reproducible and verifiable workload running within a reasonable runtime on current generation hardware.

7.3 Recommendations

There are still several interesting improvements that can be made with continued development of this workload.

- Addition of the three remaining Dwarfs: N-Body Methods, Combinational Logic and Graphical Models. With the addition of these remaining Dwarfs we complete the entire set of common algorithms used in science and engineering as defined by Berkeley [6].

- Additional sub-domains such as Bio-Informatics or Intel Recognition, Mining and Synthesis would be a good addition to the workload. With the addition of different sub-domains we can create an even broader testing base for the workload and potentially cover additional data access patterns generated by these other sub-domains.

- Support for Linux and Microsoft Windows would make the workload cross-platform and could give additional insights in the differences on an OS level.

- Increased robustness support so that the workload can continue running even after restarts or machine panics on experimental hardware, to make it even more suitable for automated testing environments used in the industry.

Bibliography

[1] *Big buck bunny*, `http://www.bigbuckbunny.org/`.

[2] *Blender 3d modeling*, `http://www.blender.org/`.

[3] Laurent Aimar, Loren Merritt, Eric Petit, Min Chen, Justin Clay, Måns Rullgård, Radek Czyz, Christian Heine, and Alex Izvorski Alex Wright, *x264 h.264 encoder library*, `http://www.videolan.org/developers/x264.html`.

[4] G.M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, AFIPS Conference Proceedings **30** (1967), no. 8, 483–485.

[5] Apple, *Computer hardware understanding developer tools (chud)*, `http://developer.apple.com/tools/performance/`.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick, *The landscape of parallel computing research: A view from berkeley*, Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, *The nas parallel benchmarks*, The International Journal of Supercomputer Applications **5** (1994), no. 3.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, *The parsec benchmark suite: Characterization and architectural implications*, Tech. report, Princeton University, 2008.

[9] Shekhar Borkar, Pradeep, Dubey, Kevin Kahn, David Kuck abd Hans Mulder, Steve Pawlowski, and Justin Rattner, *Platform 2015: Intel processor and platform evolution for the next decade*, Tech. report, Intel, 2005.

[10] P. Colella, *Defining software requirements for scientific computing*, presentation, 2004.

[11] Jack Dongarra, Jim Bunch, Cleve Moler, and Pete Stewar, *Linpack*.

[12] Pradeep Dubey, *A platform 2015 workload model recognition, mining and synthesis moves computers to the era of tera*, Tech. report, Intel, 2005.

[13] Huiyu Feng, Rob F. Van der Wijngaart, Rupak Biswas, and Catherine Mavriplis, *Unstructured adaptive (ua) nas parallel benchmark, version 1.0*, Tech. report, NASA Ames Research Center, 2004.

[14] Huiyu Feng, Rob Van der Wijngaart, and Rupak Biswas, *Design of unstructured adaptive (ua) nas parallel benchmark, featuring irregular, dynamic memory accesses*, Tech. report, NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, 2001.

[15] R. Fovell, `http://macwrf.blogspot.com/`.

[16] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, *Introduction to parallel computing*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[17] W. Heisenberg, *Über quantentheoretische umdeutung kinematischer und mechanischer beziehungen*, Annalen Der Physik **33** (1925), no. 1, 879–893.

[18] Jim Held, Jerry Bautista, and Sean Koenhl, *Research at intel from a few cores to many: A tera-scale computing research overview*, Tech. report, Intel, 2006.

[19] H. Jin, M. Frumkin, and J. Yan, *The openmp implementation of nas parallel benchmarks and its performance*, Tech. report, NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, 1999.

[20] Alan H. Karp and Horace P. Flatt, *Measuring parallel processor performance*, Commun. ACM **33** (1990), no. 5, 539–543.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: building customized program analysis tools with dynamic instrumentation*, SIGPLAN Not. **40** (2005), no. 6, 190–200.

[22] Loren Merritt, *x264: a high performance h.264/avc encoder.*

[23] G.E. Moore, *Cramming more components onto integrated circuits*, Electronics **38** (1965), no. 8, 114–117.

[24] Kunle Olukotun and Lance Hammond, *Chip multiprocessors: The future of microprocessors*, Queue **3** (2005), no. 7, 26–29.

[25] PASC, *Standard for information technology - portable operating system interface (posix). shell and utilities*, IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell and Utilities (2004), –.

[26] M. Rabbani and P.W. Jones, *Digital Image Compression Techniques*, Society of Photo-Optical Instrumentation Engineers (SPIE) Bellingham, WA, USA, 1991.

[27] R. Ramananthan, *Intel multi-core processors leading the next digital revolution*, Tech. report, Intel, 2005.

[28] ———, *Architecting the era of tera*, Tech. report, Intel, 2006.

[29] Steven M. Rubin and Turner Whitted, *A 3-dimensional representation for fast rendering of complex scenes*, SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques (New York, NY, USA), ACM, 1980, pp. 110–116.

[30] William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow, *New implementations and results for the nas parallel benchmarks 2*, Tech. report, NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, 1997.

[31] SPEC, *Spec background*, `http://www.spec.org/spec/#background`.

[32] ———, *Spec cpu 2006*, `http://www.spec.org/cpu2006/`.

[33] ———, *Spec retired benchmarks*, `http://www.spec.org/retired.html`.

[34] Sun, *Ultrasparc t2 processor*, Tech. report, Sun microsystems, 2007.

[35] J. Verwiebe, *Yaf(a)ray source file*, `http://www.jensverwiebe.de`.

[36] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli, *Image quality assessment: from error visibility to structural similarity*, Image Processing, IEEE Transactions on **13** (2004), no. 4, 600–612.

A

Torque detailed study

A more detailed study of the behaviour of Torque was done to explain the variability and the second peak in the Torque histogram shown in Figure 2.2. Since the initial test was run without cache-affinity, the 8 processes were run with 2 different memory access types (4 Load processes and 4 Bzero processes) were allocated at random on the 8 available cores. The resulting throughput numbers, because of the large number of runs, display all possible combinations of cache localization. While the majority of the throughput numbers form one large group around the mean, there are other, smaller groups further away from the mean, but still within the standard deviation range. Cache affinity, a process where the schedular forces a process to stay with a specific cache, was used to bind the processes to a specific shared cache, with two cores sharing the same Level 2 Cache. This resulted in 4 possible cache configurations when two processes of the same type (Load Load (L) or Bzero Bzero (Z) pairs)were always bound to the same shared cache, with options of LLZZ, ZZLL, LZLZ and ZLZL. For example LLZZ where the first 4 cores (with their 2 shared caches) are occupied with Load processes and the last 4 cores (with their 2 shared caches) are occupied with Bzero processes. For these configurations, an exhaustive test of 100,000 iterations per configuration was performed.

The resulting performance numbers were plotted along with the completely random test results. The resulting graph, shown in Figure A.1, shows us that the mixed configurations LZLZ and ZLZL both cluster on the mean, where the total Load and Bzero number is added to get around 5750MB/Second), and are not the configurations that cause the higher performance. The configurations that have 4 processes of the same class (LLZZ and ZZLL) paired together on the same physical package/socket were observed to be in the higher performing groups. It is interesting to note that the placement of the group on the first package defines the difference in performance benefiting either the Load or Bzero group. Since the processors are identical from the core level all the way up to the package level, it is our theory that the differences in performance must be caused by the memory controller in the North-Bridge. Package 0 gets priority in some cases, resulting in a minor speed gain. The most likely cause for this priority is that the default state for the memory controller is package 0 / FSB 0. The remaining scattering of performance numbers are caused by the influences of the operating system's scheduler and the interference caused by different processes fighting over the same cache ranges.

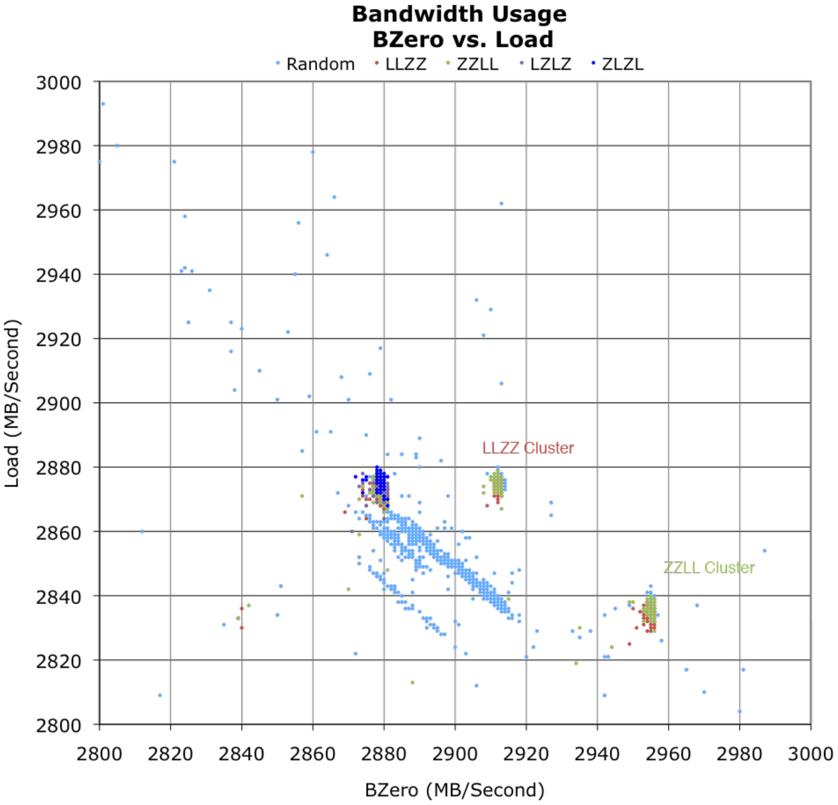


Figure A.1: Torque bandwidth usage

B

Leviathan test harness

All the applications in this workload are open-source applications. Each application is designed to operate independently of the other applications so that each application folder can be taken out of the workload and executed independently of the main test harness. Each application is built as plugin structure in the main test harness.

To add another workload component, you just need to perform three steps:

1. Find the leviathan shell script and add the name of your application to the supported_applications list at the top.
2. Make a new folder with the name you added to the supported_applications list, and put a copy of your application there, along with any necessary datafiles.
3. Create a new function with the application's name in the run script that invokes your application.

The default way to run a application is to call the run script from within the application folder. Running the leviathan shell script can be done with various arguments:

-a Runs all the available workload components and output a report.

-h Display the help information.

-l LABEL Add a label to the output folder. This argument needs to be placed before the -a and -m arguments.

-m APP_NAME Only runs the selected application, not producing a report. Can be used several times to specify multiple applications.

-r Generates a report when running applications selected using -m.

-R TSTAMP Generates a report for a selected timestamp. This should not be used with other arguments.

-t Runs the workload with a testset. This must be the first argument.

Examples of how to use the leviathan shell script:

./leviathan.sh -a Run the entire workload and generate a report.

./leviathan.sh -t -l runtest -a Run the entire workload with only the testset and add the label "runtest" to the output names.

./leviathan.sh -m linpack -m x264 -r Run only "linpack" and "x264" and produce an report based on those two applications.

The -a/-r/-R arguments all generate reports in a comma separated value list called report_timestamp.csv. This report can be imported into Excel or other programs for post processing. All the data collected during the run is put in the "results" directory under the same timestamp or label.

Before the leviathan workload can be used, all the components need to be unpacked and built. By running the unpack_all script in the main directory, the first step of unpacking the workload is done. Afterwards, for each component the just_build script must be invoked in each sub-directory.

B.1 Linpack

B.1.1 Where to get

`http://www.netlib.org/benchmark/`

B.1.2 How to build

This application has a build- and run-script called apglinpack. To build your own binaries Intel ifort and icc is required, as well as the Intel MKL libraries that supply optimized math functions used by Linpack. Changes need to be made to the apglinpack script if you use a version of the Intel compilers older than version 9 or newer than 11. To build all the executables, run the just_build_all scripts. Based on the amount of memory installed in your system, the larger matrices will cause disk paging if you run out of memory. To calculate the maximum memory size your system can handle, use the following equations:

$$\begin{aligned}\sqrt{\frac{total\ memory\ available - OS\ X\ memory\ usage}{32\ bits}} &= SP\ Matrix\ Size\\ \sqrt{\frac{total\ memory\ available - OS\ X\ memory\ usage}{64\ bits}} &= DP\ Matrix\ Size\end{aligned} \tag{B.1}$$

B.1.3 How to run

Linpack can be executed in several ways, depending on what you would like to do. Running linpack using the leviathan shell script can be done simply by calling the leviathan script with the "-m linpack" argument. To run linpack without the test harness, you can call the run_linpack script directly with the appropriate arguments. The arguments are:

-T Gives linpack a special label for its output.

-m MATRIX_SIZE Runs linpack with the selected matrix size. Supported sizes are small, medium, large, insane.

Examples of how to call the run_linpack script:

./run_linpack.sh -m large Runs linpack with the large size selected and outputs to a folder named using the current time.

./run_linpack.sh -T runtest -m small Runs linpack with the testset with the label "runtest."

For all runs with either script, an output file called "batch_results.csv" will be produced in the linpack folder, where all the aggregated test results are combined into a comma separated value list for easy Excel importing, for post processing.

B.2 NPB

B.2.1 Where to get

The NASA Parallel Benchmarks are available on the NASA NPB website: `http://www.nas.nasa.gov/Resources/Software/npb.html`. You will need to register to download the software.

B.2.2 How to build

The default configuration file from NPB's config directory, that contains the compiler settings, was modified to use the Intel compiler suite and compile static binaries. When using this configuration file, and assuming that you have the required Intel compilers installed, running the just_build shell script will build the entire NPB suite. After building, all of the benchmarks will be placed in the bin directory.

B.2.3 How to run

NPB can be run as a separate application suite using the run_npb shell script inside the application directory, or using the leviathan test harness. The run_npb script will accept a label argument that will add a label to the output files. Additionally, NPB supports two datasets: the normal dataset or the testset, which is executed when the environment variable TESTSET=1 is set. The output will be placed in the npb_results folder in the application directory.

Example of how to call the run_npb script:

./run_npb.sh LABEL Will run npb with the label added to the output.

B.3 x264

B.3.1 Where to get

`http://www.videolan.org/developers/x264.html`

B.3.2 How to build

To build the x264 codec you will need to build the support software and the main application. Run the just_build script to unpack and build the support software and the x264 encoder. To build the x264 encoder and use the optimized assembly routines, Yasm is required. Additional changes to the source code have been made to increase the efficiency of the code by replacing malloc() with alloca() in the encoder/me.c file. This will allow x264 to use the stack for quick allocations of small chunks of temporary memory.

List of support software in the support directory:

Yasm `http://www.tortall.net/projects/yasm/`

B.3.3 How to run

The input files are stored in the input directory in a compressed format. They will be unpacked automatically the first time you run the application via the run_x264 shell script. The x264 encoder settings we used to encode the HD 1080 sample movie are the following:

–keyint 250 Maximum interval between keyframes.

–bframes 16 Maximum number of concurrent B-frames.

–ref 16 Maximum number of previous frames that each P-frame can use as a reference.

–b-pyramid Allows x264 to use B-frames as reference for other B-frames.

–crf 19 Constant ratefactor determines the output quality of the movie.

–partitions all Allows x264 to use all available macroblock sizes.

–direct auto Allows x264 to choose between spatial and temporal direct motion vectors.

–weightb Allows x264 to weight B-frames how they affect other frames.

–me tesa Sets x264 to use transformed exhaustive motion estimation.

–subme 7 Sets x264 to use subpixel motion estimation with Rate Distortion Optimalization (RDO).

–b-rdo Allows RDO for B-frames.

–mixed-refs Allows references to be selected by 8x8 partition ratio.

–bime Enables bi-directional motion estimation for B-frames.

–8x8dct Enables 8x8 discrete cosine transforms.

–trellis 2 Sets x264 to use higher Trellis quantization to increase efficiency.

–no-fast-pskip Disables skip detection on P-frames to improve quality.

–progress Displays the progress indicator while encoding.

–threads auto Lets x264 decide how many threads to use, defaults to 1.5x number of available cores. For better performance set to 2x.

-o OUTPUTFILE Name of the output file.

INPUTFILE The input file to be encoded, must be the last argument.

B.4 Yaf(a)ray

B.4.1 Where to get

The main page where to get Yaf(a)ray is `http://www.yafray.org/index.php`. The howto wiki to build Yaf(a)ray is `http://wiki.yafray.org/bin/view.pl/UserDoc/YafaRay`.

B.4.2 How to build

To build yaf(a)ray use the supplied just_build script. This will unpack and build all the required support packages and the yaf(a)ray main package. The support packages will be built from the yafaray_support directory and will be installed in the yafaray_install directory. Here is a list of items in the support directory:

Freetype `http://freetype.sourceforge.net/`

Ilmbase `http://www.openexr.com/downloads.html`

ImageMagick `http://www.imagemagick.org`

Libjpeg `http://www.ijg.org/`

Libpng `http://www.libpng.org/pub/png/libpng.html`

Libxml2 `http://xmlsoft.org/`

OpenEXR `http://www.openexr.com/downloads.html`

Scons `http://www.scons.org/`

Zlib `http://www.zlib.net/`

B.4.3 How to run

Yaf(a)ray accepts XML formatted description files made by the open source 3D modeling application Blender [2]. The input models are located in the yafaray_bottle directory. This directory also contains the run_yafaray script and the support scripts. The current version of yaf(a)ray does not support dynamic switching of the number of threads used to render a scene, since the thread count is hardcoded into the XML description format. The yafaray_threads_selector script will modify the XML file based on the number of available cores on the system the application is running on. This will produce a yafaray_thread_set XML formatted input file for actual rendering. Like all the applications in the workload, Yaf(a)ray has two input sets to be rendered, one is a testset that will render significantly faster than the full render set. Here is an example of how to call the run_yafaray script:

./run_yafaray.sh LABEL Runs Yaf(a)ray and outputs the result with the passed argument as its label.

B.5 WRF

B.5.1 Where to get

`http://www.mmm.ucar.edu/wrf/users/downloads.html`

B.5.2 How to build

There are some special requirements that need to be met for WRFV221. The first requirement is that WRF requires a Case-Sensitive file system. If your hard drive is not already formatted this way, you can simulate it by running a Case-Sensitive disk image mounted at /Volumes/wrf. The second requirement is that the support software and the wrf applications must be compiled by the same compiler and compiler version. A modified configure.defaults was made to allow WRF to work with the Intel fortran compiler, although some of the modifications will not be necessary in future versions of the Intel Fortran compiler. The FCOPTIM option -mP2OPT_vec_xform_level is one of the modified options. When running WRF needs to know where the NETCDF shared library is. Therefore, in the run_wrf script, we export the NETCDF environment variable to point to the correct library path. We default to the 64-bit version. Using OpenMP as our threading model also required us to set some special OpenMP stack flags to allow WRF to run with large datasets. The OpenMP KMP_STACKSIZE was set to 500MB of virtual memory to allow the stack to grow large enough to prevent application crashes, since the default 8MB was not sufficient to run our selected dataset. Up-to-date information can be obtained from the website of Prof. R. Fovell `http://macwrf.blogspot.com/`

B.5.3 How to run

We use the run_wrf shell script to execute the application. This script can be called either by the leviathan test harness or directly from the WRF folder. When we do not use the leviathan test harness, we need to manually mount and unmount the disk image to provide access to the WRF executables. The leviathan test harness will mount and unmount the disk image when necessary, and will perform additional cleanup to reclaim wasted disk space within the disk image. If the environment variable TESTSET is set to 1, WRF will execute with a testset that can run significantly faster. The run_wrf script will also accept an argument to change the label of the output. Examples of how to call the run_wrf script:

./run_wrf.sh LABEL Will run WRFV221 and will output the result with the passed argument as its label.

Curriculum Vitae

Erick Martijn van Rijk was born in Utrecht, The Netherlands on March 5th, 1980. He obtained his VWO degree in 1999 at Instituut Vrijbergen in Leiden. In Januari 2005 he received his Bachelor of Engineering from HTS Haarlem by completing the Electrical Engineering Programme with a minor in Computer Engineering. He joined the Computer Engineering department of Delft University of Technology in February 2005. He will graduate in April 2009 by completing his MSc. thesis *Development of a workload set for multi-core architectures*