Performance Profiling with Cactus Benchmarks

Sasanka Madiraju

April 6th, 2006

System Science Master's Project Report

Department of Computer Science

Louisiana State University

Acknowledgment

This project has evolved out of a cumulative effort at the Center for Computation & Technology towards the proposal written in reply to the bigiron solicitation from NSF. Over a span of six months there was a lot of activity to collect and organize data and participate in rigorous analysis. I would like to thank all the people who helped me during the course of this project. I would like to thank my committee members Dr. Gabrielle Allen, Dr. Edward Seidel and Dr. Gerald Baumgartner, for their support and suggestions. I would also like to offer my appreciation to Dr. Thomas Sterling and Dr. Ram Ramanujam for their constant encouragements and help.

I would especially like to thank Dr. Erik Schnetter and Dr. Maciej Brodowicz for their constant advice and support and I am grateful to them for their exceptional guidance. I am also grateful to Mr. Yaakoub El-Khamra and Dr. Steven Brandt for their help and suggestions. Finally, I would like to thank all the people at CCT for being a source of inspiration.

Sasanka Madiraju

Contents

1 Introduction **1**

1.1 Project Scope 1

1.2 Goals 3

1.2.1 Phase 1 3

1.2.2 Phase 2 4

1.3 Contributions and People Involved 4

2 Cactus Toolkit: Benchmarks **6**

2.1 Cactus Toolkit 6

2.2 Types of Benchmarks 8

2.3 `Bench_BSSN_PUGH` 9

2.4 `Bench_Whisky_Carpet` 9

2.5 Duration of Run 10

2.6 Problems with IPM Results 10

3 Phase 1: Analysis of Results **12**

3.1 IBM Power5 Architecture 12

3.2 Intel Itanium2 Architecture 13

3.3 Intel Xeon Architecture 14

3.4 Details of Machines Used 15

3.5 Classification of Results 15
3.5.1 Single Processor Performance Results 16
3.5.2 Scaling Results for different machines 19

4 Phase 2: Profiling `Bench_BSSN_PUGH` **29**

4.1 Reason for Profiling 29
4.2 Tools used 30
4.2.1 PAPI: Performance Application Programming Interface 31
4.2.2 Valgrind: A linux based profiling tool 33
4.2.3 Valkyrie: A GUI based tool for Valgrind 34
4.2.4 gprof: GNU Tool for profiling 34
4.2.5 VTune: Intel Performance Analyzer 35
4.2.6 Oprofile: Linux based Profiling tool 35
4.3 Results and Analysis 36
4.3.1 Gprof Results: 36
4.3.2 Valgrind Results 39
4.3.3 Oprofile Results 40

5 Conclusion **44**

Chapter 1

Introduction

1.1 Project Scope

Modeling real world scenarios requires high performance systems capable of performing extremely large number of concurrent operations. The increasing complexity included in these models implies, there will always be a perennial demand for faster and more robust computing resources. There is a cyclic dependency between the applications and the resources as the applications are being driven by the presence of the resources and the resources are being enhanced in order to accommodate the demands of the applications.

Supercomputers and cluster systems are used for performing operations on similar kinds of data, but on a gigantic scale. For example, an average desktop system can perform several million complex calculations per second. Supercomputers and clusters on the other hand can perform extremely large number of operations per second when compared to a stand alone system. In a very simplistic sense a supercomputer is a collection of large number of smaller computers, put together to exploit the availability of computational power. It is a well know fact that, over the past few decades the field of Computer Engineering has shown exponential growth in terms of microprocessor technology, resulting in overall development

of computing resources and architectures. Hence there is a need to understand and improve the performance of the existing computing resources to accommodate the requirements of these applications.

One tricky aspect of using supercomputers is to come up with applications that can maximize the utilization of available resources. This is not easy as there are a number of limiting factors that prevent maximal utilization of the existing resources. So, there are two solutions to this problem, understand the limiting factors and try to find a work around for them or try to add measured resources to the system to improve its performance. The major goal of this project is to understand the limiting factors and come up with a detailed analysis of the results.

In order to write better code, it is imperative to understand the working of the application on the given architecture. This is not a simple task as the possible number of architectures and machine configurations is quite large, bordering on being limitless. Each computational resource is unique in terms of the processor architecture, peak performance, type of interconnect, available memory and the I/O system used.

The aim of this project is to study the performance characteristics of Cactus [6] benchmarks on different architectures and analyze the results obtained from the benchmark runs. Cactus [6] is a computational toolkit used for problem solving in different scenarios by scientists and engineers. The reason Cactus [6] was picked for profiling was, it is a good example of a software that has excellent modularity and portability. It is known to run on different architectures and on machines of different sizes. For example, these benchmarks can be run on a laptop or standalone system or they can just as easily be run on a supercomputer or a cluster. Also Cactus [6] supports many technologies like HDF5 parallel file I/O, the PETSc scientific library, adaptive mesh refinement, web interfaces and advanced visualization tools.

The study of profiling results also helps to understand the different aspects of the system architecture that effect the performance of applications. The project also aims to turn on the

trace on these benchmarks so that the time spent in different operations can be measured. A number of profiling tools like PAPI [1], IPM [2], gprof [8], valgrind [4], Oprofile [3], are used to collect information about the total number of floating point operations performed, the number of Level1, Level2 and some times Level3 data as well as instruction cache misses, and call graph information. This information can be used to tweak the performance of the benchmark. The results of the first phase of the project show a need to understand the benchmark in detail. Hence this project not only tries to understand the performance of the code on different architectures but also tries to find avenues to optimize the benchmark code.

1.2 Goals

This project aims to understand the performance characteristics of the Cactus [6] benchmark `Bench_BSSN_PUGH` on different architectures. This analysis helps to understand the architectural aspects of such complex systems. Also, this study aims to provide a basis to tweak the performance of the actual benchmark so that it can become more efficient. This project is divided into two phases. The following sections explain the details of the two phases.

1.2.1 Phase 1

The first phase of the project deals with running Cactus [6] benchmarks on different architectures and collecting the run statistics. These results are analyzed and the reasons for the differences in the wall times are studied in detail. This phase of the project was important for collecting results for the recent CCT proposal in reply to the NSF "bigiron" solicitation. In this phase, it was found that among different NSF benchmarks the Cactus [6] benchmarks "Bench_BSSN_PUGH" and "Bssn_Whisky_Carpet" were found dominated by load and store operations. The reasons for this result are studied in detail in the second phase of the

project.

1.2.2 Phase 2

The second phase of the project involves profiling Cactus [6] benchmark `Bench_BSSN_PUGH`. This benchmark was chosen over the `Bench_Whisky_Carpet` benchmark as it is already known to scale well and shows more consistent behavior. `Bench_Whisky_Carpet` benchmark also shows consistent results, but if the need arises to run it on a large number of processors its performance deteriorates rapidly making it unsuitable for profiling. Moreover both the benchmarks have similar kinds of code with the main change being the kind of driver being used, as one uses PUGH (structured unigrid) and the other used CARPET (adaptive mesh refinement). Profiling tools like "IPM" [2], "PAPI" [1], "gprof" [8], "valgrind" [4] and "Oprofile" [3] were used to investigate the reasons for the number of load and store operations. Also, it is clear that the load and store operations happen when there are a lot L2 (or L3 if there is one) cache misses, due to which the processor needs to fetch the instructions from the cache. Hence, the profiling tools are used to identify those parts of the code, which are responsible for a majority of cache misses at L2 or L3.

1.3 Contributions and People Involved

My contributions to the proposal submitted by CCT for the NSF "bigiron" solicitation, have been compiled and further analysis is added as a part of this project. A lot of data and results from the benchmark runs went into the proposal. There were a number of other benchmarks, that were run by other members of the team which are not part of this project. Results from the Cactus [6] benchmark runs on the NERSC machine called Bassi, were provided to me by one our team members Dr.Erik Schnetter. More over a lot of information about the architecture and the kind of runs that need to be performed and invaluable suggestions

about the analysis of data were provided to me by Dr. Maciej Brodowicz and Dr. Erik Schnetter. All the results from the NCSA machines and LSU machines, were collected by me and I have compiled a lot of information about different machines in different institutes. Also, I have compiled a large set of results from the profiling of Bench_BSSN_PUGH using various profiling tools. This effort can be used by the Center for future research and especially the Performance group would benefit from this data. Most of this information was used for the BigIron proposal team which comprised of Dr.Gabrielle Allen, Dr.Edward Seidel, Dr. Thomas Sterling, Dr. Steven Brandt, Dr. Erik Schnetter, Dr. Maciej Brodowicz and me with help from many others at the Center for Computation and Technology at LSU. I would also like to give ownership of the results from the runs on PSC machines like the BigBen-Cray XT3 and Lemieux to Mr. Yaakoub El-Khamra.

Chapter 2

Cactus Toolkit: Benchmarks

2.1 Cactus Toolkit

Benchmarking different achitectures can be quite tricky as it is difficult to find an application which is not optimized to run on one type of architecture. Most real world applications are optimized to run well on specific platforms and hence are not suitable to be used as benchmarks across different platforms. In order to measure and understand the true performance of a machine there is a need for highly portable application which is not biased towards one particular architecture and also it would help if it is open source. This is where Cactus [6] Toolkit comes into play as it is an open source, modular, portable, programming environment for collaborative HPC computing. Since we are planning to understand the architectural issues in HPC it is relevant to use an application which has high degree of generic parallel computations, with modules providing parallel drivers, coordinates, boundary conditions, elliptic solvers such as PETSc, interpolators, reduction operators, and efficient I/O in different data formats. Interfaces are generic, (e.g. an abstract elliptic solver API) making it possible to develop improved modules. Moreover, Cactus [6] has excellent provisions for visualizing simulation data for analysis, as it employs a range of external applications, such as

Amira, IDL or OpenDX. The data can also be analyzed in-line by use of a Web-server module. Cactus [6] was mainly developed by Max Planck Institute for Gravitational Physics in Potsdam, Washington University, Lawrence Berkeley National Laboratory, National Center for Supercomputing Applications, University of Tuebingen and many other scientific groups around the world. Currently, most of the development is done by the Frameworks group at Center for Computation & Technology and researchers at Postdam in Germany. Cactus [6] is being used and developed by many application communities internationally. It is used in Numerical Relativity, Climate Modelling, Astrophysics, Biological Computing and Chemical Engineering and it acts as a driving framework for a number of computing infrastructure projects, particularly in Grid Computing, such as GridLab and GriKSL.

Cactus [6] consists of "flesh" and a set of "thorns" or modules that can be used to make different configurations. Cactus [6] is extremely portable, modular and robust software as there is a good degree of freedom for the programmer to add new thorns to the distribution and create new configurations. Each thorn performs certain tasks and the same task can be performed by different thorns. This means for a given configuration, a thorn which performs certain function can be replaced by another thorn which offers the same functionality. This is the advantage of modular design where we can plug the thorns in and out of the flesh to get different configurations. This enabled us to easily use an existing PAPIClocks thorn which is a high level API to keep track of the hardware counters in the system. We used the PAPIClocks thorn by adding it to the thornlist of `Bench_BSSN_PUGH` and creating a configuration with it. Then we use a new parameter file which can be used to count the hardware events.

Each Cactus [6] configuration needs a parameter file to set the size or resolution of the benchmark and there are two parameter files for each of our benchmarks. So, there were a total of four benchmarks that were run on different architectures. Each parameter file is used to control the simulation and it has many options which can be set. These options are

present in the param.ccl file which is part of the benchmark. So, we have a parameter file which has a fixed set of parameters which can be tweaked to perform different operations and these options or parameters are listed in the param.ccl file in the benchmark source code. The .ccl extension stands for Cactus [6] Configuration Langugage, which is a high level language used to set things like the order in which the functions are called during execution, the way different modules interact with each other, dependencies among all the thorns, the active thorns and mainly the parameters available to the user. There are three files which keep track of all these factors, schedule.ccl, interface.ccl and param.ccl.

2.2 Types of Benchmarks

A benchmarking suite should have a good mix of benchmarks designed to test different aspects of a system. Cactus [6] toolkit has an excellent suite of benchmarks, and hence a few of the benchmarks from this suite are a standard subset of the overall benchmarks used by NSF. The benchmarks present in the Cactus [6] Toolkit can be found on the Cactus webpage [6]. In this list, the main benchmarks that we are interested in are `Bench_BSSN_PUGH` and `Bench_Whisky_Carpet` as these benchmarks were choosen to complement the NSF benchmarks for the "bigiron" slicitation. PUGH and Carpet are the drivers that provide MPI Communication and memory management to the underlying physics simulation. PUGH is known to be a very scalable driver whereas Carpet is not scalable. From the results we can see that Carpet does not scale well beyond 100 processors. After that the simulation time is really large and the nodes could run out of memory and hence kill the job. As far the purpose of the code is concerned both the benchmarks offer same functionality. The two benchmarks can be run using different parameter files.

2.3 Bench_BSSN_PUGH

It is a Cactus [6] benchmark application of a numerical relativity code using finite differencing on a uniform grid. It uses the CactusEinstein infrastructure to evolve a vacuum spacetime, using the BSSN formulation of the Einstein equations. It employs finite differencing in space, an explicit time integration method, and relies on the driver PUGH for distributing grid functions over processors.PUGH is a mature and efficient memory management and communication driver in Cactus [6]. PUGH uses MPI for communication; it has been shown to scale up to thousands of processors on essentially all computing architectures that there are, from small notebooks to the worlds largest supercomputing installations. This benchmark comes in two sizes, 80l and 100l, which use about 400 MB and 800 MB of main memory per CPU, and consume approximately 512 GFlop and 508 GFlop, respectively.

There are two parameter files `Bench_BSSN_PUGH_80l.par` and `Bench_BSSN_PUGH_100l.par`. The one with 80l has 80 grid points per processor and performs 80 iterations and the other parameter file specifies 100 grid points per processor and it performs 40 time steps or iterations. These parameters can be modified according to the machine on which they are being run. However, to have a uniform benchmark run on all the machines used for comparision purposes these parameters remain fixed. On each machine the benchmark is recompiled as the same binary will not run on all the machines. The executable is run with the same parameter file so that the benchmark ends up performing similar operations. Now, the time it takes to finish the run is highly dependent on the machine architecture. The set of compiler settings is fixed for each machine and a standard set of options is used for each architecture.

2.4 Bench_Whisky_Carpet

This benchmark is relativistic hydrodynamics code and employs mesh refinement. This benchmark is similar to `Bench_BSSN_Carpet`, but it solves the relativistic Euler equations

in addition to the Einstein equations. This requires more memory and more floating point operations per grid point. The relativistic Euler equations are implemented in the Whisky code. You find more information on Whisky on its web pages. This benchmark comes in two sizes, 36l and 48l, which use about 400 MB and 800 MB of main memory per CPU, and consume approximately 265.9 GFlop and 312.8 GFlop, respectively.

2.5 Duration of Run

The problem size increases with the number of processor for both the benchmarks. We have decided to select the problem size based on the time it takes for the simulation. It was decided that 20 minutes would be a good time to run the simulation and hence the parameter files for both the benchmarks were tweaked by changing the resolution and number of iterations.

2.6 Problems with IPM Results

IPM: Integrated Performance Monitoring is a low over head tool developed by David Skinner at Lawrence Berkeley Lab, that was used for getting total floating point operations and load/store operations for each benchmark. The analysis of results from the IPM runs on an IBM Power5 machine reveal that there are some possible bugs in the software. IPM does not produce reliable results when the number of processors increases. This is evident from the fact that, as the number of processors increased the floating point operations performed by each processor decreased. It should be noted that the benchmarks have increasing problem sizes with increasing number of processors. This means that the floating point operations should increase along with the number of processors. When per processor floating point operations are considered, we see that there is a large disparity between the number of floating point operations performed as the number of processors increased. So, it can be concluded that

the IPM results when large number of processors are employed are not reliable and should not be considered for analysis. However, results from single processor runs using PM API, a library which has access to hardware counters on IBM Power5, have shown similar numbers as IPM. This is useful information as this proves that IPM is reliable with smaller number of processors.

Chapter 3

Phase 1: Analysis of Results

In order to understand the numbers projected in the plots presented in this chapter, it is essential to have some introduction to the three main architectures that are being compared. There are specific aspects of these architectures which are responsible for the performance of the benchmarks. It is interesting to note that the `Bench_BSSN_PUGH` benchmark performs well on IBM Power5 architecture when compared to the other architectures, the reasons for which are elaborated in this chapter. In the second phase we will be looking at the profiling results and they clearly show that most of the results here are architecture dependent. For example, the way cache memory is organized on the processor plays a significant part in the performance of a benchmark on that architecture.

3.1 IBM Power5 Architecture

The Power5 is an extension of the IBM Power4 architecture. It is a dual core chip capable of handling both single threaded and multi-threaded applications. There are some interesting differences between Power4 and Power5 which directly translate into performance difference. From the IBM pages the following differences can be listed:

- L3 cache was moved from the memory side of the fabric to the processor side. Whenever there is an L2 cache miss the processor can fetch the instruction or data from the offchip L3 cache. L3 is connected to the processor with a bus running at half the processor speed and hence is not a major bottleneck. So, L3 operates as a victim cache for L2, i.e. if there is an L2 cache miss the data or instructions are fetched from the large L3 cache.

- Size of L3 cache is increased from 32MB to 36MB and also L3 cache in the Power4 needed two chips when compared to one chip for Power5.

- The memory controller has been moved on chip as the L3 cache is moved off the chip. This makes memory access faster and reduces the latency and bandwidth to L3 cache and memory.

- Addition of an L3 cache directory for lookup, to the chip makes access to L3 cache extremely fast.

3.2 Intel Itanium2 Architecture

The Intel Itanium2 [5] processor according to Intel's technical documentation is neither a RISC nor a CISC based architecture, instead it is uses an Explicitly Parallel Instruction Computing (EPIC) instruction set. Also, it is expected to scale to thousands of processors and the EPIC design enables tighter coupling between the hardware and software.
The interesting thing about the Intel Intanium2 Instruction Set Architecture is that it uses instruction bundles which are three instructions handled at the same time. It can execute two instruction bundles in a clock cycle, which means it can execute six instructions per clock cycle as opposed to the eight instructions per clock cycle performed by the IBM Power5. Also, the EPIC design consists of a 6-wide, 8-stage deep pipeline. It has six integer and

six multimedia arithmetic and logic units, two extended precision floating point units, two single precision floating point units, two load and two store units, three branch units. Design of Itanium assumes that the compiler is in better position than the hardware scheduler to schedule instructions. Compiler uses static scheduling as it has extensive knowledge of the code, where as hardware scheduler uses dynamic scheduling and hence it has limited control over the order of execution.

The biggest disadvantage of Itanium processor is that an L2 cache miss can be very costly as it ends up stalling the process of scheduling as the hardware scheduler in the CPU is not in control. This dependence on compiler to performing scheduling is the main drawback as there is no out-of-order execution. The effect on performance due to this is quite evident in our benchmark results. `Bench_BSSN_PUGH` has a large number of load/store operations and hence performs poorly on this architecture. Also, according to [5] code inflation is a big problem with EPIC as there are a number of NOP (no operations) in the code generated to make it fixed length bundles. This method of fitting code into fixed size bundles is part of the Itanium's EPIC architecture. IBM Power5 fetches 8 instructions per machine cycle and has four FPU's and a large L3 cache. This is one of the reasons why the overall wall time for the benchmarks on the Power5 is lower than that of Itanium architectures.

3.3 Intel Xeon Architecture

The Intel Xeon processor is available in different types of chips, with speeds ranging from 2GHz to 3.20GHz. Tungsten (NCSA-1280 nodes) and Supermike (LSU-512 Nodes) are two machines that are based on this architecture. The clock speeds for both the machines are different, with processors on Tungsten clocking at 3.2GHz and those on Supermike clocking at 3.06GHz. The major difference between this architecture and an IBM Power5 is the position of L3 cache. L3 cache is present on chip in a Xeon and it is only 2MB, where as

on a Power5 it is off chip and it is as large as 36MB. This is the main reason why there is a performance difference between the two processors when it comes the Cactus [6] benchmarks which are computationally intensive. On the whole, Xeon processors are compact and have lower cache memory when compared to IBM Power5.

3.4 Details of Machines Used

To come up with a machine list that is diverse in the selection of architectures was easy as there were accounts on a lot of supercomputing institutes available via CCT researchers. So, almost all kinds of architectures were used for benchmarking. The results of the runs are documented and every run was repeated atleast thrice to get a consistent value for the wall time. There are two types of results that are studied in the project, viz. single processor performance and scaling of the benchmark with the number of processors. Not all machines have both the results. Some of them do not have the scaling numbers as they had too few processors or it was not possible to get the results from the queuing systems in the time this report was prepared. Intel and IBM based processors are the most commonly used among the above list. Hence, the focus for this project was mostly on the two types of architectures. Moreover, among the IBM based processors the Power5 based architecture is a the most commonly used one, and hence majority of the results that were analyzed for this project were obtained from the benchmark runs on that architecture.Table 3.1 shows different machines and their architectures:

3.5 Classification of Results

This section explains the type of results that were collected from the benchmarks. Cactus [6] benchmarks without using any external software like PAPI clocks or IPM, can provide the

Machine Name	Architecture	Institute	Number of Processors
Supermike	Intel Xeon(IA32)	LSU	1024
Pelican	IBM Power5	LSU	112
LONI	IBM Power5	LSU	112
Nemeaux	Apple G5	LSU	1
XWS4	Intel Xeon(IA32)	CCT	1
GreenGrass	Apple G5	CCT	1
Bassi	IBM Power5	NERSC	888
Tungsten	Intel Xeon(IA32)	NCSA	2560
Cobalt	SGI Altix (IA64)	NCSA	2048
Copper	IBM Power4	NCSA	350
Bigben	Cray XT3	PSC	2068
Lemieux	Alpha Server	PSC	750
BlueGene/L	IBM Power5	PSC	-

Table 3.1: Table showing different machines

wall time for simulation which is called the "gettimeofday" and the CPU time spent in the simulation which is called as"getrusage". The most useful result for our purpose is the wall time as it gives more information about the actual time that has transpired in running the benchmark and it is different from the time as measured by counting the machine cycles of the processor. Hence in our analysis we use the wall time, which is generally the norm in most simulations. When the PAPI counters and IPM are turned on the results are more detailed and can be used for tracing and profiling. There are two types of results that can be obtained for each machine:

- Single processor wall time to calculate the single processor performance
- Scaling of wall time with the number of processors to analyze the scaling properties of the system

3.5.1 Single Processor Performance Results

The results for the wall times for the four benchmarks are presented in this section. The machines used for running the benchmarks are listed in the table 3.1. Not all the benchmarks

were run on all the machines. Machines in different institutes with different architectures and resources are shown in figure 3.1. This plot can be used as a reference to the resources of the machines, when analyzing the results of benchmark runs. The results for single processor benchmark runs have been used to calculate the percentage utilization of the peak performance of CPU. There are a number of factors that make each machine independent, of these the following are significant:

- Architecture of the processor: Number of cores, type of interconnect between the processors if it is a multi-processor, available cache (it could be shared among multiple cores or each core has its own cache)
- Available memory per processor: could be shared or distributed
- Type of interconnect used between the nodes if it is a symmetric multi processor, topology, latency and bandwidth of the interconnect. Interconnect does not effect the single processor performance it only effects the scaling results
- Amount of storage for an IO intensive benchmark
- If the processors on a node are being used to run more than one user's jobs, the interconnect between the processors is being shared between the two jobs. This is another factor which could effect the results from the benchmark run. On the other hand if the node is completely used by a user then the effect can be discounted. On systems with dual processor nodes this is generally not an issue as both the processors are generally used by the same user.

Wall time is also known as "gettimeofday", as it is defined as the time it takes for the user to submit the job and get the results of the simulation back. It is measures the overall system performance or response time instead of just the CPU time. CPU Time on the other hand is also called "getrusage", as it is the actual time spent by the processor executing the

user's job. It does not involve any other delay in the system and hence it is a strict measure of the performance of the CPU. For this project we are trying to measure the overall system performance for a given benchmark and hence "wall time" is a good measure for analysis. CPU time is not considered here, moreover the values of CPU times are very close to those of Wall times.

In order to get a good estimate for the wall time each data point in figure 3.3 is taken from a pool of runs. Each run is performed thrice and the best case time is projected here. It shows the wall times for single processor runs of each of the four benchmarks on different machines. Some of the data points in figure 3.3 are empty as the simulations were not run for those values or the data obtained is incomplete. Also, as we get more results from the runs we will be updating them on to the Cactus [6] database. There is a lot of information that was collected for this project which is present in an internal web page [7] on the CCT website. Right now these web pages are password protected but, very soon they will be ported to the public domain. Figure 3.3 shows all the four benchmarks at the same time as it will give a clearer picture about their performance. For example, from this figure 3.3 it is clear that `Bench_BSSN_PUGH` with both the parameter files in general runs faster than `Bench_Whisky_Carpet`. There is another benchmark called `Bench_BSSN_Carpet` which also solve the same problem solved by `Bench_BSSN_PUGH` but uses mesh refinement.

It is important to note that even though massively parallel systems employ hundreds and even thousands of processors to complete a single job, the single processor performance utilization is very low. Each processor is capable of performing a certain fixed number of floating point operations in a second and this number depends on a variety of factors like, the number of floating Point Units in the processor, the number of instructions executed the processor in a machine cycle, the amount of L2 or L3 (If present) cache for each processor, etc. These factors effect the performance of the benchmark drastically as we can

see from figure 3.3.It shows the percentage of peak performance utilized by each benchmark for different machines. We can see that `Bench_BSSN_PUGH` benchmark with the two parameter files has higher percentage of utilization than `Bench_Whisky_Carpet` benchmark. The reason behind this disparity is the difference in the number of floating point operations performed by the benchmark. `Bench_BSSN_PUGH` has more floating point operations than `Bench_Whisky_Carpet` and hence utilizes the peak performance of a processor in a better way. Each processor has a fixed value for the number of floating point operations that can be performed in a second and utilization is dependent on the application being run.

It is desirable, for a benchmark designed to test the performance of a processor, to have more floating point operations. Number of floating point operations performed in the application is dependent on factors like the compiler settings (optimizations), architecture specific flags used during compilation which could reduce the number of floating point operations by tweaking the application, etc. It can be beneficial to turn off all the optmization on an application and then run it to see the raw floating point operations performed. This is not a very reliable number as it also depends on the Instruction Set Architecture of the processor and other architecture dependent features.For testing the scalability of the application on different architectures we have used the machines show in 3.2

3.5.2 Scaling Results for different machines

An application is said to be scalable if the execution time decreases as the number of processors on which it is running is increased. It is a good metric for a benchmark and it can be used to test the performance of the benchmark on a large computing resource. In general, scaling requires the run time for the benchmark to decrease as the number of processors used increases. Ideally it there should be an inverse relation between the wall time and the

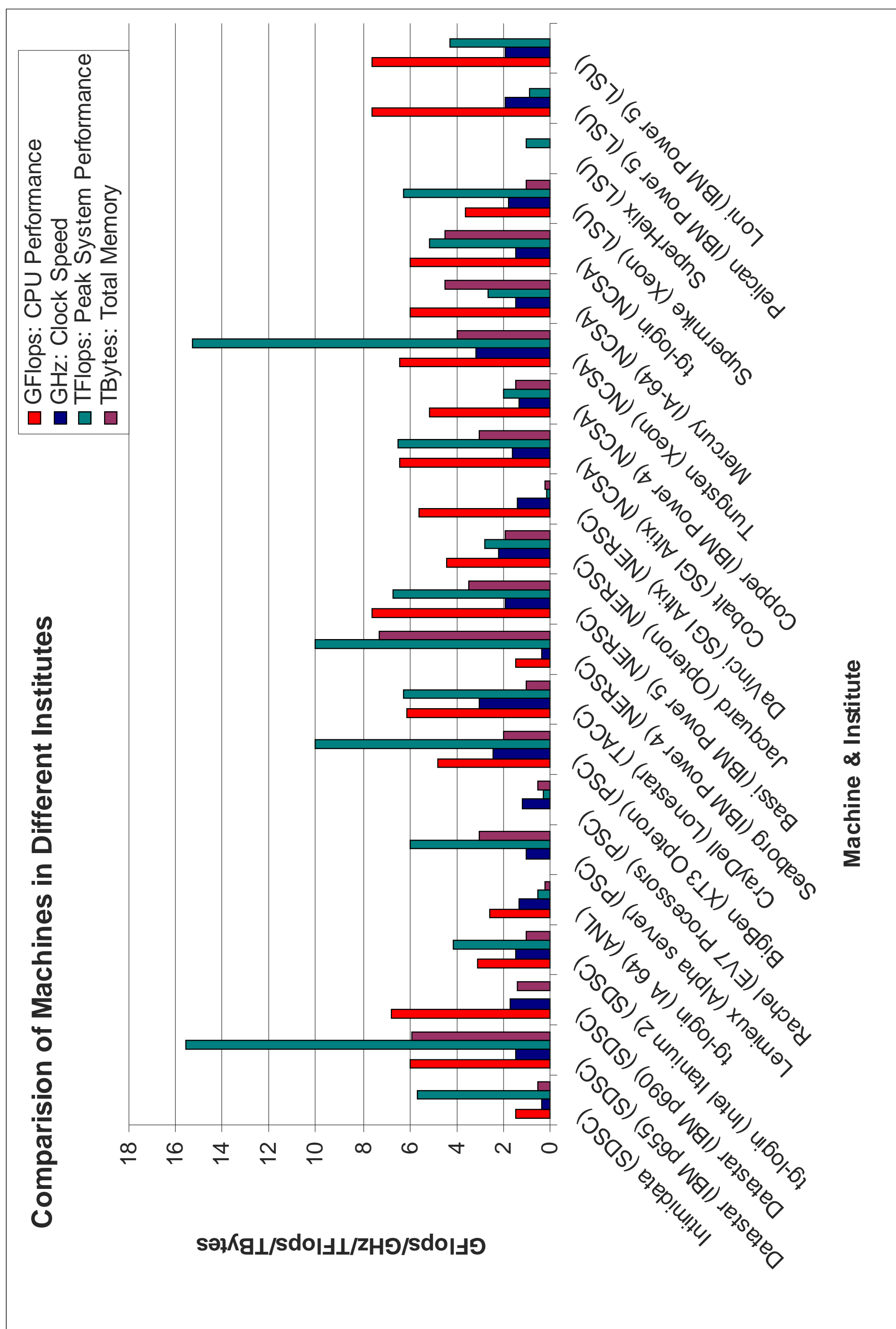

Figure 3.1: Comparison of machines from different institutes

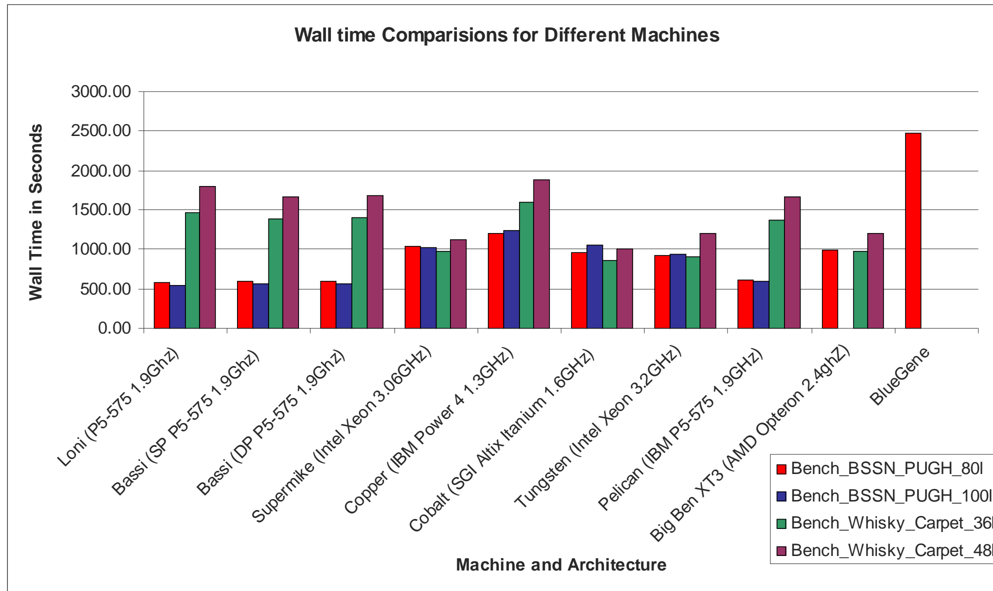

Figure 3.2: Single processor wall time results for all four benchmarks

	Bassi	**Loni**	**Copper**	**Tungsten**	**Cobalt**
Institute	NERSC	LSU	NCSA	NCSA	NCSA
Architecture	Power5	Power5	Power4	IA32	IA64
Processor	Power5	Power5	Power4	Intel Xeon	Itanium2
Clock Speed	1.9GHz	1.9GHz	1.3GHz	3.2GHz	1.6GHz
Proc Peak GFLops	7.6	7.6	5.2	6.4	6.4
Dual Core	Yes	Yes	Yes	Yes	Yes
L1 Cache	64I/32D	64I/32D	64I/32D	8KB	
L2 Cache	1.92MB	1.92MB	1.44MB	512KB	-
L2 Cache Shared	Yes	Yes	Yes	Yes	-
L3 Cache	36MB	36MB	128/8	1MB	9MB
Memory per SMP	32GB	32GB	64GB	3GB	2GB
Processor Count	888	112	350	2560	2048
Interconnect	GigE	GigE	GigE	Myrinet	Infiniband

Table 3.2: Machines used for scaling results

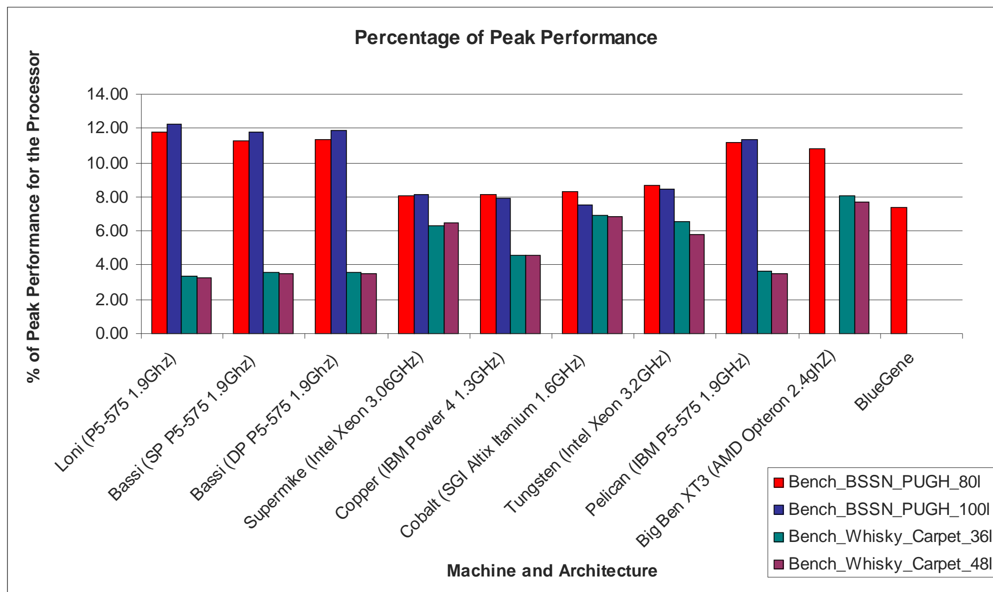

Figure 3.3: Percentage utilization of peak performance of processor for each benchmark

number of processors used. Some benchmarks required for NSF solicitation are known to scale extremely well. If the number of processors is doubled, the wall time for simulation reduces by a factor of two. This is almost perfect scaling and it is a very desirable quality of a benchmark, especially when it occurs for any number of processors. From the results obtained `Bench_BSSN_PUGH` scales very well on the other hand `Bench_Whisky_Carpet` does not scale with the number of processors.

Figure 3.4 shows the scaling of `Bench_BSSN_PUGH` with increasing number of processors. It can be seen from the graph that the benchmark scales differently on different machines. On IBM Power5 based architecture the benchmark scales very well, on the other hand the benchmark also scales on the Intel Xeon and Itanium 2 architectures. The same benchmark is run with a different parameter file and the scaling results are shown in figure 3.5. The benchmark again shows good scaling on all the machines and performs well on the IBM Power5 machines. As the number of processors increases, the time spent in sending messages between the processors increases drastically. It is also interesting to note that, after a certain point the performance shows degradation as the communication time dominates and there

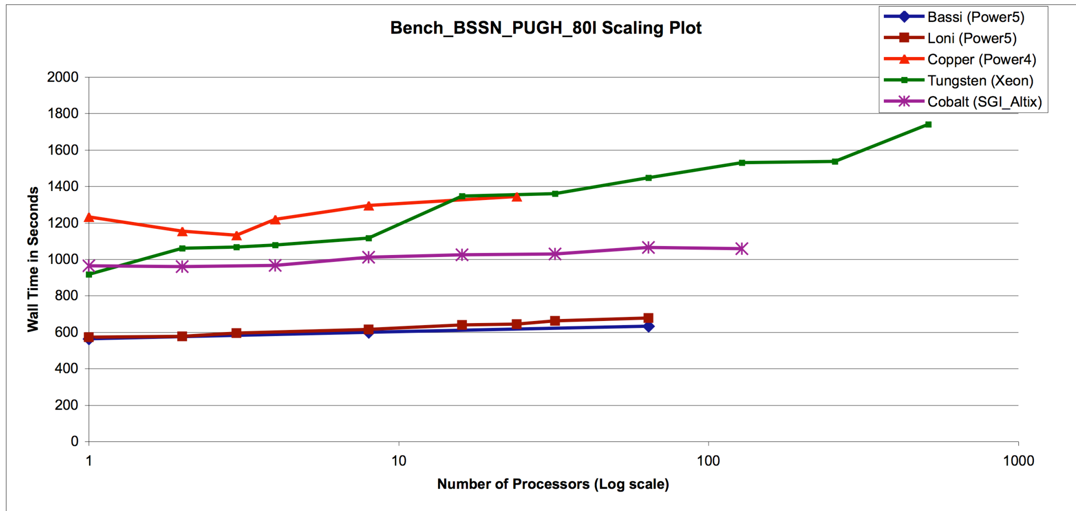

Figure 3.4: Scaling results for `Bench_BSSN_PUGH_80l` on different machines

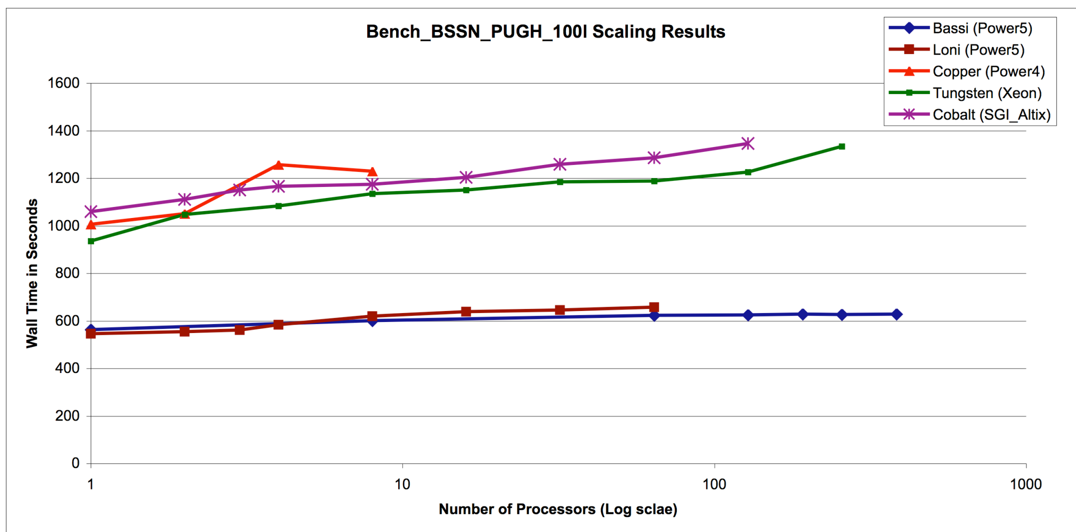

Figure 3.5: Scaling results for `Bench_BSSN_PUGH_100l` on different machines

are too many processors being employed to run the application. `Bench_BSSN_PUGH` scales well and as the number of processors increases the time for simulation does not increase drastically. For a perfectly scalable application, the curve will be a straight line, as the plots shown for scaling are semi logarithmic with the log scale on X-axis. The plot will be a flat line with negative slope as the time decreases with increasing number of processors.

In both figure 3.4 and figure 3.5 we can see that IBM Power5 architecture shows better numbers and the benchmarks scale well on this architecture. The scaling of an application depends not only on the way the code is written, but also on the architectural features of the entire machine. The interconnect plays a significant part in the scaling of an application. Most cluster systems or supercomputers can be of the following two types:

- SMP: Symmetric multiprocessor: Here each node has a certain number of processors sharing the same memory and each one of these nodes is connected to the others through an interconnect. So each node uses shared memory model and between the nodes there is message passing. Some nodes can have up to 32 processors and to test the scaling on these machines it is important to run the application on more than 32 processors so that more than one node is used and the interconnect starts playing a role. Even though Cactus [6] benchmarks use MPI within a node, in general it is better to get results by using more than one processor as it involves more communication. This could be a good test for the interconnect.

- Distributed memory architectures: These machines are individual processors connected by an interconnect, where each processor has its own memory and the processors use message passing to communicate. It is difficult to write code for a distributed memory machine as the programmer has to keep track of the location of data among different processors.

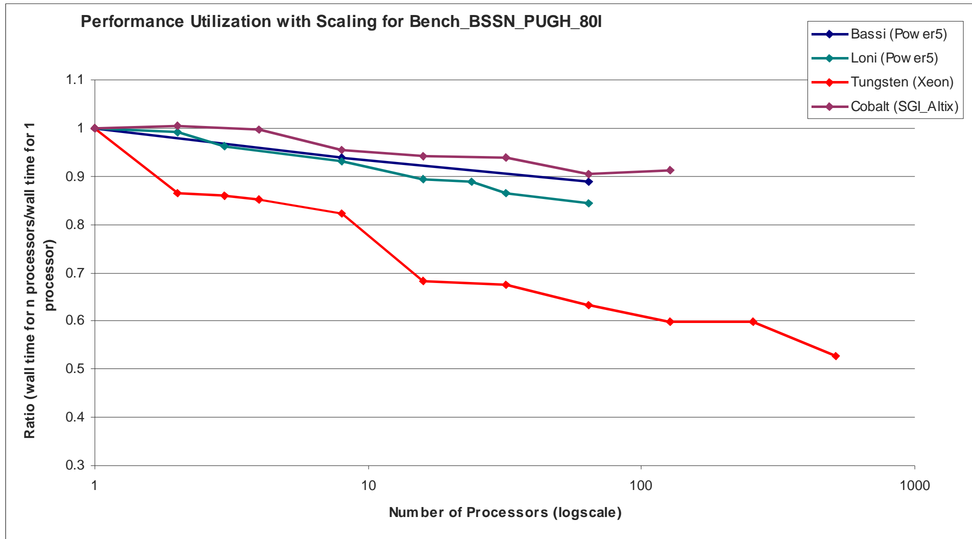

Figure 3.6: Scaling results for average utilization of performance for `Bench_BSSN_PUGH_80l` benchmark

The scaling results show that as the number of processors employed increases, the time taken for the simulation also increases. The reason is the increased communication time. This means that when one processor is used and the problem takes a certain amount of time, provided the benchmark is not scaled with the number of processors. In such a scenario, as the number of processors increases the time spent by each processor to solve the problem decreases as the work load gets distributed. Hence the scaling results are dependent on the benchmark and the way the code is written. From figure 4.1, it is clear that some of the benchmarks are computation intensive and some are communication intensive. As the time spent by each processor on the execution of code, those benchmarks which have a lot of communication show a degradation in performance as the processors spend more time communicating even though the overall computations per processor have reduced.

On all five machines we can clearly see that as the number of processors increases there is a steady decline in the utilization of the peak performance of each processor. These percentages are obtained as follows:

1. The total wall time taken for simulation is known from the runs
2. Total number of floating point operations in the benchmarks are already known
3. The total number of floating point operations performed per second by the benchmark can be obtained by dividing total floating number of floating point operations performed by the benchmark from item 2 by the total wall time for the run obtained from item 1. This number is the measured average performance obtained from the simulation
4. The exact value of the peak performance that can be obtained from the processor is known from the architecture
5. Item 3 divided by item 2 and the ratio multiplied by 100 gives the percentage of peak performance per processor
6. The above result is for single processor performance. If the scaling results are considered then the procedure changes slightly
7. As the number of processors increases the problem size increases and the number of floating point operations performed also increases.

The scaling numbers are also dependent on the type of compiler settings used and the nature of MPI used. It was noted that on Nemeaux an Apple G5 cluster located at LSU OpenMP produced better simulation times than MPICH. Cactus [6] benchmarks were run on 4 processors with OpenMP and MPICH and the results were quite conclusive as the OpenMP performed twice as fast as the MPICH.

As mentioned earlier the scaling results are a function of the performance of the interconnect. As the number of processors increases the interconnect performance becomes bottleneck and the wall time increases. The role of interconnect in scaling can be summarized as follows:

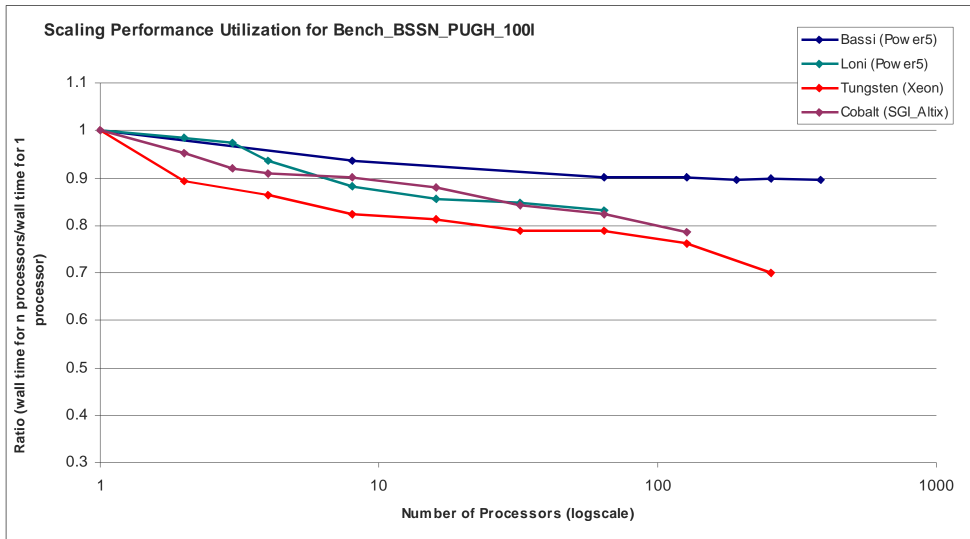

Figure 3.7: Scaling results for average percentage utilization of peak performance for `Bench_BSSN_PUGH_100l` benchmark

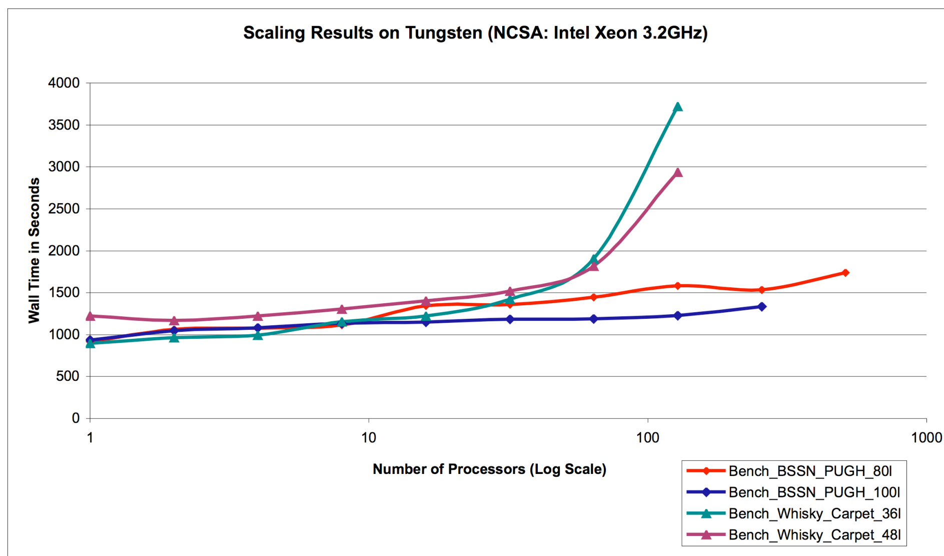

Figure 3.8: Scaling results of benchmark runs on Tungsten (NCSA: Intel Xeon 3.2GHz)

- Bandwidth of the interconnect is the measure of the maximum size of data that can be transfered on the interconnect in a second. As the number of processors increases the number of messages exchanged also increases and hence the requirement for bandwidth of the interconnect increases. So, it is good to have an interconnect with good bandwidth. There is a limit beyond which the bandwidth cannot be increased due to increasing cost in dollars. This means there will always be a certain number of processors beyond which there will be always be a performance degradation.

- Latency of the interconnect is the time taken for the data to reach from one processor to another. Even though it does not depend on the data size or the number of messages, it still has a great effect on the performance of machine when scaling is considered. As the number of messages increases the latency remains constant but the total time it takes for a processor to receive all the information required increases and this will add to the wall time of the simulation. Hence a desirable property of the interconnect is to have very low latency.

- There are many types of interconnects like Myrinet, Infiniband, GigEthernet, QsNet, Quadrics, High speed cross bar switch, proprietary interconnect. Depending on the system budget and requirements the interconnect is chosen. Some of them are scalable and others, some are expensive and some are not very expensive. So, the type of interconnect used has a significant impact on the scaling performance of a benchmark on a machine.

- The topology of interconnect is another factor which effects the scaling numbers. For example a cross bar switch provides excellent connectivity, but there it is very expensive and impractical as the number of processors increases. BlueGene uses a 3D Torus type of topology for its interconnect as it is know to scale extremely well.

Chapter 4

Phase 2: Profiling Bench_BSSN_PUGH

4.1 Reason for Profiling

During the analysis of the performance results of the NSF benchmarks used for the BigIron proposal, it was found that Cactus [6] benchmarks had more number of load and store operations when compared to the other benchmarks. In order to understand the reason for the large number of load/store operations of the benchmarks it is essential to identify those parts of the benchmark which are responsible for the load/store operations. According to the 80-20 rule 80% of time is spent in executing 20% of the code and 20% of time is spent executing 80% of code in a given program. This is a good indicator which shows that most of the program execution takes place in some critical portion of the code. In general this will be a nested loop which runs for a really large count and involves a lot of computation. It could also be the setup phase of the program. In the results we obtained from profiling, it was clear that almost a third of total time for execution is spent in the initialization phase. So, to improve the performance of the benchmark the setup phase can be broken down into smaller steps. This was one area where the benchmark could be tweaked for performance.

4.2 Tools used

A number of profiling tools were used to gather more information about the benchmark. There were tools which gave detailed timing analysis, other tools gave the cache misses at L1, L2 and L3 which directly translate into load/store operations, and finally some tools also checked for memory leaks in the program. The choice of using the right tool for this purpose was tricky and I had to use a set of tools to get a comprehensive report about the performance of the benchmark. The following set of tools were available to us:

- PAPI [1]: A PAPIClocks Thorn was already implemented in Cactus [6]
- Valgrind [4]: It contains tools like, cachegrind, memcheck, massif (Heap profiler)
- Valkyrie [10]: A GUI tool for Valgrind [4]
- gprof [8]: A GNU profiling tool
- Intel VTune Performance Analyzer [9]: Only works with Intel architectures
- Oprofile [3]: Works only on Linux kernels 2.2 and above

All the profiling tools mentioned above, except the Intel VTune Performance Analyzer [9] are open source and hence were freely available. The choice of a particular tool was entirely dependent on its features, as some tools had specific features which made them more useful than the others. Moreover, there are many features of a performance tool that effect its use for profiling an application. Some profiling tools are machine independent, others are operating system independent. However, there is a price that needs to be paid for being independent of the operating system and architecture. The tool which is independent does not give architecture specific information and hence can only be used to do timing analysis. On the other hand, some profilers take a lot of time and slow down the simulation as they are

required to count certain machine dependent events. In the sections that follow we explain the features and justification behind using each one of the tools.

4.2.1 PAPI: Performance Application Programming Interface

PAPI: Most of the profiling tools are plagued by their dependence on the operating system and the machine architecture and hence are not suitable to profile applications that are portable. PAPI [1] has the desirable features of being independent of operating system and architecture. It has been implemented to give the programmer an API that gives him/her access to a fixed set of hardware counters present on all architecture. Since, one of the aim's of this project is to profile Cactus [6] benchmark `Bench_BSSN_PUGH`, we used a thorn called PAPIClocks created by Thomas Schweizer from AEI. This thorn can be compiled with Cactus [6] and it will give the programmer access to the hardware counters provided by PAPI [1]. A parameter file which is used as input to a Cactus [6] executable can be used to specify the counters that have to be monitored. This means the user who is profiling the benchmark has to decide as to which counters are needed. It is a architecture dependent choice. If the simulation is being run on a machine with L3 cache, the counter which counts the L3 cache misses can be turned on. The results obtained from PAPI [1] runs are not exhaustive for the following reasons:

- PAPI [1] does not itself highlight the portions of code which are responsible for the cache misses. It gives a total number for the counter, which can be used to get a rough idea of the total number of cache misses.

- It gives the total number of instructions that were executed by the program, but not the portions of code which are responsible for the count.

Hence, PAPI [1] was used to collect the information about the total cache misses and the number of instructions and further analysis required using other profiling tools.

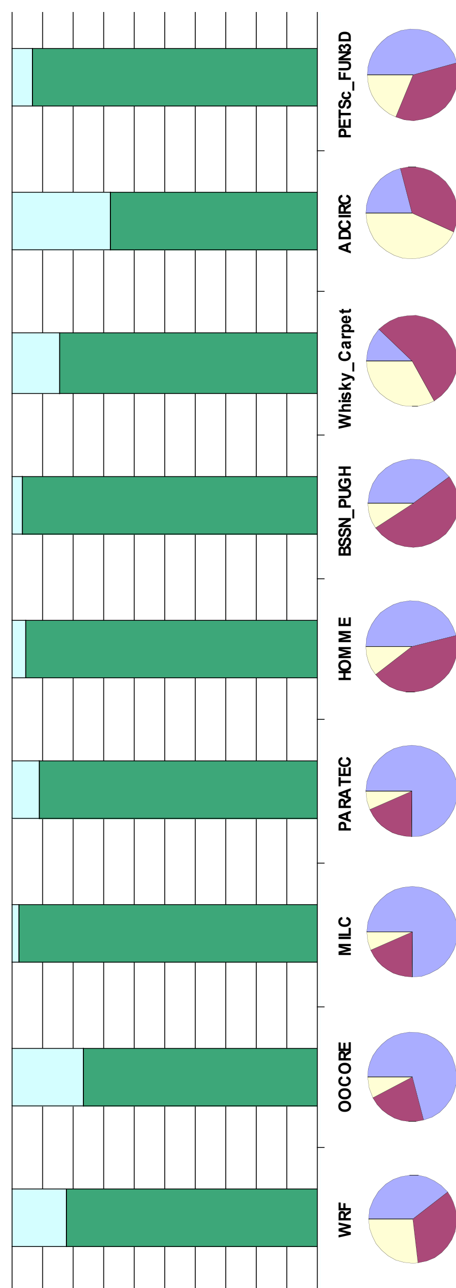

Figure 4.1: Comparison of different benchmarks. Legend: Blue-Flops, Maroon-load/store operations, Yellow-other operations.

4.2.2 Valgrind: A linux based profiling tool

Valgrind: It is a good performance profiling software as it has a set of tools which are very useful to get precise information about the benchmark. It can be used to get the cache misses caused by each function in the code, the memory leaks if any, and also it can do heap profiling to get more details about the memory management of the code under test. The following is a list of tools are available to the programmer:

- memcheck: A tool which detects memory leaks in the program. This tool was used to see if there were any memory leaks in the Cactus [6] Benchmark.
- cachegrind: An excellent tool used to find all the cache misses due to individual sections of the code. This tool was extensively used for this project. It was used with different problem sizes on different machines.
- addrcheck: Used to find memory leaks at a superficial level. Not used for this project.
- massif: Used for profiling the heap. Not used for this project.
- lackey: Used to create more tools for this software. Not used for this project.

The major disadvantage of Valgrind [4] is that it slows down the application to a crawl as there is a lot information that is being collected in the background. The simulations are known to take more than 10 times the normal time. The Cactus [6] benchmark `Bench_BSSN_PUGH` is expected to run for about 20-30 minutes on a normal system using a single processor. When Valgrindvalgrind was turned on to test for cache misses, the same benchmark ran for about 24 hours. This is quite normal due to the design and limitations of the Profiling tool. However, the information provided by this tool has been quite useful.

Moreover, the output produced by Valgrind [4] needs to be parsed and some sort of script should be used to extract the relevant information. I had to write a Perl script which parses

the output file and reports all the functions which contribute to more that 5% of cache misses. This number can be changed so that programmer has a control over the quantity of information that is being reported. For example, if 5% of total cache misses produces very little output, the number can be reduced in order to find more functions, which were responsible for cache misses. On the other hand if there are too many functions that are being reported, this number can be increased to find the most significant functions.

4.2.3 Valkyrie: A GUI based tool for Valgrind

Valkyrie: This is a GUI based front end for Valgrind [4] and all the advantages and disadvantages of Valgrind [4] are evident in this tool as well. This tool was not used for the project as the information retrieved would be redundant.

4.2.4 gprof: GNU Tool for profiling

Gprof: gprof [8] is a GNU tool for profiling the performance of an application that has been compiled with a GNU Compiler. It is dependent on gcc, gfortran and g77 for generating output. It can also generate the information about the call graph. In our case, the Cactus [6] Benchmark has a lot of code which is automatically generated. The call graph information would be cluttered with the automatically generated peal binding functions. This information is not really useful. Hence there were only a few runs with gprof [8]. Also, the major draw back for this tool is it dependent on the presence of GNU compilers on the system. Hence it is compiler specific and not portable to all systems. There could be ways to make it work on other platforms like AIX, but as we got good results from the cachegrind, we decided against using gprof [8] extensively.

4.2.5 VTune: Intel Performance Analyzer

VTune: This is a commercial software capable of retrieving comprehensive information from benchmark runs. It has several excellent features that can be used to get an exhaustive set of results for performance analysis of an application. This software was installed on Supermike and it is working there. It was not used for this project as the main objective of identifying trouble spots has been achieved. Hence, there was no need to use this software for further analysis. In the future, this software could be used to get more information about the benchmark once the optimization of the code starts. I may try to work with this software after the project to get more information about the Cactus [6] Benchmarks.

4.2.6 Oprofile: Linux based Profiling tool

Oprofile: The major drawback of Valgrind [4] was it became a huge bottle neck due to the time it takes to finish the simulation. Several test runs had to be created with smaller problem sizes in order to see if the profiler is producing useful information. On the other hand, if Oprofile [3] is installed on the system, it provides similar information very quickly as it is built into the kernel. It does not hold up the system as Valgrind [4] does. Oprofile [3] can be used to profile the entire system including the kernel. This is a very powerful tool as it is built into the kernel and has very low overhead and it can also profile the interrupt handlers. To use this tool the user needs to specify a set of counters that are to be turned on. There are a wide variety if counters that can be used ranging from the one's that count the Branch prediction errors to the ones that count the number of times the FSB has been accessed. So, the information generated by this tool is very specific and useful. The following are the major features of Oprofile [3]:

- Generates Call graph information
- Captures the performance behavior of the entire system including the kernel

- Very non intrusive and hence provides faster results
- Provides source annotation and instruction level profiles

There is a GUI tool which is bundled with the latest version of this software. It can be used to set the counters and the select the specific events that are monitored. The way this tool works is, it issues an interrupt whenever the counter crosses a minimum count. So, having a really low value for the count will create more samples, but may result in crashing the system as the number of interrupts will be very high. Hence a good balance must be maintained when choosing the counters and their minimum count values.

4.3 Results and Analysis

The results from the profiling runs are analyzed in this section. In the first section the results from gprof [8] are presented and in the second section results from valgrind [4] runs are presented and finally the results from oprifile [3] runs are shown.

4.3.1 Gprof Results:

Gprof produces two kinds of results, a flat profile and the call graph information about the application being profiled. Cactus [6] Benchmark `Bench_BSSN_PUGH` was compiled with gprof [8] and the results of the runs with the two par files `Bench_BSSN_PUGH_80l`, `Bench_BSSN_PUGH_100l` are presented in this section.

In the flat profile we can see the time spent in different sections of the code and this helps to isolate those portions of the code which are responsible for a lot of computation. Just like many other applications and programs, this benchmark also follows the 80-20 rule and spends more than 80% of its time in less than 20% of the code and vice versa. This is confirmed from the gprof [8] results using both the parameter files.

The flat profile consists of the following information:

- Percentage of time spent in the function
- Actual time spent in the function
- Number of calls made to the function, self and total
- Name of the function

On the other hand the call graph information contains the following:

- Number of times each profiled function is called
- Reports the number of recursive calls by each function
- Also gives the total time spent by the children of this function. By children, we mean all the functions which are called by this function.
- Name of the function

Some times the names of the parents for a function cannot be determined, in such cases gprof [8] call graph reports "spontaneous" as the name for the parent for this function. Also, the call graph information reports the presence of a cycle, and all the functions in the cycle.

Flat profile: Each sample counts as 0.01 seconds.

Table 4.2 shows the flat profile from the `Bench_BSSN_PUGH` run with the `Bench_BSSN_PUGH_100l.par`. The flat profile will produce similar results for every run provided the same parameter file is used. The reason being the time spent in a function is dependent on the code which is constant for each run. The call graph would also show similar results.

%time	self(Sec)	calls	self(Ks/call)	total(Ks/call)	Name
87.19	9530.79	1	9.53	9.53	adm_bssn_sources__
4.05	442.70	1	0.44	0.44	newrad_
2.14	234.03	-	-	-	MoL_RK3Add
1.84	201.59	240	0.00	0.00	adm_bssn_removetra__
1.29	141.29	242	0.00	0.00	adm_bssn_standardvariables__
1.07	116.86	240	0.00	0.00	MoL_InitialCopy
1.02	111.11	242	0.00	0.00	adm_bssn_shiftsource__
0.63	68.39	81	0.00	0.00	MoL_SetupIndexArrays

Table 4.1: gprof results

%time	self(Sec)	calls	self(Ks/call)	total(Ks/call)	Name
89.32	10223.07	120	0.09	0.09	adm_bssn_sources_
2.92	334.10	2640	0.00	0.00	newrad_
2.02	230.68	120	0.00	0.00	MoL_RK3Add
1.46	167.38	122	0.00	0.00	adm_bssn_removetra_
1.16	132.33	122	0.00	0.00	adm_bssn_standardvariables_
1.00	114.86	120	0.00	0.00	MoL_InitRHS
0.84	95.71	120	0.00	0.00	adm_bssn_shiftsource_
0.80	91.16	40	0.00	0.00	MoL_InitialCopy
0.19	22.11	120	0.00	0.00	adm_bssn_lapsesource_
0.12	13.37	1	0.01	0.01	adm_bssn_init_

Table 4.2: Flat profile results from gprof

4.3.2 Valgrind Results

Valgrind [4] has a range of tools out of which the most useful tools are the memcheck and the cachegrind. Memcheck finds the memory leaks in a program, and cachegrind counts the number of cache misses from running the program. The output generated by valgrind is very detailed as it produces output for the number of cache misses on a function to function basis. The main drawback of this tool is, it slows down the simulation to a crawl as it is collecting a lot of profiling information about the application. It is found that Oprofile produces similar output, but it does not slow down the simulation. The output of the Perl script is presented here. It takes the cachegrind output file as input and parses it for those functions which are responsible for the maximum number of cache misses.

```
Total_Instr = 1134560730685

L1_Instr_Rd_Misses = 407583356

L2_Instr_Rd_Misses = 4056114

Dt_Rds = 469688730466

L1_Dt_Rd_Misses = 67435193050

L2_Dt_Rd_Misses = 4664152671

Dt_Wrts = 111410992236

L1_Dt_Wrt_Misses = 5845650768

L2_Dt_Wrt_Misses1273853924


Total_Instr L1_Instr_Rd_Misses L2_Instr_Rd_Misses Dt_Rds

L1_Dt_Rd_Misses L2_Dt_Rd_Misses Dt_Wrts L1_Dt_Wrt_Misses L2_Dt_Wrt_Misses

fn=MoL_RK3Add

Percentage of Total Data Read Misses= 21.9557401361916
```

```
Level 2 Data Read Misses =1024049240
0 35840223280 6240 6240 11264066560 1028103400 1024049240 3072044080 6480 6240
fn=adm_bssn_sources_
Percentage of Total Data Read Misses= 49.5664023687358
Level 2 Data Read Misses =2311852680
0 966587729040 405394641 3398721 418926496320 61895223840 2311852680
92573709840 3480932640 288970800
fn=newrad_
Percentage of Total Data Read Misses= 10.2513436786255
Level 2 Data Read Misses =478138320
0 27420523680 163440 163200 8314996800 597969840 478138320 246818880
101271720 82897440
```

4.3.3 Oprofile Results

Unlike valgrind, Oprofile [3] does not slow down the simulation as it is built into the kernel. It can also profile the kernel and there is a special flag which can make it profile only a specific application. Only one counter is turned on at a given time with this profiler as these are hardware counters and Oprofile [3] is known to perform better with one counter at a time. In fact some of the combinations of the counters are not possible. There is a mask that can be set to collect specific events from the counter. The most important feature of Oprofile [3] is that it can collect information about the external libraries that were used by the application during the execution. Hence, we get a complete picture of the entire trace and the call graph information. The depth of the call graph can be specified in the configuration file. Oprofile [3] returns the number of samples collected for a given function. The more

Event	**Run1**	**Run2**
References	78.03e9	350.8e9
L1 Instr Misses	7.83e9	36.92e9
L2 Instr Misses	7142	116,703
L1 Instr Miss Rate(%)	10.3	10.52
L2 Instr Miss Rate(%)	0	0
Data References	57.106e9	259.33e9
Data Read Refs	46.89e9	213.17e9
Data Write Refs	10.215e9	46.16e9
L1 Data Misses	5.855e9	28.06e9
L1 Data Read Misses	5.088e9	24.27e9
L1 Data Write Misses	0.767e9	3.78e9
L2 Data Misses	859,435	801,459,700
L2 Data Read Misses	1829	475,356,853
L2 Data Write Misses	857,606	326,102,847
L1 Data Miss Rate(%)	10.2	10.8
L2 Data Miss Rate(%)	0	0
L2 References	13.69e9	64.99e9
L2 Read Refs	12.922e9	61.2e9
L2 Write Refs	0.767e9	3.78e9
L2 Misses	866,577	801,576,403
L2 Read Misses	8,971	475,473,556
L2 Write Misses	857,606	
L2 Miss Rate	0	0.1

Table 4.3: valgrind results using cachegrind on `Bench_BSSN_PUGH_100l`

samples	Percentage	symbol_name
223998	94.6481	adm_bssn_sources_
3496	1.4772	adm_bssn_standardvariables_
2876	1.2152	adm_bssn_removetra_
2380	1.0056	newrad_
1388	0.5865	MoL_RK3Add
852	0.3600	adm_bssn_init_
573	0.2421	adm_bssn_shiftsource_
380	0.1606	MoL_InitRHS
378	0.1597	MoL_InitialCopy

Table 4.4: Oprofile results

the number of samples collected the more time is spent in that function. The results shown below are obtained from turning the BSQ_CACHE_REFERENCE counter on and running the profiler. There are different ways to parse the output from the profiler. We can extract the call graph information and the detailed annotated source code and other useful results from the profiler output using opreport. A sample output from the run is shown below:

The following output is from the call graph information of the same oprofile run:

```
CPU: P4 / Xeon with 2 hyper-threads, speed 3391.89 MHz (estimated)
Counted BSQ_CACHE_REFERENCE events (cache references seen by the bus unit)
with a unit mask of 0x73f (multiple flags) count 100000
Samples on CPU 0
Samples on CPU 1
samples  %        samples  %        symbol name
-------------------------------------------------------------------------------
  1099967  100.000  1141195  100.000  CCTKi_BindingsFortranWrapperBSSN_MoL
1043897  94.8471  1080415  94.7265  adm_bssn_sources_
  1043897  99.0934  1080415  98.8971  adm_bssn_sources_ [self]
  8684      0.8243  10912     0.9988  adm_bssn_shiftsource_
```

```
  856        0.0813  1130       0.1034  adm_bssn_lapsesource_

  10       9.5e-04  5         4.6e-04  .plt

  1        9.5e-05  1         9.2e-05  SymBase_SymmetryTableHandleForGrid

  0              0  1         9.2e-05  util_tablegetintarray_

-------------------------------------------------------------------------------

  21087    100.000  22391    100.000  CCTKi_BindingsFortranWrapperBSSN_MoL

16507     1.4998  17478     1.5324  adm_bssn_removetra_

  16507    100.000  17478    100.000  adm_bssn_removetra_ [self]

-------------------------------------------------------------------------------
```

Chapter 5

Conclusion

The aim of the first phase of this project was to study the performance of Cactus benchmarks on different machines and architectures and understand the results. After getting the results from different runs, the second phase of the project required profiling the benchmark code and understand the reasons for its performance. From the results in the first phase, it can be concluded that the `Bench_BSSN_PUGH` benchmark performs well on a IBM Power5 architecture for the following reasons:

- Power5 has four FPU's instead of the two FPU's in Intel architectures
- Power5 picks up 8 instructions per machine cycle when compared to Intel architectures which pick up only 6 instructions per machine cycle
- Power5 has a large L3 cache
- `Bench_BSSN_PUGH` performs better than `Bench_Whisky_Carpet`

The trends in processor architecture suggests an emphasis on more cache, thread level parallelism and lower power consumption. So, the next generation of processors will have larger caches as we have seen in the case of Power5 which has more cache than Power4. thread

level parallelism makes the operating system believe that there are more than one processors by executing multiple threads on each core.

The EPIC design for Itanium processor has certain drawbacks which make it unsuitable for running applications that inherently have a large number of load/store operations. Also, its performance is severely limited due to the use of instruction bundles as this results in bloated assembly code with a lot of NOP's. There is no provision for out of order execution which is another deterrent to the performance of these applications on Itanium architecture.

For the second phase of the project a machine independent analysis was needed to understand the performance of the benchmark code. The overall objective of the project was to look into the performance of the Cactus Benchmarks and try to improve `Bench_BSSN_PUGH`, by reducing the cache misses there by number of load/store operations. The profiling results show that the benchmark actually spends a lot of time in some specific set of functions and also these functions are responsible for the cache misses during the run. Hence, the second objective of identifying the trouble areas of the code has been achieved as we have conclusive results which show the exact location of code which takes up most of the computation time.

Bibliography

[1] *http://icl.cs.utk.edu/papi/.*

[2] *http://ipm-hpc.sourceforge.net/.*

[3] *http://oprofile.sourceforge.net.*

[4] *http://valgrind.org/.*

[5] *http://www.anandtech.com/printarticle.aspx?i=2598.*

[6] *http://www.cactuscode.org.*

[7] *http://www.cct.lsu.edu/ smadiraju.*

[8] *http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html.*

[9] *http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm.*

[10] *http://www.open-works.co.uk/projects/valkyrie.html.*