NUMA effects on multicore, multi socket systems

Iakovos Panourgias

9/9/2011

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2011

Abstract

Modern multicore/multi socket systems show significant non-uniform memory access (NUMA) effects. In recent years, multi socket computational nodes are being used as the building blocks for most high performance computing (HPC) systems. This project investigates the effects of the NUMA architecture on performance.

Four micro benchmarks were developed and two kernel benchmarks from the NPB suite were used in order to quantify the NUMA effects on performance. These benchmarks were run on three HPC platforms. The micro benchmarks produced very detailed results regarding the low-level design of each platform, while the kernel benchmarks simulated the performance of a real application.

It was found that the location of memory on a multi socket platform could affect performance. In most cases, it is preferable to override the default processor binding. The use of simple command line tools or environment variables was enough to increase performance up to 50%.

Contents

List of Figures iv

List of Tables v

1. Introduction 1

2. Background and Literature Review 3

    2.1 Non Uniform Memory Access (NUMA) 3

    2.2 OpenMP 6

    2.3 Hardware 8

3. Benchmarks, requirements and implementation 13

    3.1 Benchmarks 13

        3.1.1 Benchmark 1 15

        3.1.2 Benchmark 2 16

        3.1.3 Benchmark 3 16

        3.1.4 Benchmark 4 17

    3.2 NPB OpenMP Benchmarks 17

        3.2.1 Kernel Benchmark CG 18

        3.2.2 Kernel Benchmark MG 18

    3.3 Requirements Analysis 18

        3.3.1 Requirements for the synthetic benchmarks 19

        3.3.2 High level overview of the synthetic benchmarks 21

        3.3.3 Implementation of the synthetic benchmarks 23

4. Results on Sun Fire X4600 platform 24

    4.1 Sun Fire X4600 Micro Benchmarks 25

    4.2 Sun Fire X4600 Applications 38

5. Results on Cray XE6 – AMD Magny-Cours platform 49

    5.1 Cray XE6 – AMD Magny Cours Micro Benchmarks 52

    5.2 Cray XE6 – AMD Magny Cours Applications 60

6. Results on Intel Nehalem E5504 platform 70

    6.1 Intel Xeon E5504 Micro Benchmarks 72

    6.2 Intel Xeon E5504 Applications 80

7. Conclusion 88

    7.1 X4600 89

    7.2 Magny-Cours 90

    7.3 Nehalem 92

7.4 Further Research 93
References 94
Appendix A – Benchmark Suite Software Design 97
Appendix B – Micro benchmarks usage 102

List of Figures

Figure 1 - The SMP Architecture 4
Figure 2 - The NUMA Architecture 5
Figure 3 - OpenMP illustration based on [13] 8
Figure 4 - MOESI state transitions, based on [16] 10
Figure 5 - Benchmark application flowchart 22
Figure 6 - Enhanced Twisted Ladder configuration, from [26] 24
Figure 7 - Sun Fire X4600 Local Read Bandwidth Measurements (32 MBs) 26
Figure 8 - Sun Fire X4600 Read Bandwidth Measurements (64 KBs) 27
Figure 9 - Sun Fire X4600 Read Bandwidth Measurements (variable sizes) 28
Figure 10 - Sun Fire X4600 Remote Read Bandwidth Measurements (32 MBs) 29
Figure 11 - Sun Fire X4600 Distributed/Local Read Bandwidth Comparison (32 MBs) 31
Figure 12 - Sun Fire X4600 Latency Measurements (lmbench3) 33
Figure 13 - Sun Fire X4600 Latency Measurements 34
Figure 14 - SUN X4600 Distributed Latency Measurements 35
Figure 15 - SUN X4600 Contention Measurements 37
Figure 16 - CG (class B) using 2 cores on X4600 39
Figure 17 - CG (class B) using 4 cores on X4600 40
Figure 18 - CG (class B) using 8 cores on X4600 41
Figure 19 - CG.B comparison on X4600 42
Figure 20 - CG.C comparison on X4600 42
Figure 21 - MG.B comparison on X4600 43
Figure 22 - CG (class B) run with numactl using four cores (two nodes) on X4600 45
Figure 23 - Modified CG.B on X4600 46
Figure 24 - A single twelve core AMD Magny Cours chip, from [32] 50
Figure 25 - A 24 core (node) AMD system, from [32] 51
Figure 26 - Magny-Cours Read Bandwidth Measurements (32 MBs) 53
Figure 27 - Distributed Read Bandwidth on Magny-Cours (32 MBs) 55
Figure 28 - Magny-Cours Latency Measurements 56
Figure 29 - Magny-Cours Distributed Latency Measurements 58
Figure 30 - Magny-Cours Contention Measurements 59
Figure 31 - CG (class B) using 2, 4 and 6 cores on Magny-Cours 61
Figure 32 - CG (class B) using 8, 12, 18 and 24 cores on Magny-Cours 63
Figure 33 - CG.B comparison on Magny-Cours 64
Figure 34 - CG.C comparison on Magny-Cours 64
Figure 35 - MG (class B) using 2, 4 and 6 cores on Magny-Cours 65
Figure 36 - MG (class B) using 8, 12, 18 and 24 cores on Magny-Cours 66
Figure 37 - MG.B comparison on Magny-Cours 67
Figure 38 - MG.C comparison on Magny-Cours 67
Figure 39 - Modified CG.C on Magny-Cours 68
Figure 40 - Nehalem 8 core platform, from [36] 71
Figure 41 - Read Bandwidth Measurements on E5504 (32 MBs) 73
Figure 42 - Distributed Read Bandwidth on E5504 (32 MBs) 74
Figure 43 - Nehalem Latency Measurements 76
Figure 44 - Nehalem Distributed Latency Measurements 77

Figure 45 - Nehalem Contention Measurements 79
Figure 46 - CG (class B) using 2,4, 6, 7 and 8 cores on Nehalem 81
Figure 47 - CG.B comparison on Nehalem 82
Figure 48 - CG.C comparison on Nehalem 82
Figure 49 - MG (class B) using 2, 4, 6, 7 and 8 cores on Nehalem 83
Figure 50 - MG.B comparison on Nehalem 84
Figure 51 - MG.C comparison on Nehalem 84
Figure 52 - Global Queue, from [39] 86
Figure 53 - Linux environment variables 102
Figure 54 - Linux execution 102
Figure 55 - CLE environment variables 102
Figure 56 - CLE execution 103

List of Tables

Table 1 - MESIF Cache States, based on [19] 11
Table 2 - Requirements table 20
Table 3 - Hops between processors in an eight Sun Fire X4600 server, from [26] 25
Table 4 - numastat results on X4600 47
Table 5 - numa_maps results on X4600 47
Table 6 - CG.C numastat average output 85
Table 7 - MG.C numastat average output 85
Table 8 - numa_maps results on Nehalem 87
Table 9 - Common functions 98

Acknowledgements

I would like to thank everyone who has helped in completing this research.

More specifically, I would like to thank Dr. Mark Bull for his support, his comments and his on-going motivation to look for answers. I would like to thank Dr. Judy Hardy for her support and understanding. Finally, I would like to thank my wife Maria for her encouragement.

1. Introduction

A typical high performance system uses hardware platforms with two or more sockets utilizing two or more cores per socket. The decision to move to multicore/multi socket systems was imposed by physical limits and not by the CPU hardware manufacturers. The speed of a CPU is limited by physical constants like the speed of light, the size of the atom and Planck's constant. It is also limited by the amount of heat that it emits, the electrical power that is consumes and the size of the silicon die. In order to keep up with Moore's law, which states that every 2 years the number of transistors in a CPU doubles [1], and marketing pressure CPU manufacturers had to introduce designs that employ multi socket/multicore chips. This design allowed CPU manufactures to achieve high performance without the power loads of increased clock speeds.

In order to benefit from the introduction of multi socket/multicore chips hardware manufacturers had to shift their designs. The new platforms feature integrated on-chip memory controllers (IMC). The IMCs are able to reduce memory latency and offer scalable memory bandwidth improvements. However, only memory controlled by the IMC demonstrates reduced memory latency. Furthermore, the use of point-to-point high bandwidth/low latency links provide cache-coherent inter-processor connections. The previous shifts in design paradigms are in fact the basis of the cache-coherent non-uniform memory access (ccNUMA) architecture. The NUMA architecture is the latest development in a long series of architectural designs, starting from a single socket/single core system to the latest multi socket/multi cores systems.

Software designers also followed the paradigm shift of the hardware manufactures. New software development models were created in order to use the new shared-memory platforms. Parallel programming paradigms like OpenMP and hybrid MPI-OpenMP have thrived during the past years, due to the explosive adoption of the NUMA systems. The ccNUMA architecture introduced the concept of shared resources. Resources like the Last Level of Cache (LLC) and the interconnect links between nodes are being shared by multiple cores.

This project investigates the NUMA effects on performance. We developed four micro benchmarks that measure memory bandwidth and latency in local and remote allocations. We use the results in order to understand the NUMA effects of each platform. Furthermore, two kernel benchmarks from the NPB suite are used in order to apply our theories on real applications. Three different platforms are used. Two are AMD based and one is Intel based. Both vendors use different designs in implementing a ccNUMA platform. Each design decision tries to minimise the effects that NUMA has on performance. Our testing uncovers that NUMA effects vary in strength amongst the tested platforms. Furthermore, we test the theory that local allocation is the best policy in a ccNUMA platform. Finally, we find some abnormal performance behaviour by one of the platforms and investigate the reasons behind a large and unexpected performance decline when executing a specific benchmark in another platform.

The results of all benchmarks are analysed and compared. The advantages and disadvantages of each platform are investigated. Strategies that increase performance are suggested and tested.

Chapter 2 provides a review of the current literature of multi socket/multicore NUMA platforms. It also presents the history and main characteristics of the OpenMP multiprocessing API. Finally, the hardware details of cache coherent NUMA platforms and details of the three hardware platforms by both vendors that will be investigated in this thesis are reviewed.

Chapter 3 describes the requirements, design and implementation of the synthetic benchmarks. A high-level overview is also presented. These benchmarks measure bandwidth and latency of a multicore/multi socket hardware platform and will be used to measure the effects of NUMA platforms to the performance of multiprocessing applications.

Chapter 4 presents the benchmark results of a Sun X4600 platform. The performance results are analysed and documented. The results show that the performance of the Sun X4600 hardware platform is affected by the location of the data in physical memory. Some strategies in order to avoid significant performance hits are proposed and tested. We also measure and test various memory and thread allocations in order to understand the NUMA effects on this platform.

Chapter 5 presents the benchmark results of the Magny-Cours platform. The performance results are analysed and documented. The results show that the Magny-Cours platform is not heavily affected by NUMA effects. However, the proper use of the Cray Linux Environment tools performance improvements of up to 47% for bandwidth bound benchmarks.

Chapter 6 reports the benchmark results of an Intel Nehalem platform. The performance results are analysed and documented. The results show that the two-chip Nehalem platform is not affected by many NUMA effects. However, the use of simple Linux tools can improve performance by up to 30% both latency and bandwidth controlled benchmarks. Furthermore, the reasons behind a thirteen fold decrease in performance occurs when we execute a NPB benchmark using a specific configuration.

Finally, Chapter 7 includes our comparison of the three platforms, presents our conclusions and includes some suggestions for future work.

2. Background and Literature Review

This chapter presents an overview of the existing literature regarding the evolution of multiprocessing platforms.

OpenMP is the defacto standard in scientific high performance multi-threading applications [2]. To ensure that the synthetic benchmarks are portable, the OpenMP standard will be used. The history of the standard and why it was selected as the building block for the benchmarks will be presented in section 2.2.

Furthermore, the hardware implementation details of cache coherent NUMA platforms will be presented in section 2.3. The use of HyperTransport and Quick Path Interconnect links will be explained along with the peculiarities of the design used by Intel and AMD platforms.

2.1 Non Uniform Memory Access (NUMA)

In order to understand the need for the NUMA architecture we need to present and describe the problems and limitations of other architectures. These architectures were designed to handle a lower number of processors or processors that can only handle certain tasks. As the number of processors increased these architectures began showing performance and scalability issues.

An architecture that is still in use is Symmetric Multiprocessing (SMP). A multiprocessor hardware platform is called an SMP system if the available memory is viewed as a single shared main memory. Many common consumer workstations use the SMP architecture, like the Athlon and Opteron processors from AMD and Core 2 and Xeon from Intel [3]. Systems using SMP access the shared memory using a common shared bus. The main disadvantage and bottleneck of scaling the SMP architecture is the bandwidth of the common bus. As the number of processors increases, demand of the common bus increases. According to Stallings [4] the limit of an SMP configuration is somewhere between 16 and 64 processors. The SMP design however offers an easy implementation and very easy task and process migration between the various processors.

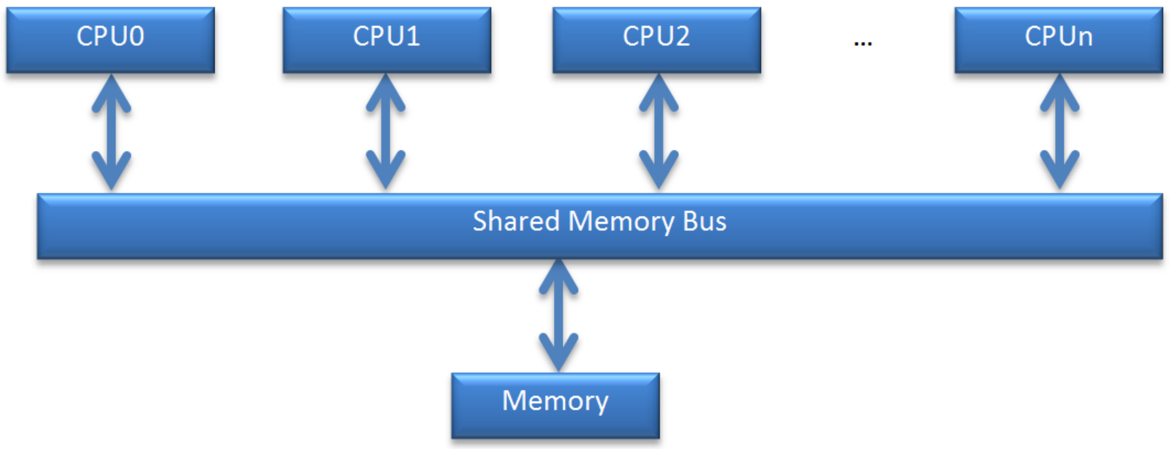

Figure 1 - The SMP Architecture

The SMP architecture offers performance increases in many situations, even when an application is written of a single processor system. The existence of more than one processor allows the Operating System to move power hungry applications to processors that are currently idle. Since moving threads and processes on an SMP system is easy and not very expensive, the OS is able to efficiently relocate processes.

Another architecture that has evolved considerably is Asymmetric Multiprocessing (AMP) [5], [6]. AMP was invented and thrived during the 1960-1970 computing era. In an AMP hardware platform, the OS allocates one (or more) processors for its exclusive use. The remaining processors can be used for executing other programs. The OS allocated processors are called boot processors. Furthermore, each processor could use a private memory and use a common shared memory [7]. This architecture allowed the addition of more than one processor on the high performance platforms of that time. It allows for an easy implementation, since processes/threads cannot migrate to other processors. However, the SMP design emerged as the winner between the two.

Currently high performance computing platforms increasingly use general purpose graphics processing units (GPGPUs) for computations. If we consider the GPGPU as a specialized CPU with a private memory, the AMP design can be applied to this new breed of high performance computing platforms that use both the CPU (to boot the machine, execute the OS code and distribute computation to the GPGPUs) and the GPGPU which actively takes part in all computations.

As we have previously shown both architectures offer advantages. SMP is very easy to implement and write applications as long as the number of processors is low. AMP can accommodate processing units that perform specialised tasks. However, the SMP architecture scales very badly when the number of processors increases. As many processors try to access different memory locations at the same time, the shared bus gets overloaded. Mitigation strategies using crossbar switches or on-chip mesh networks exist, however they make SMP implementation very difficult and the platform becomes very expensive to build.

As stated earlier the SMP design dictates that all memory access are using the common memory bus. Furthermore, the common memory space of the SMP design allows easy process/thread migration and data sharing amongst them. However, the speed of processing of each processor has increased faster than the speed of memory [8]. Therefore, computation times decrease whereas memory access (read/write) times increase. As processors try to access the main memory more frequently the shared nature of the memory bus creates bottlenecks. As more processors are added, the scalability of the system becomes problematic. Thus, a new architecture was required.

To overcome the scalability issues of the SMP architecture, the Non Uniform Memory Access (NUMA) was developed [9], [10]. As a remedy to the scalability issues of SMP, the NUMA architecture uses a distributed memory design. In a NUMA platform, each processor has access to all available memory, as in an SMP platform. However, each processor is directly connected to a portion of the available physical memory. In order access other parts of the physical memory, it needs to use a fast interconnection link. Accessing the directly connected physical memory is faster than accessing the remote memory. Therefore, the OS and applications must be written having in mind this specific behaviour that affects performance.

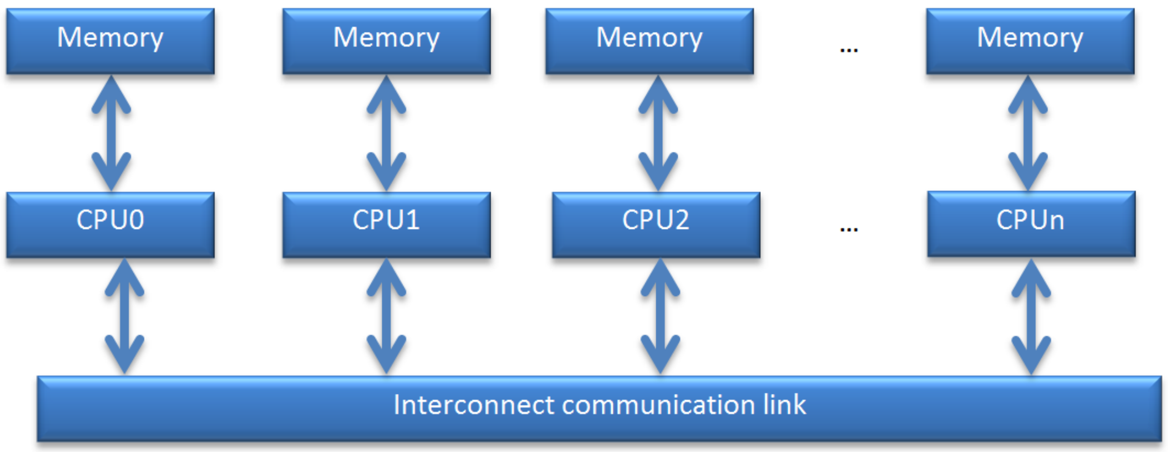

Figure 2 - The NUMA Architecture

Figure 2 illustrates a conceptual representation of a NUMA platform. Hardware implementations use direct point to point interconnects links between processors. Furthermore, each processor could have more than one physical core. Therefore, each CPU in the diagram can be a collection of cores that share the same cache memory and I/O devices. A node "is a region of memory in which every byte has the same distance from each CPU" [9]. If a CPU has four cores, then the four cores along with the memory that is directly attached to them and any other I/O devices are called a node.

The NUMA architecture offers performance advantages as the number of processors increases. The scalability issues of the SMP design are properly handled and memory access performs better. However in order to make use of the advantages, applications and operating systems must use the NUMA design and try to place data closer to the processor that is accessing them. This low level data management increases programming difficulty. Furthermore, it can be hard to analyse data access patterns on non-scientific applications. Even in scientific applications data access patterns can change during the runtime of an application, thus requiring relocation of the data in order to find a better, faster, allocation. However, operating systems (especially newer versions) are NUMA aware and they try to allocate memory close to the processor that handles a process/thread. If the process/thread is migrated to another processor that does not have direct access to the allocated memory, they try to migrate allocated memory closer to the processor in order to make full use of the NUMA performance effects.

The migration of processes/threads is the main task of the OS scheduler. The scheduler assigns processes/threads to available processors, taking into account some heuristics rules. A scheduler is required only for multitasking computing environment, however most modern operating systems are multitasking. A scheduler can be either cooperative or pre-emptive. Cooperative schedulers cannot interrupt programs, whereas pre-emptive schedulers can. The main goals of a scheduler are to maintain high throughput, low turnaround, low response time and fairness [11]. However, these goals often conflict. The scheduler must also take into account that moving processes/threads and allocated memory to another processor is time consuming. Therefore, it tries to minimize such reallocations.

Most schedulers use the first touch policy. If a process requests a memory chunk, the scheduler makes sure that the memory chunk is allocated from the nearest memory segment. The first touch policy is the easiest to implement. It is also the best solution for long running applications that are not migrated to another processor. If however the application is migrated to another processor that is located further away from the allocated memory, then each memory access will come at a cost. The cost will depend on the distance of the processor from the allocated memory segment. In order to reduce the costly memory operations, the scheduler can either relocate the process back to original processor or relocate the data in a different memory segment that lies close to the process. Since the option of relocating the process back to the original processor may not be always available, the scheduler might use several techniques to alleviate the cost of the memory access.

Affinity on next touch is one technique that schedulers use. When the OS scheduler senses that a process is accessing a memory chunk that is not located in the same node, it marks that memory chunk as ready to be moved. It will then try to find a suitable time to move the memory chunk to the appropriate memory segment, which is the one closest to the running process. According to Lankes et al. [12] the Sun/Oracle Solaris operating system uses the affinity-on-next-touch distribution policy, whereas Linux still lacks this functionality. However, Linux supports moving data pages amongst NUMA nodes, but not using an automated approach like the OS scheduler. Instead, the programmer must make the decision of moving specific memory pages. The OS however simplifies this task by providing programming ways of binding a process/thread to a processor. It also allows querying and retrieving the id of the current processor per process/thread. Thus, an application can take advantage of these APIs to move data pages.

2.2 OpenMP

This section will present the OpenMP standard. A summary of the standard, a brief historical review and a high level review of the main OpenMP clauses will be presented.

OpenMP is an API that enables the writing of multi-platform shared memory multiprocessing applications. OpenMP v 1.0 was introduced in October 1997 and it was only applicable for FORTRAN. The C/C++ version was released on October 1998. Since then the standard has evolved and the newest version is 3.1, released on July 2011. OpenMP is supported by almost all available operating systems, including Microsoft Windows, many flavours of Linux, Sun/Oracle Solaris and others.

OpenMP uses compiler directives, library routines, and environment variables that influence run-time behaviour [13]. Compiler directives are an easy and incremental approach of parallelising an application. OpenMP parallelisation is based on the notion of threads. The same notion can be found in POSIX threads and boost::threads. These two libraries provide a richer API for writing applications. They also provide fine-grained thread creation and destruction support. The extra functionality however comes with a higher difficulty in writing applications that use either POSIX threads of boost::threads. OpenMP abstracts the inherent difficulty of multiprocessing applications by exposing an easy to remember and use set of directives. However, OpenMP directives are usually converted to the most suitable form of thread functionality that each OS supports. OpenMP directives can be translated to POSIX threads calls. However, most modern compilers use the thread support that the underlying OS provides, in order to increase performance by using an API that is closer to the kernel.

An application written with OpenMP directives needs to be compiled using an OpenMP compatible compiler. Most compilers support OpenMP, including Visual Studio 2005 and newer, GNU GCC since version 4.2, Sun Studio since version 10.0 and the Portland Group compilers. A supported compiler modifies the code by replacing the OpenMP directives with actual function calls. The compiled application is also linked with the appropriate OpenMP runtime library that is loaded during execution of an OpenMP application.

When executed, an OpenMP application starts as a single threaded process. This thread is also called the master thread. The master thread is responsible for creating the other threads. This can happen when the application is run or when execution reaches the first parallel region. The appropriate OpenMP directives define a parallel region and the code is executed in parallel only in these regions. When the execution code reaches the end of the parallel region, then only the master thread executes the remaining serial code, whereas the other threads sleep or spin. The number of threads and the behaviour of the children threads when executing serial code can be defined by environmental variables or by programming functions..

OpenMP is used in scientific applications more often than in non-scientific ones. As mentioned above, the OpenMP API is smaller and more abstract than other APIs. Therefore, it is more suited to scientific applications where most of the runtime is spent in a specific part of the code. Usually that part is a loop construct. In fact, OpenMP has many special options for parallelising loops that other APIs lack support. Parallelisation of loops is reached by distributing the iterations of the loop to the participating threads. OpenMP allows the use of different distribution strategies in order to achieve better performance. Each participating thread executes the portion of the loop that was assigned, updates any shared variables and then returns control to the main thread. Other APIs provide greater control for sharing variables, however using OpenMP the creation of a number of threads and the distribution of workload is accomplished with fewer lines of code and far less complexity.

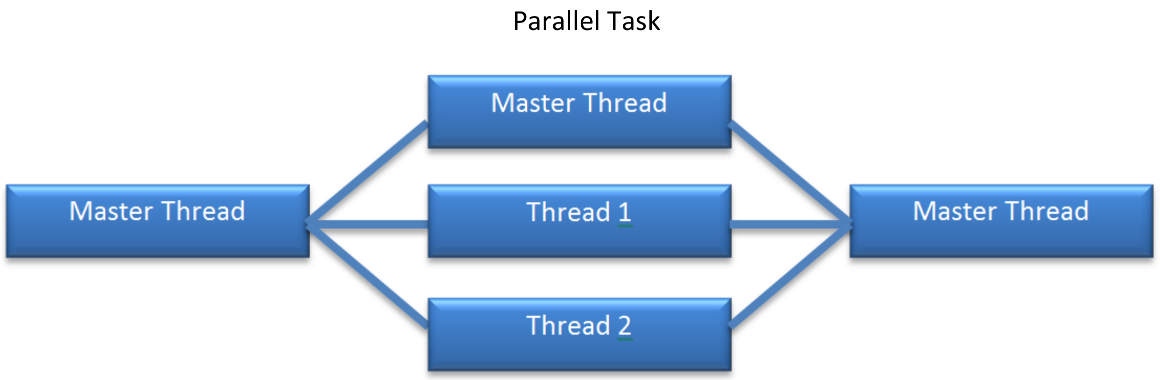

Figure 3 - OpenMP illustration based on [13]

In addition to the API that enables the creation of threads and of parallel regions, OpenMP provides support for synchronisation, initialisation of variables, data copying and reduction clauses. Synchronisation is used in all multithreaded APIs in order to avoid memory corruption by simultaneous or out of order copying/reading of shared variables. Synchronisation introduces performance hits for an application, however if it is needed then OpenMP supports atomic operations, critical sections and barriers that stop execution of the code until all threads have reached the barrier. Atomic operations advise the compiler to use the appropriate hardware instructions in order to reduce the performance hit. Critical sections only allow execution of the code for a single thread at a time, thus reducing the parallel section code to serial code. Initialisation of variables and data copying is used to populate private data variables. Private data variables are created with an undefined value. In order to initialise private data variables or to copy private values to global shared variables the clauses *firstprivate* and *lastprivate* are used respectively. Finally, OpenMP uses reduction clauses in order to accumulate local private values into a global shared variable. This is a very useful operation, which hides the complexity of reducing private variables into a single shared variable. Writing an efficient reduction code is a very complex and prone to errors task. However, OpenMP only allows the use of certain operators and does not allow one to write one's own implementation.

2.3 Hardware

This section presents the hardware implementations of NUMA. The role of caches, communication protocols, interconnects and cache coherence protocols are presented in this section. Finally, this section gives a short overview of the three main hardware platforms used in this project.

As mentioned earlier the NUMA architecture attempts to solve scalability and performance problems by separating the available memory; thus providing a portion of the available memory for each processor. Since each processor can access any part of the available memory, data that are on the shared memory are always up to date. However nearly all modern processors are equipped with a small (ranging from a few KBs to a couple of MBs) amount of very fast memory. This memory is not shared and it is known as cache. The physical location of the cache memory is very close to the processor in order to reduce communication costs. Furthermore, it is very fast which increases the production costs. Therefore, each processor/hardware manufacturer must take into account the production costs versus the performance increases of a larger or faster cache memory.

When the processor needs to access (read/write) a location from main memory, it first checks the cache. If that location is stored in the cache, then it uses the value of the cache. If the location is not stored in the cache, then it has to wait until the data are fetched from main memory. In order to improve throughput the memory controller reads data from the main memory of size equal to the cache line. To complicate further this behaviour the memory location may also be from a memory bank that is not directly attached to the processor. The processor then executes the appropriate instructions on the data from the cache. At some point in time, the data in the cache are evicted and are written to the main memory.

The NUMA architecture is mainly used in multi socket hardware platforms. Global memory is shared and each processor can access any data element, albeit with reduced performance if the data element is located on a memory bank that is not directly connected. However, the non-shared internal cache memory of each processor (or of each node for multi-node processors) is not accessible from other processors. A processor therefore does not know if the state of a data element in main memory is in a current state or if a more updated state exists in a cache of another processor. Thus, NUMA platforms use a mechanism to enforce cache coherence across the shared memory.

There are hardware platforms that use a non-cache-coherent NUMA approach. These systems are simpler to design and build since they lack the hardware features that are used to enforce cache coherence. However writing operating systems and applications for these systems is very complex and prone to errors. Software that runs on these systems must explicitly allocate and de-allocate data elements from the cache memory in order to maintain data consistency. It is therefore up to the programmer to ensure that all available data are in a current state.

Most NUMA systems are however cache coherent and they are formally called ccNUMA platforms. Writing operating systems and applications for ccNUMA is easier and is similar to writing applications for SMP systems. ccNUMA platforms use special hardware that enables inter-processor communication between the cache controllers. For every cache miss (read or write), every processor must communicate with all the others processors in order to make sure that the next fetch from main memory is in a current state and not stored in another cache. This communication has a significant overhead. In order to reduce this overhead we need to avoid access to the same memory area by multiple processors. Furthermore, NUMA platforms use specialised cache coherency protocols like MOESI and MESIF that try to reduce the amount of communication between the processors.

All current AMD processors that require cache coherency use the MOESI protocol. MOESI stands for Modified, Owned, Exclusive, Shared, Invalid and corresponds to the states of a cache line [14], [15], [16].

• **Invalid —** A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or in another processors' cache.

• **Exclusive** — A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.

• **Shared** — A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the owned state, then the copy in main memory is also the most recent.

• **Modified** — A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.

• **Owned** — A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state—all other processors must hold the data in the shared state.

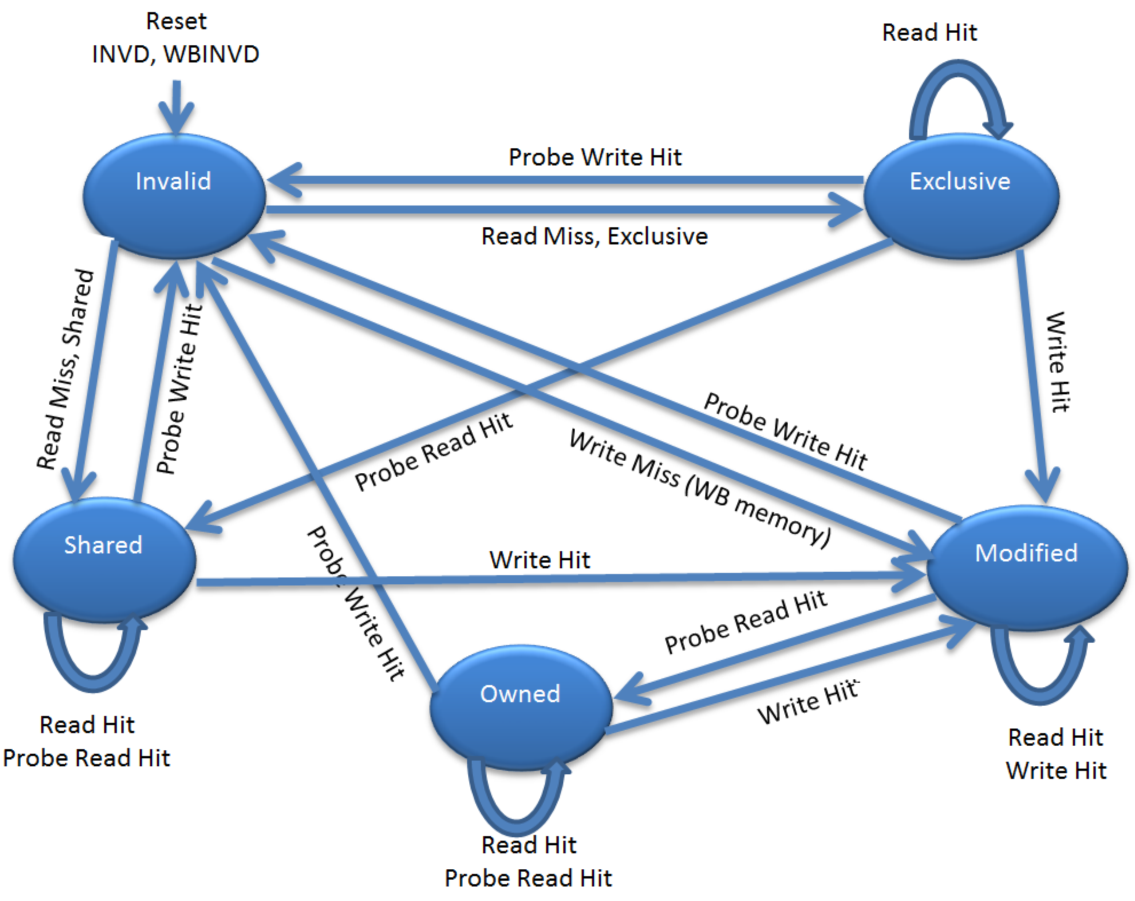

Figure 4 - MOESI state transitions, based on [16]

Figure 4 shows the general MOESI state transitions possible as described in the AMD Architecture Programmer’s Manual volume 2 System Programming [16]. The protocol uses read probes to indicate that the requesting processor is going to use the data for read purposes or write probes to indicate that the requesting processor is going to modify the data. Read probes originating by a processor that does not intend to cache the data, do not change the MOESI state. Furthermore, read hits do not cause a MOESI-state change and usually write hits cause a MOESI-state change.

AMD processors use HyperTransport (HT) links as a communication medium for ccNUMA implementation. Snoop messages are transported via the HT links. The snoop messages themselves are very small and contain the offsets for the address, pages and cache lines that the message is addressed for. However, the actual data transfer does not always take place via a HT link. If the two cores are part of the same processor chip, then they utilise the System Request Interface (SRI) path, which is considerably faster. This interface runs at the same speed as the processor. Therefore, transfers between the two caches take place very quickly. If a transfer needs to take place between two cores that are not in the same processor chip, then this transfer travels through an HT link. Even in this case the data are transferred from one cache to the other bypassing completely the main memory.

Xeon (Nehalem) Intel processors that require cache coherency use the MESIF protocol. MESIF stands for Modified, Exclusive, Shared, Invalid, Forward and corresponds to the states of a cache line [17], [18], [19].

- **Modified —** The cache line is present only in the current cache, and is dirty; it has been modified from the value in main memory. The cache is required to write the data back to main memory at some time in the future, before permitting any other read of the (no longer valid) main memory state. The write-back changes the line to the Exclusive state.
- **Exclusive —** The cache line is present only in the current cache, but is clean; it matches main memory. It may be changed to the Shared state at any time, in response to a read request. Alternatively, it may be changed to the Modified state when writing to it.
- **Shared —** Indicates that this cache line may be stored in other caches of the machine and is "clean"; it matches the main memory. The line may be discarded (changed to the Invalid state) at any time.
- **Invalid —** Indicates that this cache line is invalid.
- **Forward —** Indicates that a cache should act as a designated responder for any requests for the given line. The protocol ensures that, if any cache holds a line in the Shared state, exactly one (other) cache holds it in the Forward state. The Forward state is a specialized form of the Shared state.

Table 1 provides a summary of the different MESIF states.

State	Clean / Dirty	May Write?	May Forward?	May Transition To?
M – Modified	Dirty	Yes	Yes	-
E – Exclusive	Clean	Yes	Yes	MSIF
S – Shared	Clean	No	No	I
I – Invalid	-	No	No	-
F – Forward	Clean	No	Yes	SI

Table 1 - MESIF Cache States, based on [19]

Intel processors use the Intel QuickPath Interconnect (QPI) to transfer snoop messages. The logic of the MESIF protocol is implemented by the last level cache (LLC) coherence engine that is called Cbox. All QPI snoop messages are intercepted by the Cbox and are properly handled. The Nehalem processor contains eight instances of the Cbox, each managing 3 MB of the 24 MB LLC. The first four instances of Cbox (C0 – C3) are associated with the Sbox0, and the last four (C4 – C7) are associated with Sbox1. The Sbox is a caching agent proxy, which receives QPI messages from the QPI router (Rbox) and converts them to QPI snoops, data, and complete messages to the cores. It also take core requests and snoop responses and transmits them on the interconnect via the Rbox. As in the case of the AMD Opteron chips, if a transfer is requested and the originating and terminating core are both part of the same processor chip (both part of Sbox0 or Sbox1), the transfer takes place internally with no data leaving the processor. If the transfer is between Sbox0 and Sbox1 extra latency is added. Only when a data transfer between processor chips is requested, data travel through a QPI link.

As we have seen, both vendors use some means of reducing internode communication. Even though the communication size is very small, the communication frequency is very high, since it takes place for every cache miss (and sometimes for cache write hits as well). The terminology of the vendors differs, however they both use a proprietary interconnect which passes along small messages. The messages are addressed to the various processors and request the information regarding the state of the data that reside in the cache lines of each processor. Both vendors try to optimize the cost of communication by sending as much data as possible via internal routes which usually run at the same speed as the processor chip. When data needs to travel via an external route, data are grouped in order to keep HT communications as low as possible.

3. Benchmarks, requirements and implementation

This chapter provides a detailed description of each synthetic benchmark and the metrics and data that each benchmark will collect. It also presents the design requirements, a high level overview and implementation details of the synthetic benchmarks.

3.1 Benchmarks

This section introduces the benchmarks composing the benchmark suite. It will also present the rationale behind each benchmark, the data that will be collected, and the metrics that will be measured.

The benchmarks are designed to stress and test specific components of the hardware platform. The following benchmarks were created and used:

1. The data array is stored on the local memory of the first CPU. All threads/CPUs try to access the data elements. Memory accesses are issued in serial. This benchmark measures peak bandwidth by timing the duration of the read operations.
2. The data array is stored on the local memory of the first CPU. All threads/CPUs try to access the data elements. Memory accesses are issued in parallel. This benchmark measures bandwidth by timing the duration of the read operations, whilst many requests are issued in parallel.
3. The data array is stored on the local memory of the first CPU. A thread that executes on the same CPU tries to access the data elements. This benchmark measures local latency by timing the duration of memory accesses.
4. The data array is stored on the local memory of the first CPU. A thread that executes on a remote CPU tries to access the data elements. This benchmark measures remote latency by timing the duration of memory accesses.

In the following sections the rational and methodology behind each benchmark will be discussed in detail. However, before discussing each benchmark we need to define some of the following used terms.

All the benchmarks create an array of variable size. The variable size allows to experiment with the on-board cache memory of processors. By varying the size of the array we can make sure that the cache is fully utilized or that the size of the array greatly exceeds the available cache memory, thus forcing the processor to offload pages to main memory. Since cache utilisation plays a significant role in achieving optimum performance, the ability to test each individual cache level greatly improves the flexibility of the benchmarks.

The array is either stored on the local processor memory or it is distributed across the processors that take part in the experiment. The local or distributed allocation of the data array allows us to measure how it affects memory bandwidth and latency. We can then infer how NUMA affects performance.

We define as local processor memory the memory that is directly attached and controlled by that processor. The integrated memory controller (IMC) that all processors in a ccNUMA platform use controls memory. Memory attached and controlled by other processors is defined as remote memory.

The local processor is a misleading term however, since Linux uses a process/thread scheduler. It is very hard to know which processor (or core) actually executes the main process. If CPU0 is busy at the time that we login to the test system, then the scheduler will create our terminal session on another processor. Thus, the processor that actually executes the benchmark application cannot be guaranteed that is always the first processor. In order to avoid this uncertainty, we use the *taskset* command to force the processor on which the application is started. t*askset* is a command line executable that instructs the Linux kernel to execute an application from a specific processor. The equivalent command for the Solaris OS is *pbind*. Both commands interact with the process/thread scheduler of the kernel. The Linux scheduler supports this functionality since 2.6.25, whereas the Solaris scheduler since version 8.

Newer versions of the Linux OS also support the *numactl* command. The *numactl* command is part of the NUMA aware project, which introduced NUMA capabilities in the Linux kernel. The *numactl* command is able to bind and initialise an application on a specific set of cores (or physical processors) like *taskset* and it can also allocate memory on specific NUMA nodes. Therefore, it allows greater granularity and control of memory allocation and processor usage. The *numactl* command cannot be used in conjunction with the *taskset* command.

However, the Cray Linux Environment (CLE) that is used to power all Cray HPC systems does not support *taskset* or *numactl*. The CLE OS uses the command *aprun* to bind threads on processors. *aprun* is also used to allocate memory on specific NUMA nodes. Therefore, *aprun* is used as a replacement for *taskset* and *numactl*. It lacks some of the functionalities of *numactl*, however it is the only option if one wishes to access all nodes on a system powered by CLE.

The PGI OpenMP runtime libraries also provide environment variables that enable process/thread binding to processors. They can also bind individual threads to specific processors. The two environment variables are *MP_BIND* and *MP_BLIST*. In order to enable binding of processes/threads to a physical processor the *MP_BIND* environment variable must be set to "yes" or "y". In order to fine tune the binding the *MP_BLIST* environment variable can be used. By setting *MP_BLIST* to "3,2,1,0", the PGI OpenMP runtime libraries bind threads 0,1,2,3 to physical processors 3,2,1,0.

When the array is stored on the local processor (the local processor is the one that executes the master thread and is selected with the use of the *taskset* command and the *MP_BLIST* variable), the whole array is created and stored on that processor controlled memory. No OpenMP directives are used to parallelize the creation of the data array. Due to the first touch policy, the array will be allocated on the memory that is controlled by the processor that executes the thread. If not enough free memory is available, then the OS system will decide on which node it should allocate the remaining portions of the array. However, all these actions are invisible to the application, which only sees and accesses one contiguous memory chunk.

When the array is distributed, OpenMP directives are used to properly distribute the array, storing the data elements on the processors that are going to access them. By using the first touch policy we are able to allocate data elements on memory locations that are controlled by remote processors.

Finally, the memory accesses can be issued either in serial or in parallel.

Serial is defined as using OpenMP directives to regulate the concurrent number and order of threads that try to read/write. The concurrent number of threads is set to one and an increasing order is used (from 0 to the maximum number of threads). Serial results should be 100% reproducible as long as the test system is under the same load and the same starting CPU is used. The serial version of the benchmarks provides more concrete data and metrics that can be verified by others and by repeated measurements.

Parallel is defined as not using any OpenMP directives to regulate the concurrent number or order of threads that try to read/write. Thus, the OpenMP runtime is responsible for ordering and scheduling the threads. Parallel results may not be 100% replicable because the load of the system may vary and that the OpenMP runtime may not use deterministic algorithms to set the order of threads. However, the parallel version of the benchmarks simulates a normal environment more closely than the serial approach.

The following benchmarks will use the Linux command *taskset* and the PGI environment variables *MP_BIND* and *MP_BLIST* to control processor affinity. Each benchmark will use different values in order to set the desirable processor affinity.

3.1.1 Benchmark 1

> “The data array is stored on the local memory of the first CPU. All threads/CPUs try to access the data elements. Memory accesses are issued in serial. This benchmark measures peak bandwidth by timing the duration of the read operations.”

The first benchmark allocates all memory on the first processor. As noted earlier the first processor is not necessarily CPU0. The array is allocated in the local memory of the first processor by not using any OpenMP parallel directives. Then all threads read the data array, copying each data element to a temporary local location. This copy operation occurs in serial.

The purpose of this benchmark is to measure the memory bandwidth between processors in GB/sec. In order to avoid any unfair speedups, the processor caches are flushed after each test, by using an array which is far larger than the largest cache level available.

We use this benchmark to measure two different setups. The first configuration is used to measure local bandwidth. We therefore execute this benchmark using only one thread. We allocate the data array on memory that is controlled by the local processor and then try to read it. Using this method we expect to measure the bandwidth of each processor. We expect that the values that this benchmark returns are grouped together and that they will show very small variances.

The second configuration is to measure remote bandwidth. We therefore execute this benchmark using two threads. We allocate the array on memory that is controlled by the first processor and then we try to read it using the other thread that executes on a remote processor. Using this method we expect to measure remote bandwidth between the two processors. We expect to observe that as the distance of the processor and the location of the array increases, the bandwidth decreases.

3.1.2 Benchmark 2

> "The data array is stored on the local memory of the first CPU. All threads/CPUs try to access the data elements. Memory accesses are issued in parallel. This benchmark measures bandwidth by timing the duration of the read operations, whilst many requests are issued in parallel."

The second benchmark allocates all memory on the first processor. As noted earlier the first processor is not necessarily CPU0. The array is allocated in the local memory of the first processor by not using any OpenMP parallel directives. Then all threads read the data array, copying each data element to a temporary local location. This copy operation occurs in parallel.

The purpose of this benchmark is to measure the memory bandwidth between processors in GB/sec at saturation speeds. In order to avoid any unfair speedups, the tests are synchronised and the processor caches are flushed after each test, by using an array which is far larger than the largest cache level available. This benchmark tries to quantify contention effects amongst the interconnect links. It also tries to investigate how the integrated memory controllers and the cache coherence protocol react, when multiple near-simultaneous requests are posted.

This benchmark collects data and metrics simulating an application which allocates memory in a single location and then tries to access that memory from other locations at the same time. We should observe that the local memory accesses are faster than the remote. We should also observe that as the distance of the CPU and the location of the array increases, the bandwidth decreases.

3.1.3 Benchmark 3

> "The data array is stored on the local memory of the first CPU. A thread that executes on the same CPU tries to access the data elements. This benchmark measures latency by timing the duration of memory accesses."

The third benchmark allocates all memory on the first processor. As noted earlier the first processor is not necessarily CPU0. The linked list is allocated in the local memory of the first processor by not using any OpenMP parallel directives. The latency benchmark uses pointer chasing to calculate the latency for accesses to main memory and different cache levels.

The purpose of this benchmark is to measure the local memory latency in nanoseconds. In order to avoid any unfair speedups, the processor caches are flushed after each test, by using an array that is far larger than the largest cache level available.

We should observe that as the size of the linked list increases and consequently the number of cache misses increases, memory latency should also increase due to the extra communication needed to maintain cache coherency.

3.1.4 Benchmark 4

> “The data array is stored on the local memory of the first CPU. A thread that executes on a remote CPU tries to access the data elements. This benchmark measures remote latency by timing the duration of memory accesses.”

The fourth benchmark allocates all memory on the first processor. As noted earlier the first processor is not necessarily CPU0. The linked list is allocated in the local memory of the first processor by not using any OpenMP parallel directives. The latency benchmark uses pointer chasing to calculate latency for accesses to main memory and different cache levels. A thread that runs on another processor tries to follow the list of pointers.

The purpose of this benchmark is to measure the memory latency in nanoseconds, when a thread that is running on a remote processor tries to access the local data array. In order to avoid any unfair speedups, the processor caches are flushed after each test, by using an array that is far larger than the largest cache level available.

We should observe that as the size of the linked list increases and consequently the number of cache misses increases the memory latency should also increase due to the extra communication needed to maintain cache coherency.

3.2 NPB OpenMP Benchmarks

The NAS Performance Benchmarks (NPB) are an industry standard suite of platform independent performance benchmarks. They are used for measuring specific scientific problems on high performance platforms. The NASA Advanced Supercomputing (NAS) Division is responsible for development and maintenance. The suite is comprised of eleven benchmarks:

- CG – Conjugate Gradient
- MG – MultiGrid
- FT – Fast Fourier Transform
- IS – Integer Sort
- EP – Embarrasingly Parallel
- BT – Block Tridiagonal
- SP – Scalar Pentadiagonal
- LU – Lower-Upper symmetric Gauss-Seidel
- UA – Unstructured Adaptive
- DC – Data Cube Operator
- DT – Data Traffic

All of the benchmarks are trying to copy computations and data movements of Computational Fluid Dynamics (CFD) applications. Each benchmark tests different aspects of a high performance computing (HPC) platform. The benchmarks were specified as algorithmic problems and included only some reference implementations. The detail of the specification however was enough to enable the creation of an implementation. The CG and MG benchmarks are used to complement the micro benchmarks. The benchmarks use different class sizes in order to imitate small, medium and large data sets and computations. Class A is the smallest data set and is not used. Classes B and C are used to stress the memory subsystem of the available platforms. The differences of the B and C class are not only restricted to the number of iterations. The size of the problem is also modified. This causes the runtime of the benchmarks to increase considerably for the class C problems. However the extra iterations and larger memory usage help to highlight any NUMA effects.

3.2.1 Kernel Benchmark CG

The CG benchmark is a solver of an unstructured sparse linear system, using the conjugate gradient method. "This benchmark uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros" [20]. The number of iterations and data size are controlled by the class size of the problem. For a detailed description of the problem and solution, readers are advised to study "THE NAS PARALLEL BENCHMARKS" [20].

This kernel benchmark tests irregular memory access and communication latency. Therefore, it was used to simulate a real life scientific application that is latency bound.

3.2.2 Kernel Benchmark MG

The MG benchmark is a simple 3D multigrid benchmark. The benchmark computes a number of iterations in order to find an approximate solution u to the discrete Poison problem:

$$\nabla^2 u = \upsilon$$

on a 256 x 256 x 256 grid with periodic boundary conditions [20]. The algorithm works continuously on a set of grids that are made between coarse and fine. It tests both short and long distance data movement. [21] The number of iterations is controlled by the class size of the problem. For a detailed description of the problem and solution, readers are advised to study "THE NAS PARALLEL BENCHMARKS" [20].

This kernel benchmark tests both long-distance and short-distance communication and is memory bandwidth intensive. Therefore, it was used to simulate the behaviour of a real life scientific application that is bandwidth controlled. The compilation of the MG class C was not possible using the PGI compiler. It seems that this is a known issue of the compiler and a solution has not been provided yet. Therefore, we used the GNU GCC compiler to compile the class C problem. The use of a different compiler should not affect our results, since we compare time duration for each class problem and we do not compare speed up results between class problems.

3.3 Requirements Analysis

Requirement gathering and analysis is an integral part of every software engineering project. According to Spectrum IEEE the FBI's Virtual Case File software project was doomed, even before a single line of code was written [22]:

> *"Matthew Patton, a security expert working for SAIC, aired his objections to his supervisor in the fall of 2002. He then posted his concerns to a Web discussion board just before SAIC*

and the FBI agreed on a deeply flawed 800-page set of system requirements that doomed the project before a line of code was written."

There are other accounts of failed IT projects [23],[24] where numbers as high as 60%-80% of failures are attributed to sloppy or inadequate requirement gathering and analysis and the cost of failed technology projects can reach billions of pounds. The acclaimed online magazine ZDNet also reports that [23]:

"According to new research, success in 68% of technology projects is "improbable." Poor requirements analysis causes many of these failures, meaning projects are doomed right from the start."

Requirement gathering is the task of getting the requirements from the main stakeholders of the project. In the case of this thesis, the main stakeholders are also the users and developers. Therefore, the chance of misinterpreting a requirement is very small.

In a normal business environments requirement gathering includes tools like questionnaires, interviews, workshops and review of user actions. However, these tools are not required in our case, since the purpose of the benchmarks and the limits of the project are clearly defined.

Requirement analysis is the task of analysing the gathered requirements in order to find out if they are inaccurate or if there are any contradictions.

The final step of the requirements phase is to properly document the requirements. Current practises also document them using use-cases, UML diagrams and user stories. Since the purpose of benchmarks is clearly defined and minimal user interaction is required the use of extra tools like use cases and UML diagrams will not be employed. Instead, the requirements will be documented in this chapter as briefly as possible and will be explained in more detail in the following chapters.

3.3.1 Requirements for the synthetic benchmarks

The main purpose of the synthetic benchmarks is to measure and quantify the NUMA effects on performance. However, there are other functional and design requirements that must be met in order to deliver a suite of synthetic benchmarks that can be extended and expanded as easily as possible.

Due to the lack of other formal requirements the authors prepared the following list of requirements based on the required functionality of the synthetic benchmarks suite (Requirements table):

Requirement ID	Requirement Details	Priority	Paragraph
Main functionality			
100	Enhancing the code of the benchmarks should be an easy task.	2	A.3.100
110	The length (lines of code) for each benchmark should be limited.	2	A.3.110
120	The benchmarks should make use of common functions to	2	A.3.120

	reduce the length of code and to reduce errors.		
130	The user must be able to execute all benchmarks, a subset of the available benchmarks or a single benchmark.	1	A.3.130
140	The user should be able to check how many benchmarks are available.	2	A.3.140
141	The user should be able to display the descriptions of the available benchmarks.	2	A.3.141
Portability			
200	The benchmarks must use portable code.	1	A.3.200
210	The benchmark source files must have some comments which explain the purpose and the output of the benchmark.	1	A.3.210
Usability			
300	The benchmarks must measure and quantify the NUMA effects.	1	A.3.300
310	At least one benchmark for latency/bandwidth must be written.	1	A.3.310
Compilation			
400	The benchmarks should be compiled using a central mechanism (Makefile).	2	A.3.400
410	The user should be able to compile a single benchmark or compile all benchmarks.	2	A.3.410
411	The compilation process could only compile changed files and ignore unchanged files.	3	A.3.411
420	An automake functionality could be used in order to check the existence of required libraries and compilers, before compilation starts.	3	A.3.420
Output			
500	The benchmarks must output the results in a properly formatted text file.	1	A.3.500
510	If more than one benchmark is being executed, then the output file should be written to disk as soon as possible in order to avoid any data losses.	2	A.3.510
520	The format of the output file could be user configurable.	3	A.3.520

Table 2 - Requirements table

The priority numbers used in the - Requirements table are categorised as:

- 1 – Critical requirements. These requirements are essential for the success of the project and must be implemented. The basic functionality of the benchmarks and other critical aspects of the project are defined with priority 1.
- 2 – Major requirements. These requirements are not essential; however they will improve the quality/reliability of the final project and should be implemented. The portability of the code and other user friendly aspects of the project are defined with priority 2.
- 3 – Optional requirements. These requirements are not major; and should be implemented only if there is enough time left. Configuration options are defined with priority 3.

3.3.2 High level overview of the synthetic benchmarks

In order to facilitate future work the benchmark suite is comprised of a single executable that dynamically loads the benchmark libraries.

The benchmarks are implemented as shared libraries. The libraries contain the actual code for the benchmarks. Each library implements a single benchmark. Common functions are provided in the files *commonHeader.h*, *benchmark.h* and *defines.h*. Each library must include a function called *run()*. The function *run()* is the entry point of the benchmark. If this function is missing then the main application program will terminate with an error. The initialization, allocation and actual benchmark code are implemented using different functions which are called from *run()*.

The executable performs basic initializations steps, checks the command line arguments and calls the libraries which implement the various benchmarks. The executable scans the local directory for libraries whose filename starts with "libtest_". If such files exist then it uses *dlopen* [25] to dynamically load the library and it tries to find the address of the symbol *run*. If the symbol is found then the function *run()* is executed, otherwise the program exits. The main executable permits the user to create white and black lists of benchmarks using command line arguments. Thus, the user can execute only some benchmarks or execute all benchmarks excluding some. The main application also provides a list function that is used to print a text description of each benchmark.

The dlopen mechanism lets the application to compile without the need to provide complete source code or compiled libraries for the benchmarks. The application does not include a hard coded list of the benchmarks; instead, it tries to find the benchmarks during runtime. Thus, development of new benchmarks can continue in parallel with the usage of existing benchmarks, without the use of commenting code or other informal methods.

In order to test the benchmarks as a standalone application each benchmark library is statically linked to an executable via the provided *Makefile*. Thus, a benchmark can be tested before being used by the benchmark suite.

A flow diagram of the benchmark process is shown in Figure 5.

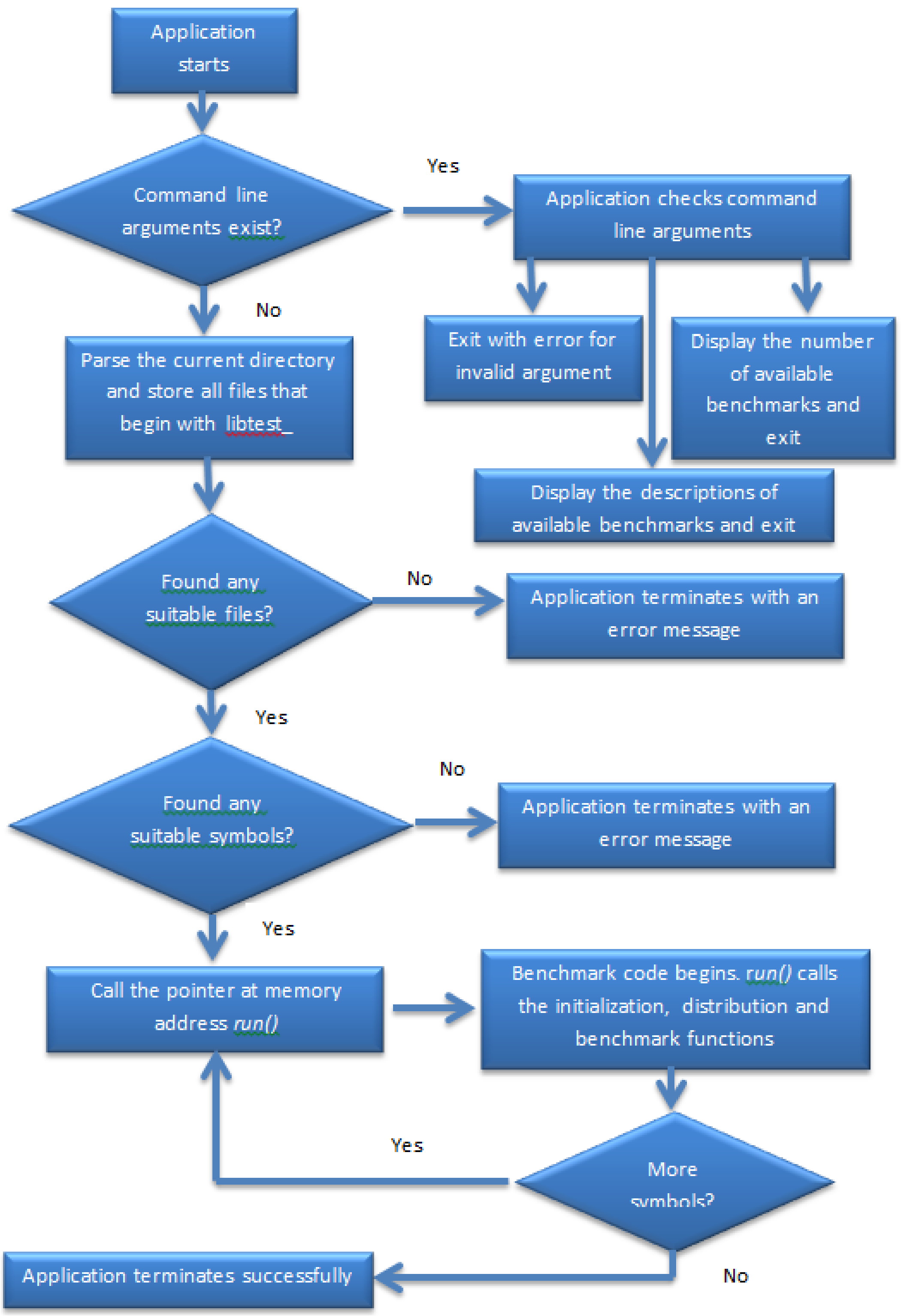
Application starts
Command line arguments exist?
Yes
Application checks command line arguments
Exit with error for invalid argument
Display the number of available benchmarks and exit
Display the descriptions of available benchmarks and exit
No
Parse the current directory and store all files that begin with libtest_
Found any suitable files?
No
Application terminates with an error message
Yes
Found any suitable symbols?
No
Application terminates with an error message
Yes
Call the pointer at memory address run()
Benchmark code begins. run() calls the initialization, distribution and benchmark functions
More symbols?
Yes
No
Application terminates successfully

Figure 5 - Benchmark application flowchart

3.3.3 Implementation of the synthetic benchmarks

The implementation of the synthetic benchmarks is based on the design decisions of the previous sections. For a detailed analysis of each requirement, please refer to Appendix A.

4. Results on Sun Fire X4600 platform

This chapter describes the experiments and analyses the results of the Sun Fire X4600 M2 hardware platform. The X4600 platform consists of eight sockets, each with a dual core AMD Opteron Santa Rosa processor (8218), with 16 available cores [26]. Each processor has 4 GBs of 667 MHz, with a single DDR2 memory channel per core. Total memory of the platform is 32 GBs. Peak bandwidth is 10.7 GB/sec.

Each AMD Opteron core has two dedicated levels of cache. Level 1 cache consists of a 64 KB 2-way associative instruction cache and a 64 KB 2-way associative data cache. Level 2 cache is a 1 MB exclusive 16-way associative cache. The cache lines are 64 bytes wide and are common for both L1 and L2 caches. The Santa Rosa processor uses the level 2 cache as a victim cache. A victim cache is used to store data elements that were replaced from the level 1 cache. It lies between level 1 cache and main memory and it tries to reduce the number of conflict misses [27]. The processor chip also has an integrated dual-channel DDR2 SDRAM memory controller, which controls the main memory that is directly attached to the processor.

As stated earlier AMD processors use HyperTransport links as a communication medium for ccNUMA implementation. The Santa Rosa processor has 3 16-bit links, clocked at 1 GHz and capable of 8 GB/sec bandwidth per link.

Each processor is connected via HT links with the others in an enhanced twisted ladder configuration [26]. The enhanced twisted ladder configuration is used in order to minimise hop counts and to reduce latency between the processors.

The enhanced twisted ladder configuration is shown in Figure 6.

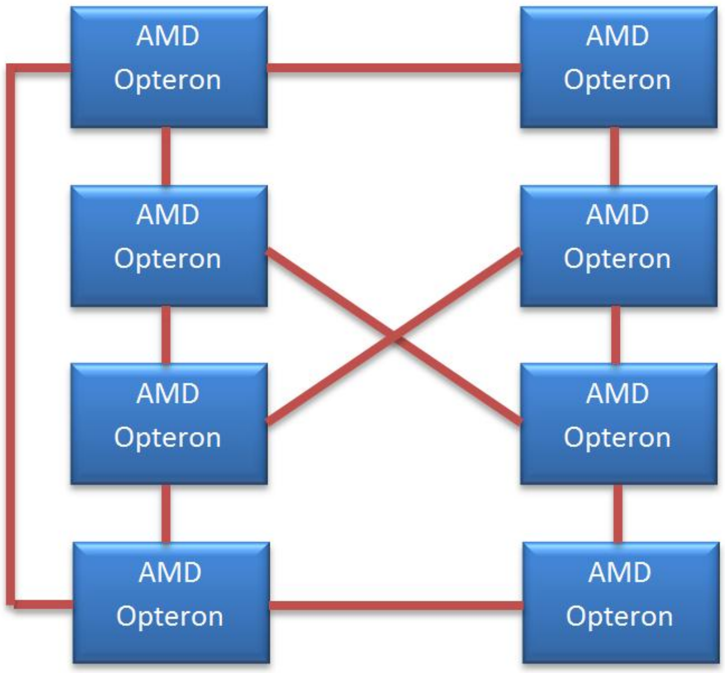

Figure 6 - Enhanced Twisted Ladder configuration, from [26]

Using this configuration, the required hops in an eight processors configuration are shown in Table 3.

CPU	0	1	2	3	4	5	6	7
0	*0*	1	1	2	2	2	2	**3**
1	1	*0*	2	1	2	2	1	2
2	1	2	*0*	2	1	1	2	2
3	2	1	2	*0*	1	1	2	2
4	2	2	1	1	*0*	2	1	2
5	2	2	1	1	2	*0*	2	1
6	2	1	2	2	1	2	*0*	1
7	**3**	2	2	2	2	1	1	*0*

Table 3 - Hops between processors in an eight Sun Fire X4600 server, from [26]

The top right and bottom right nodes need to access the I/O interfaces thus two HT links are used for peripheral communication. Thus, the maximum number of hops is three. This configuration creates an inequality, as the top right and bottom right nodes need an extra hop. However, as the benchmarks tests show some hops affect performance more than others.

4.1 Sun Fire X4600 Micro Benchmarks

The benchmarks described in Chapter 3 were run on the two available Sun Fire X4600 platforms. Even though the two platforms are identical, the results varied. Small variations are expected when micro benchmarks are executed on platforms that are not specifically designed for high performance computing tasks. The X4600 platform uses a vanilla Linux OS, which is not designed for long duration high performance computing applications. Therefore, simple OS functionalities (like disk bookkeeping, swap file reorganisation, signal and interrupt handling, etc.) have an impact on the results that the micro benchmarks report. As stated earlier, each benchmark is executed five times and the average value is used. This approach reduces the impact of obvious outliers. For simplicity and readability purposes, the results of a single platform are presented.

The micro benchmarks were executed using the *taskset* or the *numactl* commands. As mentioned earlier the *taskset* or *numactl* commands are used to set the processor affinity of the executed application. The Linux OS uses the first touch policy memory allocation. The first touch policy, as discussed earlier, allocates memory on the processor on which the application or thread is running. By using the *taskset* or *numactl* command to set the processor affinity, we are also setting the memory locality. The *numactl* command however also allows the binding of an application to a specific node and the allocation of memory on another node.

The first metric to be investigated is the read memory bandwidth. This metric will provide us with the first data of any performance differences of the NUMA nodes. The Sun platform has eight NUMA regions, the most compared to the other platforms. By measuring the read memory bandwidth we exclude any differences in read/write memory speed. Furthermore, by not modifying the array we eliminate the performance impacts of the MOESI protocol. An array is allocated on a memory segment controlled by a processor and a thread that runs on the same processor tries to access all the elements. The accessed element is copied in a temporary variable. In order to avoid the elimination of the temporary variable by the compiler, the variable is used in an *if*-test after the end of the loop. The duration of the loop is recorded and it is divided by the amount of copied data. Thus, a MB/sec value is calculated. Compiler and processor optimizations will affect the runtime; however since these optimizations will take place on all runs, the final result will not be affected.

The read memory bandwidth value indicates how fast each processor can retrieve data from memory. On a uniform memory platform, this value is the same on all processors. On a NUMA system, memory bandwidth is affected by the distance of the requesting processor to the allocated memory. However, this benchmark always allocates memory locally. Therefore, we expect to see an almost constant uniform value, similar to an UMA platform. The slight differences of the results can be attributed to the cache coherence protocol or to OS activities.

By following the methodologies described in the previous chapter, we executed 3.1.1 Benchmark 1 five times. Before the execution the *MP_BLIST* environment variable was set to using the following command:

export MP_BLIST=x

x is the number of the processor that we measure. *x* is a value in the range 0 to 15, where 0 is the first processor and 15 is the last processor. The *taskset* command is also used in order to bind and set the executable to a specific processor. The syntax of the command was:

taskset –c x bench.exe

x is the number of the processor that we measure. The benchmark was executed for an array of 2,000,000 elements or 32 MBs in size. The average value was used and is presented in Figure 7.

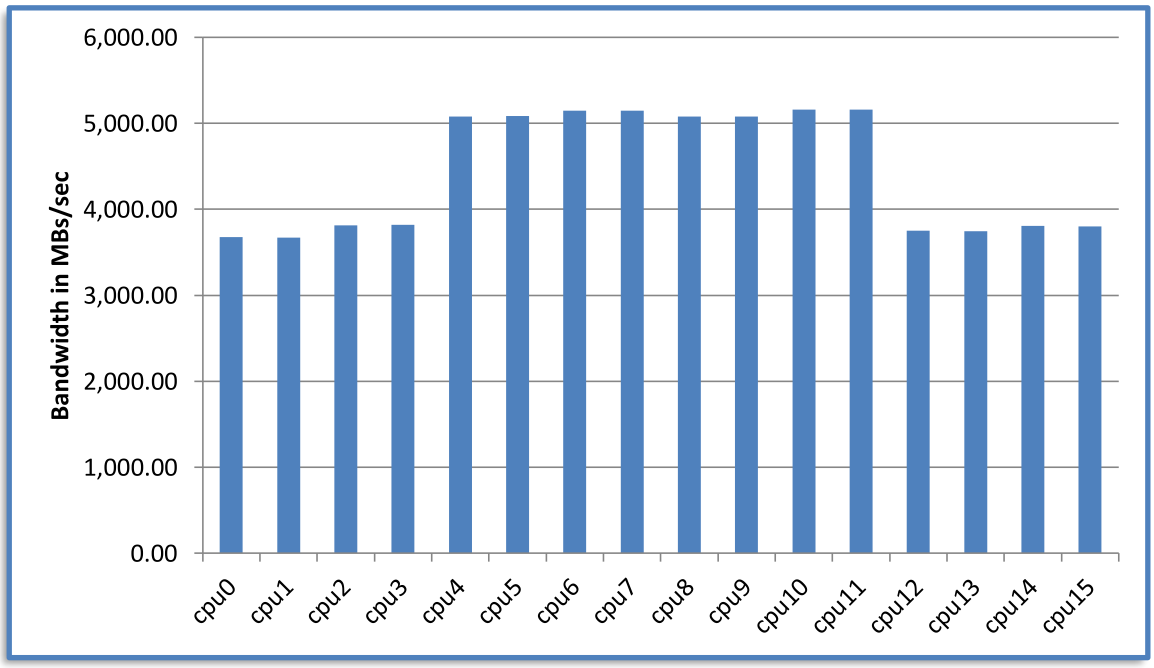

Figure 7 - Sun Fire X4600 Local Read Bandwidth Measurements (32 MBs)

The graph shows a difference of 39% between the best and worst bandwidth values. The dissimilar performance is not expected, since all nodes are using the same processor, have the same model of integrated memory controller and are attached to the same type of memory. Since all the nodes are identical, the difference in the performance must be caused by something else.

The different performance of the nodes can be accounted to the fact that the Sun Fire has a unique hardware topology and to the MOESI coherence protocol. For each level 1 cache miss the MOESI protocol requires at least 15 probe signals to be sent to every other processor in order to check the status of each cache line. These 15 probe signals require some processing on the remote processors and each remote processor has to reply with a negative acknowledgement. The MOESI messages use the HT links in order to reach remote processors. Therefore the latency of each HT link is added to the trip time of each message. Furthermore some processors are more than one hop away, which further increases the trip time of the message.

In order to check this assumption we executed 3.1.1 Benchmark 1 for an array of 4000 elements or 64 KBs in size which is the size of level 1 cache. We expect that by using a value that is equal or less than the size of the level 1 cache, the probe signals will not affect the bandwidth values. The results are shown in Figure 8.

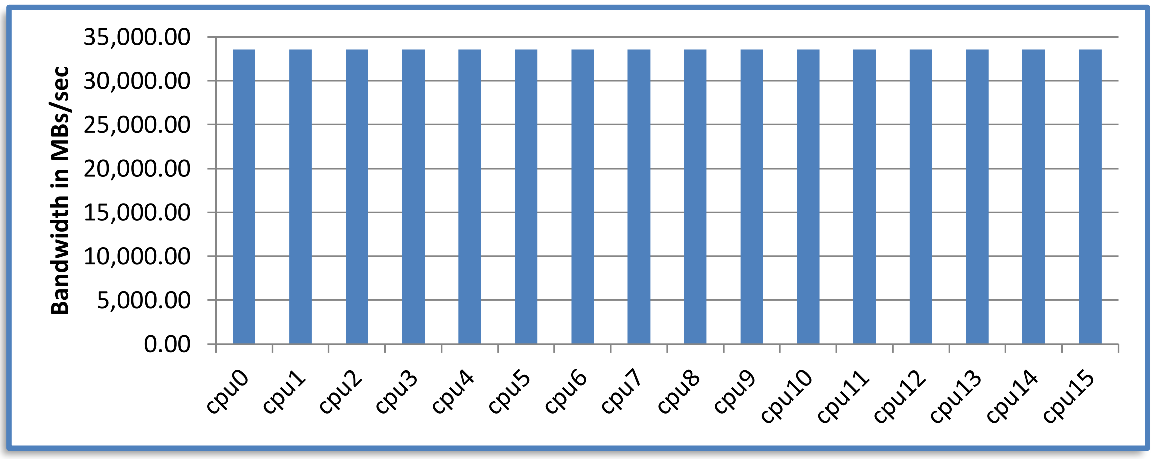

Figure 8 - Sun Fire X4600 Read Bandwidth Measurements (64 KBs)

The bandwidth results are grouped together measuring read memory bandwidth close to 33,554 GBs. Using micro benchmarks like the one above could introduce some small variations in the results. However, the 39% difference between the best and worst score is not present anymore. As we increase the size of the array, we observe that the bandwidth rate decreases as the size of the array increases. This behaviour is expected because for very small array sizes the array fits fully in the level 1 cache. Thus no cache misses occur, which do not trigger any MOESI probes. As the size of the array increases, the number of level 1 cache misses start to increase and delays due to the cache coherence protocol start to build up. Unfortunately, other researchers also used the X4600 platform and the installation of a kernel with hardware performance counters was not possible. Therefore, we were not able to measure the bandwidth and latency values of the HT links. However, the results of read memory bandwidth using different array sizes were plotted and presented in Figure 9.

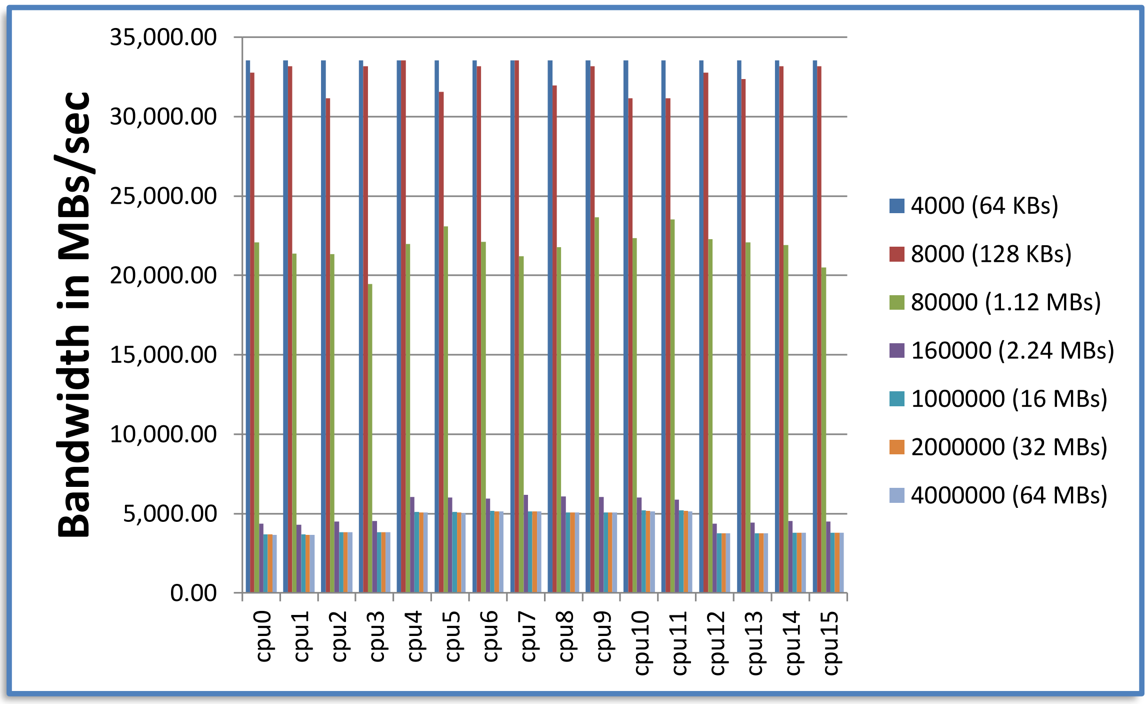

Figure 9 - Sun Fire X4600 Read Bandwidth Measurements (variable sizes)

The local read memory bandwidth benchmark showed that a platform with many NUMA nodes, like the X4600, achieves different bandwidth levels even when memory is placed close to a processor (local access). However, the benchmark also showed that four NUMA nodes (eight processors) are slow and four NUMA nodes are fast. Bandwidth performance should decrease as the number of hops increases, since each extra hop adds to the delays. According to Table 3 only two nodes require three hops in order to reach the other processors, node0 and node7. However, the benchmark data showed that node1 and node6 are also slow. Therefore, the number of hops and the NUMA distance is not the only cause of delay.

We have also showed that the real cause of the delay is the MOESI cache coherence protocol and the extra traffic that it creates. The cause of the slow performance of node1 and node6 however cannot be justified by the extra MOESI traffic. Kayi et al. [28] have also observed the same behaviour of an X4600 platform. They also fail to pinpoint a cause for the slow performance of node1 and node6.

Since local memory access is affected by the NUMA topology of the X4600 platform, the remote read memory bandwidth might help us to understand the irregular performance patterns. The remote read memory bandwidth benchmark allocates memory on the first processor on the first node (cpu0, node0). It then reads the allocated memory using threads that run on the other processors. We use both the PGI compiler environment variable *MP_BLIST* and the Linux command *taskset* to make sure that the threads are bound on a specific processor. Furthermore, the benchmark prints the hardware locality of the processor and memory location that it is bound in order to verify the binding.

This micro benchmark is used to measure the difference in read memory bandwidth caused by the NUMA distance of the requesting processor to the requested memory. In a uniform memory platform the bandwidth is constant. However, by definition a ccNUMA platform exhibits different memory performance levels when accessing remote memory. Using an application that deliberately allocates memory on a single processor and then tries to access from others; we hope to simulate the extreme scenario where a badly designed multithreaded application initialises data locally and then tries to access them using multiple threads that are executed on remote processors. This design decision might not cause any degraded performance on an UMA platform, however it can have catastrophic results on a ccNUMA platform. Since many applications are ported directly from older UMA to newer ccNUMA platforms we want to quantify the performance hit that a bad design decision would cause.

By following the methodologies described in the previous chapter, we executed the remote read memory bandwidth benchmark (3.1.1 Benchmark 1) 5 times. Before the execution, the *MP_BLIST* environment variable was set to using the following command:

export MP_BLIST=x,y

x,y are the number of the processors that we measure. *x,y* are in the range 0 to 15, where 0 is the first processor and 15 is the last processor. *x* is the processor that allocates the data array and *y* is the processor that access the array. The *taskset* command is also used in order to bind and set the executable to a specific processor set. The syntax of the command was:

taskset –c x,y bench.exe

x,y are the number of the processors that we measure. The benchmark was executed for an array of 2,000,000 elements or 32 MBs in size. The average value was used and is presented in Figure 10.

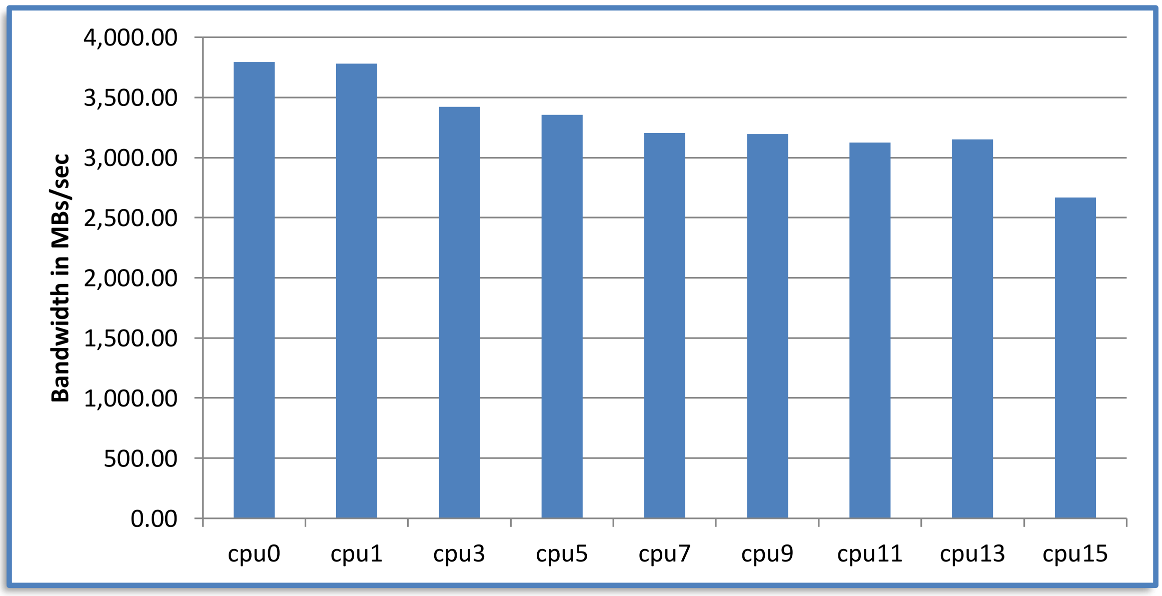

Figure 10 - Sun Fire X4600 Remote Read Bandwidth Measurements (32 MBs)

For clarity purposes, we present only odd numbered processors. Memory is allocated on the first processor (CPU0) and is then accessed serially by threads executing on the other processors. The results show that the effective read memory bandwidth for CPU0 and CPU1 is the same.

Furthermore, the value of the remote read bandwidth is the same as in the local read bandwidth benchmark (Figure 7). CPU0 and CPU1 are physically located on the same chip and therefore use the same memory controller and have access to the same RAM chips. Therefore, all memory transfers take place locally with little to no extra cost. The fact that their values are the same for the local and remote bandwidth benchmarks is validating the design of our benchmarks.

On the other hand, the other processors (CPU2 – CPU15) take a performance hit from accessing remote memory. These processors are physically placed on other chips. Therefore, all memory transfers take place through the HT links. Furthermore, the distance of each node from CPU0 is not the same. According to Table 3, the distance varies from 0 to 3 hops. The number of hops can be clearly observed from Figure 10 as small steps in the performance values. CPU2-CPU5 are one hop away from CPU0, whereas CPU6-CPU13 are two hops away and CPU14-CPU15 are three hops away.

The cost of transferring data through the HT links is not a constant reduction to read memory performance. One would assume that the reduction of read memory bandwidth would have been a constant percentage. Instead, it seems that there is a hard limit cap of data that can be transferred through one hop, two or three hops link. For example CPU3 read memory bandwidth is reduced from 3.8 GBs/sec (for local access) to 3.4 GBs/sec (for remote access), a reduction of around 10%. CPU5 is reduced from 5 GBs/sec (for local access) to 3.4 GBs/sec (for remote access), a reduction of around 32%. Both processors are one hop away from CPU0; however, their performance reduction differs by 300%. Therefore, the number of hops or NUMA distance cannot explain the drop in bandwidth performance.

The bandwidth limit of a HT link is 8 GBs/sec, according to AMD [16]. However, this limit is bi-directional. For unidirectional bandwidth the limit is at 4 GBs/sec. Therefore, the value of 3.4 GBs/sec that processors with one hop distance can achieve is very close to the limit of the link. It seems that the MOESI coherence protocol implementation on the X4600 and the integrated memory controllers are very efficient and can achieve bandwidth values close to 85% of the maximum theoretical bandwidth.

Figure 11 presents a comparison of the read memory bandwidth differences due to local and remote placement of memory (memory is allocated on node0, cpu0).

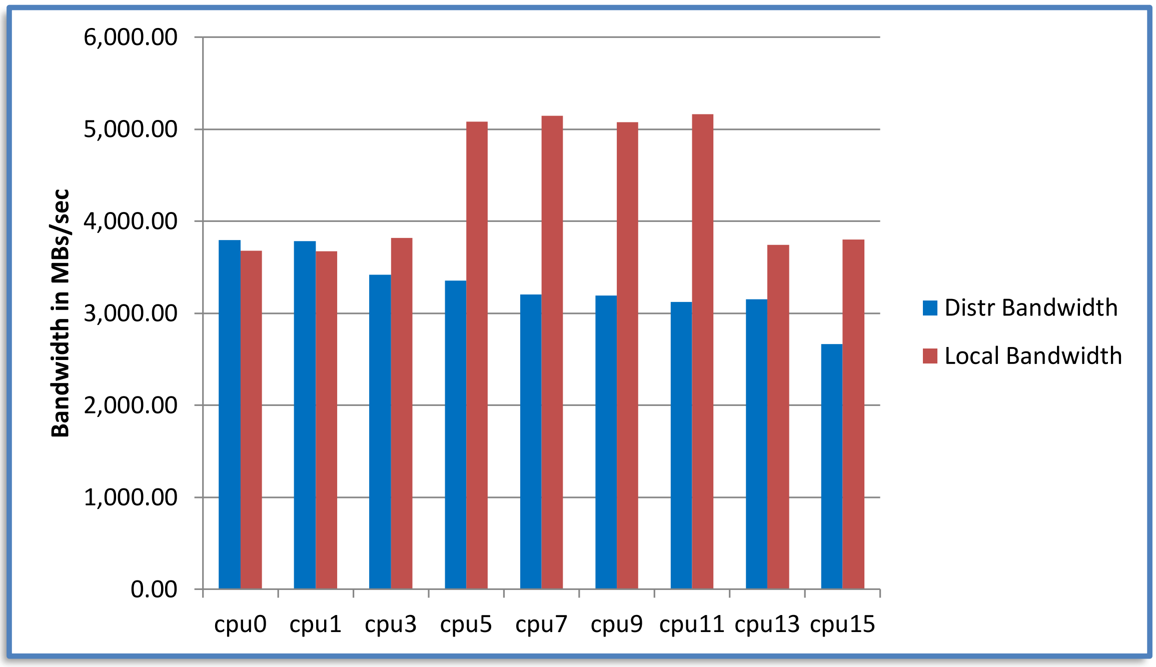

Figure 11 - Sun Fire X4600 Distributed/Local Read Bandwidth Comparison (32 MBs)

From the available data, it seems that the worst hit nodes are the central ones. However if we allocate memory on the central nodes, then this extreme behaviour is lightened and all nodes show a smaller more consistent drop in performance. Thus, the best location of memory which is shared between threads that run on different nodes is on the central nodes. These nodes offer the best read bandwidth for local access and satisfactory performance with a minor hit for remote accesses.

Another explanation pertaining to the slow performance of the outer nodes (node0, node1, node6 and node7) is the fact that their bandwidth performance is limited by the 4 GBs/sec limit of a unidirectional HT link. Even for local memory access, they never show bandwidth values more than 4 GBs/sec. However, the central nodes (node2, node3, node4 and node5) achieve local read bandwidth of 5 GBs/sec. This might point to a misconfiguration of the platform (either in hardware or in the BIOS) or abnormal calibration of the MOESI protocol that causes memory transfers for the outer nodes to occur over HT links. Since the two X 4600 platforms are a shared commodity, we were not able to access them as superuser in order to collect ACPI data. Furthermore rebooting the platforms in order to modify BIOS settings was not possible.

The second metric that we need to measure is memory latency. This metric will help us to improve our understanding of a shared NUMA platform. Memory latency refers to the time that a processor needs in order to retrieve a set of data from memory. The location of data in memory (local or remote cache lines, local or remote memory) significantly affects memory latency. Furthermore, the size of the requested memory affects latency. Contiguous small requests can sometimes be prefetched from main memory to level 1 or level 2 cache, thus saving the processor hundreds of CPU cycles. However, the open source benchmark *lmbench3* and our own latency benchmark try to make prefetching as hard as possible by using non-contiguous memory patterns.

As a first step, we use the open source benchmark *lmbench3*. *lmbench3* was used in conjunction with the *taskset* command. The format of the command was:

taskset –c x ./lat_mem_rd –N 1 –P 1 64M 512

x is the number of the processor that we measure and is in the range 0 to 15, where 0 is the first processor and 15 is the last processor. This command forces *lmbench3* to run on a specific processor. It also forces *lmbench3* to allocate memory on that processor. The "-N 1" switch is used to execute the benchmark only once. The "-P 1" switch is used to invoke only one process. The "64M" option indicates that the benchmark should measure latency up to 64 MBs of memory. Finally the "512" option indicates the stride of the benchmark, which is the amount of memory skipped until the next access. By using the *taskset* command we force the Linux OS to place the *lmbench3* executable on a specific processor. Thus, the first touch policy will allocate memory on that node. The output of *lmbench3* is memory latency in nanoseconds and the results are shown in Figure 12. The size of the array (x axis) is represented in logarithmic scale. From the graph, it can be seen that for small data size, less than the size of the level 1 cache (which is 64 KBs) all processors have the same memory latency. At around the 64 KBs size a small peak is introduced to the latency. The peak of local latency is noticeable because the array that *lmbench3* creates is slowly becoming larger than the level 1 cache. Therefore, more level 1 cache misses start to occur and the latency starts to increase.

As the memory size increases above the size of the level 2 cache (which is 1 MB) the memory latency of the processors is split in two distinct groups. The inner group shows low memory latency around 93 ns, whereas the outer group shows high memory latency at around 122 nanoseconds, which is around 35% slower than the outer group. The nodes are split in two groups in the same manner as in the local read memory benchmark. The size of the array (x-axis) is represented in logarithmic scale.

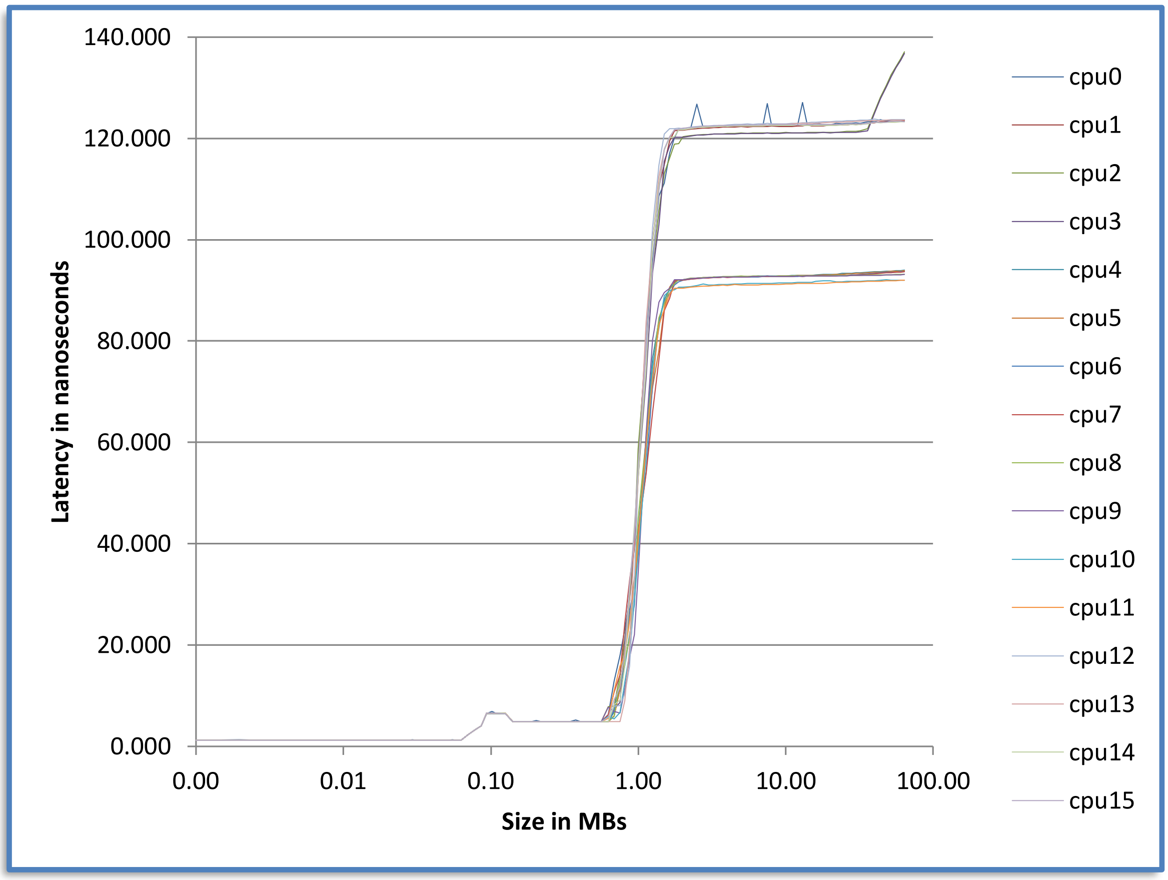

Figure 12 - Sun Fire X4600 Latency Measurements (lmbench3)

In order to verify the validity of the results of our own latency benchmark, we compared them with the results of *lmbench3*. The two sets are very similar, proving that our implementation is correct and is able to measure memory latency. 3.1.3 Benchmark 3 was executed five times and it followed the methodologies described in the previous chapter. The PGI environment variable *MP_BLIST* was set to:

export MP_BLIST=x

x is the number of the processor that we measure. *x* is a value in the range 0 to 15, where 0 is the first processor and 15 is the last processor. The *taskset* command is also used in order to bind and set the executable to a specific processor. The syntax of the command was:

taskset –c x bench.exe

x is the number of the processor that we measure. The benchmark was executed for memory sizes up to 64 MBs in size. The average value was used and is presented in Figure 13.

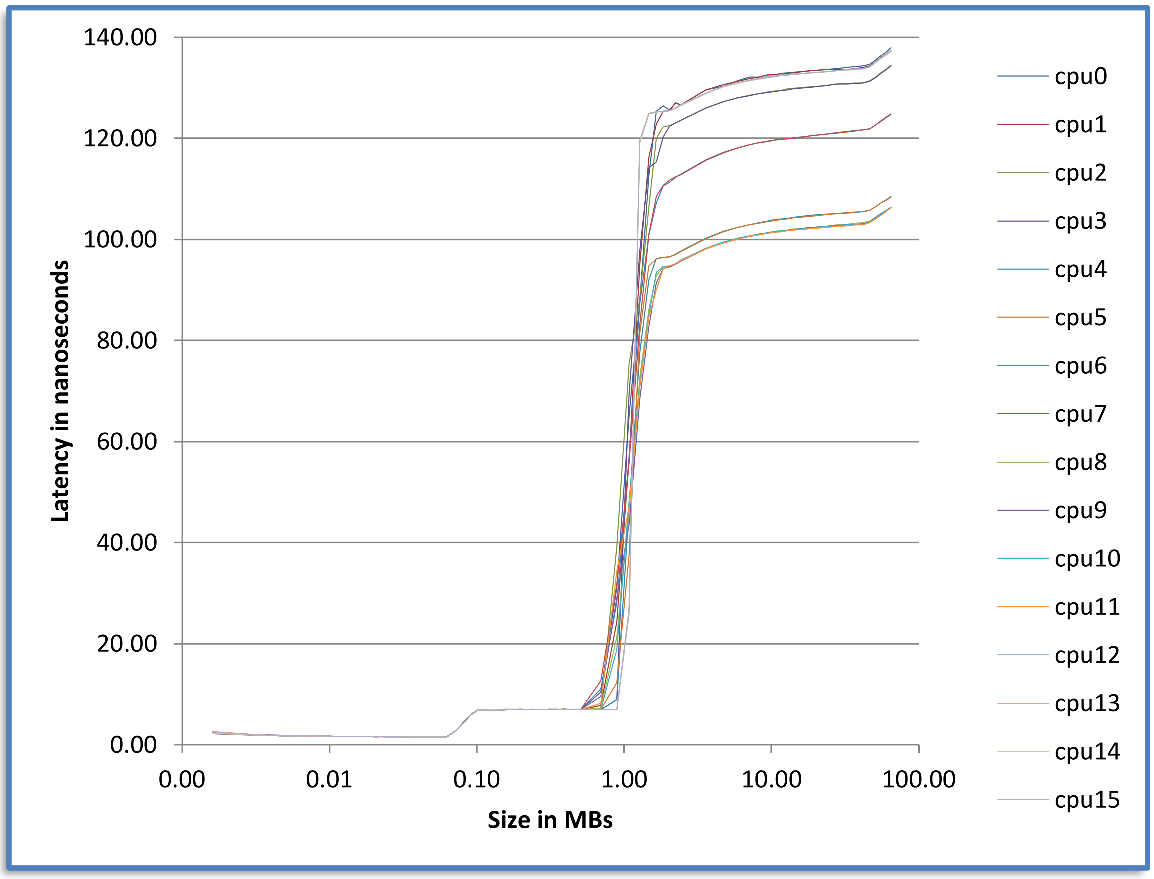

Figure 13 - Sun Fire X4600 Latency Measurements

The size of the array (x-axis) is represented in logarithmic scale. The two graphs are remarkably similar and they show the same behaviour by the X4600 platform. There is an increase of the latency around the 64 KBs point, which is expected since data are now coming from level 2 cache. Whilst the array is smaller than level 1 cache, the processor could prefetch the whole array in level 1 cache. However as the array becomes larger than level 1 cache, access to memory is dictated by the slowest link which the level 2 cache. Level 1 cache accesses are measured in the range of 1-2 nanoseconds. Level 2 cache accesses are measured in the range of 6-8 nanoseconds.

As the array increases in size, the data become larger than the cache memory. Thus, the inability of knowing the next element of the array causes the memory latency to increase, as the processor is not able to prefetch the data. Due to the design of the micro benchmark all prefetching efforts by the processor should result in the selection of the wrong data. The array is constructed in a pseudo-random arrangement in order to further hinder the prefetching abilities of the processor. As the processors cannot effectively prefetch data, level 1 cache misses start to occur. Each level 1 cache miss incurs the extra cost of the MOESI protocol signalling. As stated earlier, each level 1 cache miss is followed by a series of signals sent to the other processors in order to find the missing data in the cache memory of another processor. If the data are found, then the cache controllers initiate a direct memory copy through the HT link. If all participating processors reply negatively, then the memory controller fetches the data from the local memory (either a higher level of cache or from main memory). Since no other processor reads data during the duration of this benchmark, the replies will always be negative. Therefore, this benchmark measures the extreme case where the data will always be fetched from local memory and never from the cache controller of another processor.

In order to check the other extreme case, where data are always fetched from a remote processor we use the distributed latency micro benchmark. This benchmark allocates the array on one processor and then tries to access it from another. This memory allocation will test the extreme scenario where memory is allocated only on the first processor. This scenario takes place when an OpenMP (or any threaded application) does not use parallel initialisation of data. Linux does not support automatic relocation of data pages, therefore all data will be allocated on the processor that executes the *main()* function of the application. Until an OpenMP directive (or a pthread create function) is encountered the application is single threaded and executes on a specific processor. If data are initialised at that point in time, the allocation of memory will take place on memory controlled by that processor.

The topology of the X4600 platform requires 0-3 hops in order to access remote memory; therefore, the latency of the operation is affected by the NUMA distance of the two processors. The number of hops can be easily seen in Figure 14. The processors close to CPU0 are grouped together. The processors two hops away are grouped together and the last processor, which requires three hops to access the data, is standing on top. The size of the array (x-axis) in the following figure is represented in logarithmic scale.

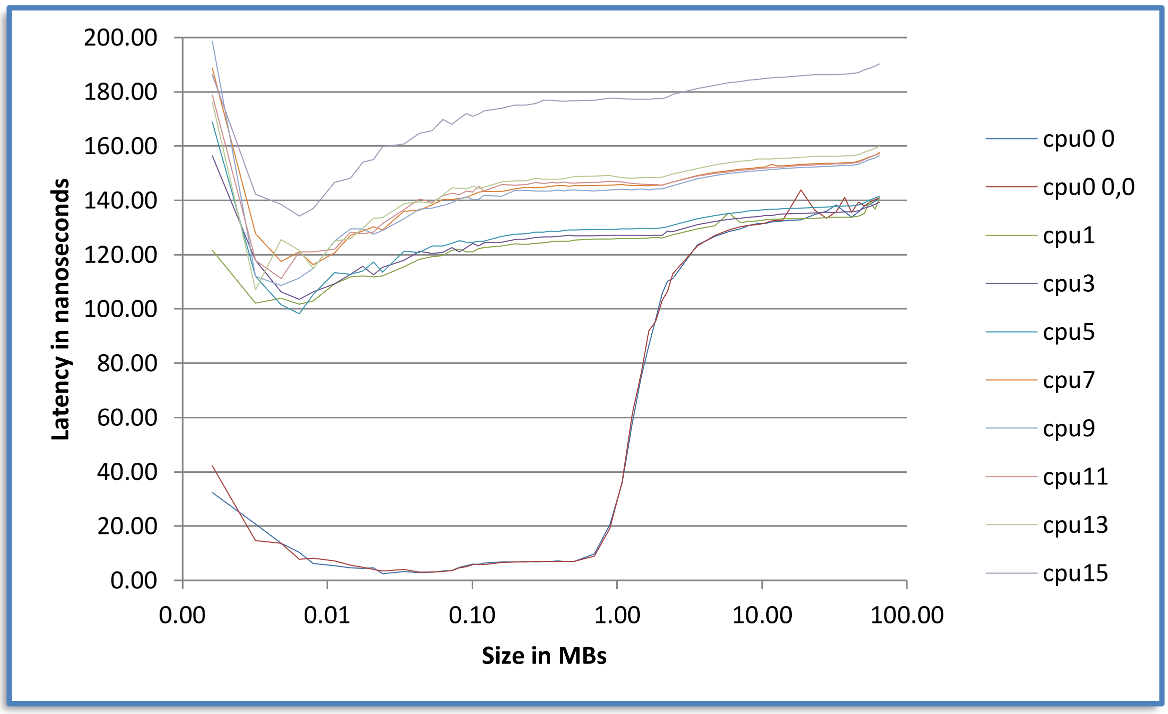

Figure 14 - SUN X4600 Distributed Latency Measurements

Memory is allocated on CPU0. A single threaded application and the dual threaded benchmark are used to traverse the array. The results from the two tests are values “cpu0 0” and “cpu0 0,0” respectively. Two different applications are used in order to check the validity of the results. From the results it seems that it doesnot matter if a single threaded or a dual threaded application is used, since the latency times reported are identical. Both threads of the dual threaded application are bound on the first processor (CPU0).

The array is traversed by threads executing on other processors. As the NUMA distance between the processors increases the latency also increases. The contrast with the previous benchmark is that when processors try to access small arrays they need to incur the full cost of the MOESI protocol. Therefore for every level 1 cache miss, the local cache controller sends MOESI probes to the other cache controllers via the HT links. It then waits until the other cache controllers look for the requested data and reply back. For small arrays only CPU0 replies with a positive answer, whereas for larger arrays every processor replies with a negative answer.

Furthermore, for small arrays, the cache contronller of CPU0 and the cache controller of the requesting processor need to update the status of each cache line and set the new onwer of the data, thus more MOESI signalling is required to be sent across the HT links. The signaling is usually embeded in the data messages, however as more messages are required in order to complete the transaction, performance takes a hit. Similarly the cost of invalidating cache lines in the first processor's cache controller also adds up. For very small arrays (couple of cache lines long) the cost of prefetching data and instructions and of unrolling loops adds time that is a considerable percentage of the duration. Therefore, any wrong prediction by the compiler, regarding the number of times that a loop needs unrolloring could affect the total runtime. This behavior explains the high latency values for very small arrays, which is represented as a spike on the left hand side of the chart.

Figure 14 shows that latency is always high for remote processors. The values range in the 100-160 nanoseconds with the exception of CPU15, which is the only that requires 3 hops, which has a range of 135-190 nanoseconds. Comparing with the previous benchmark the latency of accessing local RAM is the same as accessing remote cache lines. This behavior means that accessing misplaced data (data that are resident in remote caches) is as costly as accessing main memory. Also since this benchmark only reads remote memory, it does not measure the effect of modifying remote memory, which will also add to the delay. As the array gets larger and it can not fit in the cache, the latency levels off. For small data arrays (less than level 1 cache) latency has risen from 1-2 nanoseconds to 100 nanoseconds for the nearest processor, 120 for the middle processors and to 150 nanoseconds for the furthest away processor. For medium data arrays (less than level 2 cache) latency has risen from 6-8 nanoseconds to 128 nanoseconds for the nearest processor, 145 nanoseconds for the middle processors and to 175 for the furthest away processor.

Another finding is that the latency of CPU1 is comparable with the latency of CPU3 and CPU5. According to Table 3, CPU3 and CPU5 are only one hop away from CPU0. However CPU1 is on the same node as CPU0. Due to the design of the AMD Opteron chip, each CPU has dedicated level 1 and level 2 cache memories and a dedicated memory controller. This benchmark shows that even the processor that is physically placed on the same chip as the data, incurrs a considerable cost when it tries to access them. Instead of using a direct memory copy through an internal bus, it tries to access the data through the MOESI protocol. It seems that even though the two processors are in the same chip, the MOESI messages must travel through an external bus, since the observed latency of CPU1 is the same with CPU3 and CPU5. This behaviour seems irregular, it however shows that the cost of the MOESI protocol is very significant. However this design decision of the AMD Opteron might favor other circumstances, where more cores are used inside a single chip. Unfortunatelly an X4600 platform with quad core AMD Opterons was not available for testing.

The processors that require two hops in order to reach CPU0 (CPU7-CPU13) are grouped together. The extra latency of the second hop is an almost consistent 16 nanoseconds difference, which is observed across all collected data points. CPU15 has the highest latency, since it requires three hops to reach the data. The extra hops requires 30 nanoseconds more than the two hops. It seems that the cost of each HT hop is not the same.

The final metric that we want to measure is contention over the HyperTransport links and on the integrated memory controller. In order to measure this NUMA effect, we allocate memory on the first processor and try to read from the same memory using all sixteen processors. This experiment will cause many snoop messages to be sent along the processors. it will also test the performance of the IMC of the first processor. Since the X4600 exhibits different performance levels for the inner and outer cores, we executed the experiment two times. Once for the inner cores and once for the outer cores. Furthermore, we varied the size of the array in order to investigate if the size of the array affects the performance. The results are presented in Figure 15.

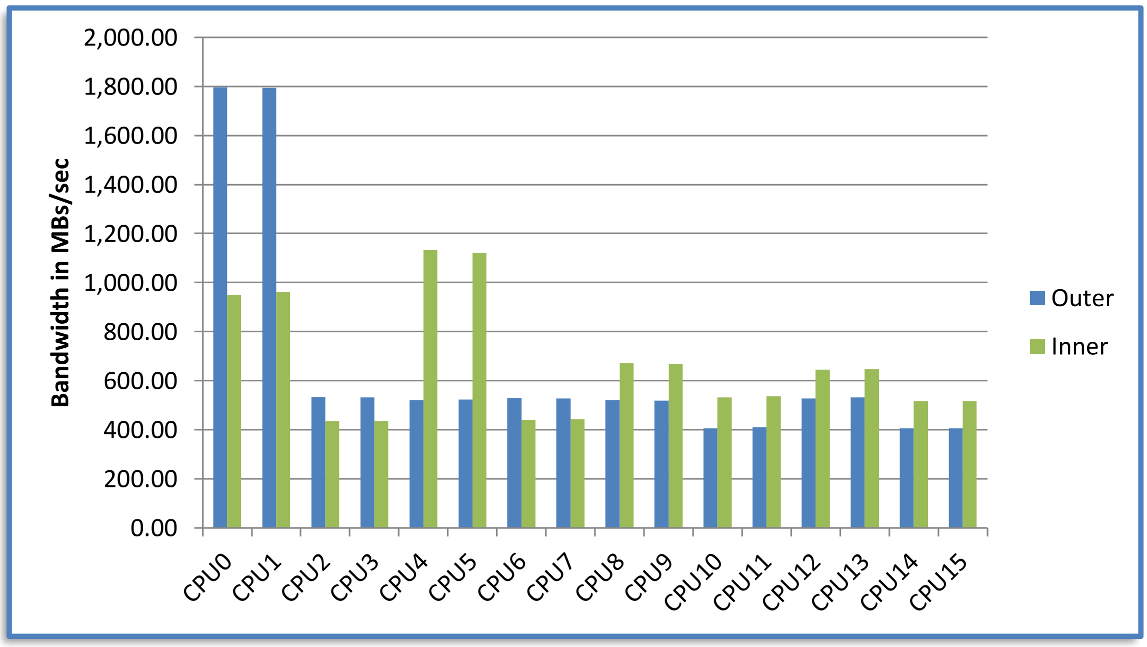

Figure 15 - SUN X4600 Contention Measurements

The results showed that the size of the array does not change the performance. We tried three different sizes (12 MBs, 32 MBs and 320 MBs), however all values are almost identical.

Memory was allocated on the first processor (CPU0) for the outer experiment, whereas memory was allocated on the fifth processor (CPU4) for the inner experiment. In order to set the proper affinity we used the *MP_BIND* and the *MP_BLIST* variables. We also used the *taskset* command. The benchmark displayed the processor and memory affinity in order to verify correct binding.

The results show that the overall read memory bandwidth has decreased. We notice that bandwidth decreased along all processors. The processors that access local memory (CPU0 – CPU1 for the outer experiment and CPU4 – CPU5 for the inner experiment) are showing better performance. However, CPU0 could access local memory at 3.7 GBs/sec when there was no contention from the other processors, whereas now it can only achieve 1.7 GBs/sec. Furthermore, CPU4 could access local memory at 5 GBs/sec, whereas with other processors trying to access the same memory location it can only manage 1.1 GBs/sec. Therefore, contention for the same memory location decrases the available bandwidth and the overall performance of an application.

Another interesting fact is that the usually slow pair of the first two processors (CPU0 – CPU1) manages to perform very well on the inner core experiment. We assume that the fact that all shared libraries and the OS reside on memory allocated on node0 causes these two processors to have an unfair advantages compared to the other processors. The output of the *numactl –show* command shows the available and free memory per node. Executing this command without any application running, we notice that each node has 4.1 GBs of memory. However all the nodes have 4 GBs of memory free, except from the first one that only has 1.5 GBs free. This observation proves that the first node is using more memory than the others on an idle system.

4.2 Sun Fire X4600 Applications

The results of the previous benchmarks showed that the location of memory affects performance. In the extreme example of the distributed latency benchmark, access to memory stored in the cache of a remote processor is as expensive as accessing local RAM. Furthermore, the bandwidth benchmarks showed that not every node in an X4600 platform exhibits the same read memory bandwidth performance. The inner cores have higher memory bandwidth than the outer cores.

These results should affect the performance of real applications. In order to check the performance differences of real applications we use two kernel benchmarks from the NPB suite. The two kernels are CG and MG. As stated earlier CG stresses the latency of a memory subsystem whereas MG stresses the bandwidth. Both benchmarks should accurately simulate the performance of a real application. For our experiments, we use classes B and C. Class B problems are medium sized whereas class C are larger problems and take more time to compute.

Both benchmarks use OpenMP directives to initialise data as close as possible to the executing threads. Therefore, they avoid placing all data on the first processor, which might cause significant drops in performance due to the increased NUMA distance.

The first experiment will try to validate our earlier finding; that the central nodes are faster than the outer nodes. Furthermore, it will try to validate if placing the two threads on the same node or on different nodes affects performance. In order to check if the central nodes are indeed faster than the outer nodes for a real application, we use the *taskset* command to restrict execution on specific processors. The command used was:

```
taskset –c x-y ./cg.B.x
```

x,y are in the range of 0-15 and represent the available processors. The results of this experiment are presented in Figure 16 and lower values are better than higher.

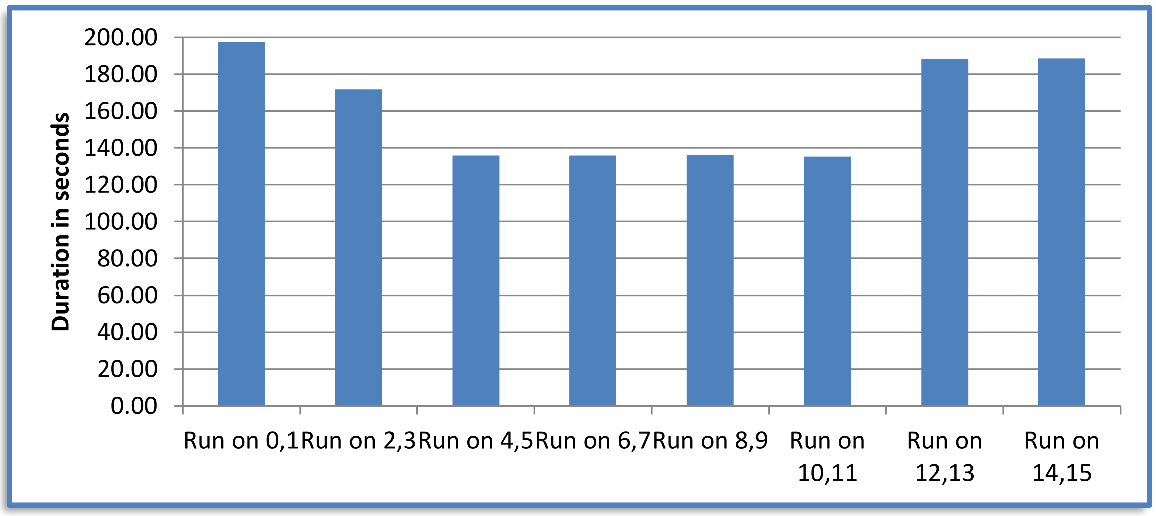

Figure 16 - CG (class B) using 2 cores on X4600

Not all combinations are used, since the use of only two threads would result in many more experiments if we wanted to exhaust all placement possibilities. However, the empirical data show us that placing threads further away reduces performance. Therefore, the best allocation is for the two threads to share the same node. Furthermore, the performance of the central nodes outperforms the outer nodes by 30%.

In our previous findings, we have calculated that the central nodes are 39% faster in terms of read memory bandwidth. However, a real application also performs writes and it uses data for computations. Therefore, the 30% difference between the inner and outer nodes is a validation of our previous results.

The first results are positive and seem to enforce our findings. However using only two threads on a platform that has sixteen processors is a waste of resources. Very few real life applications would use only two threads. Therefore, our next experiment tries to quantify the performance differences when we use four threads for the CG.B benchmark. The results are shown in Figure 17.

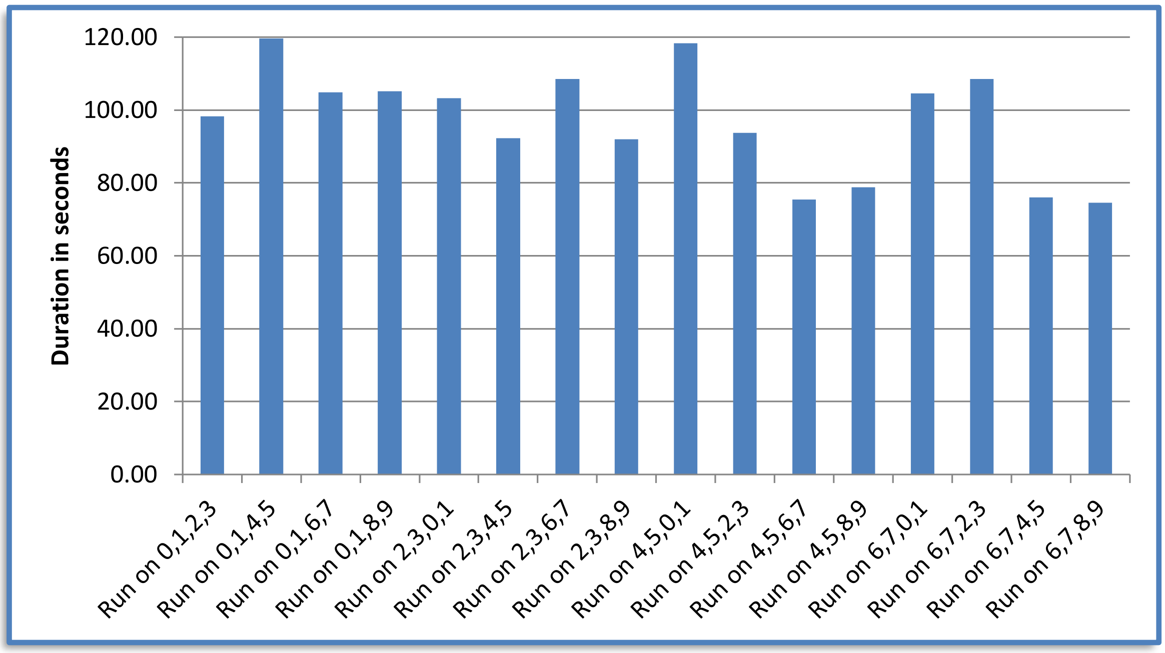

Figure 17 - CG (class B) using 4 cores on X4600

For clarity and readability purposes, the first 16 benchmarks are shown, since exhausting all placement possibilities would consume too many CPU cycles. Using four threads, we can reduce the runtime of the CG.B kernel benchmark by 38% if we place the threads on the appropriate processors. The worst placement is a combination of outer and inner nodes that runs for 119 seconds. However, by placing the four threads on inner nodes we manage to reduce runtime to 75 seconds. The fastest combination used node2 and node3.

According to AMD guidelines on performance for AMD Opteron cache coherent NUMA systems [29], there are four factors that influence performance on a multiprocessor system:

- The latency of remote memory access (*hop latency*)
- The latency of maintaining cache coherence (*probe latency*)
- The bandwidth of the HyperTransport interconnect links
- The lengths of various buffer queues in the system

By placing the four executing threads in central nodes, we use two of these factors in our advantage:

- Higher memory bandwidth
- Lower latency

The faster values are consistent for all inner nodes. However, performance suffers when some inner and some outer nodes are combined. When a combination of inner and outer nodes is used, we suffer from low bandwidth and from high latency. Furthermore, the number of hops between the nodes increases. Therefore, it is advisable to avoid such combinations. By using only inner or only outer nodes, we manage to minimise the number of hops. Using the same number of hops, the inner cores are faster because they show higher bandwidth and lower latencies.

Furthermore, CPU0 shows up in all slow runs. CPU0 is a special case, since most OS related activities take place on CPU0. There are solutions on distributing OS activities to other processors, however most desktop (and some server) Linux systems run OS related activities on the first processor by default.

Therefore, when an application can scale up to four threads it is advisable to use the inner nodes of an X4600 platform. Our next experiment tries to confirm if this trend holds true for applications that scale up to eight threads.

The results from executing the CG class B benchmark using eight threads are available in Figure 18. The results show that the inner cores are consistently faster than all other combinations. The best placement strategy yields a 21% performance improvement and takes place when the inner nodes are used. The worst possible placement strategy is to mix inner and outer nodes.

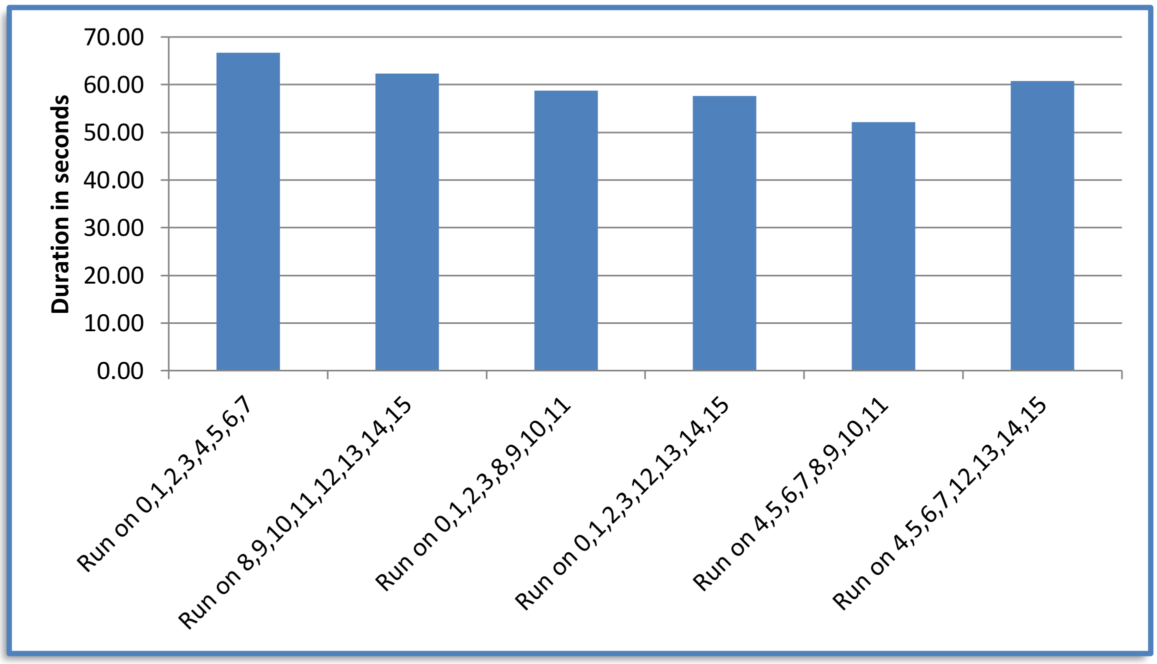

Figure 18 - CG (class B) using 8 cores on X4600

The same behaviour is exhibited by the CG class C benchmark. The CG class C benchmark uses a larger array and consumes more memory. The number of iterations is the same as the class B benchmark. The following figures (Figure 19 and Figure 20) show a summary of the experiments performed on CG.B and CG.C showing the best and worst runs. Since the list of possible placements is not exhausting, the actual best and worst runs might not be presented here. However, the differences in duration should be negligible. The results for both benchmarks are very similar and show that explicitly bounding threads to nodes can increase performance. Eight thread runs come even closer to the duration achieved by using sixteen threads.

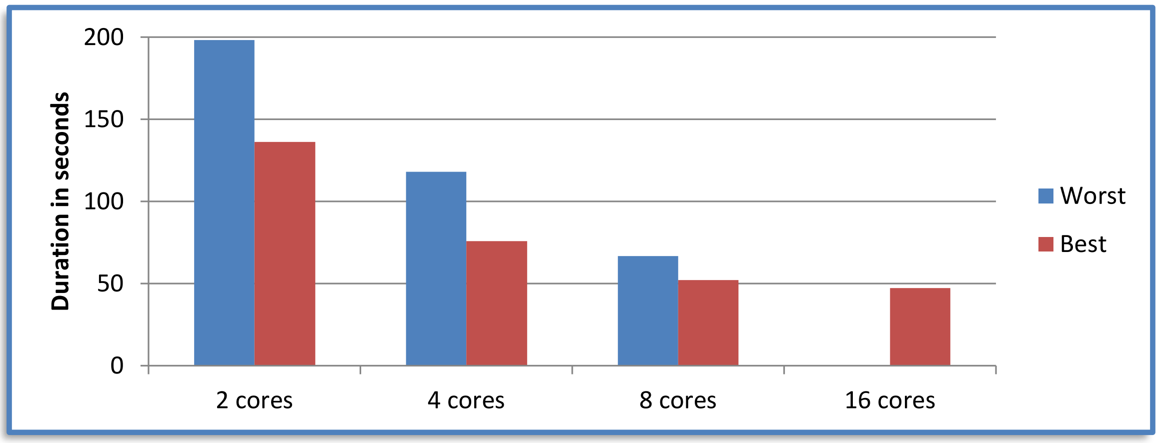

Figure 19 - CG.B comparison on X4600

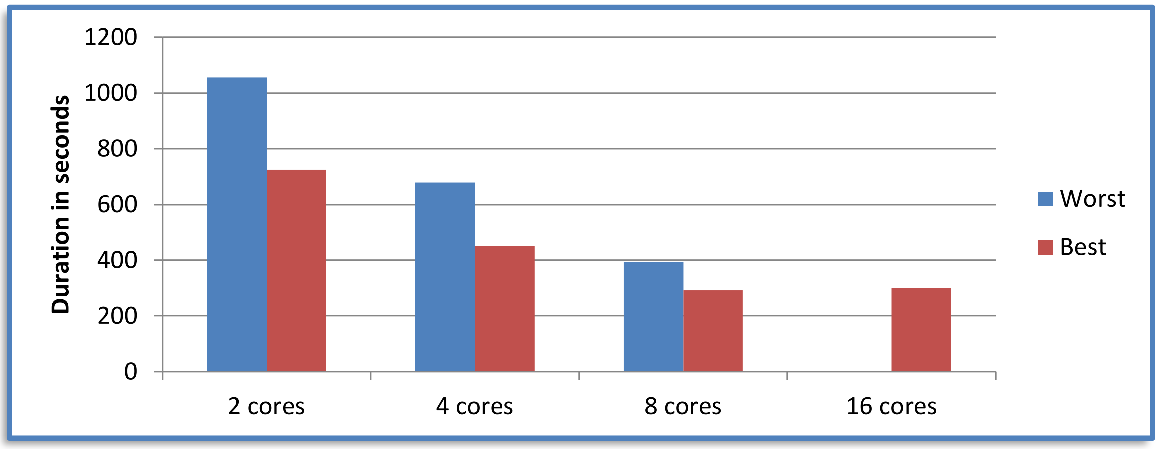

Figure 20 - CG.C comparison on X4600

The comparison shows a dramatic increase in performance for the CG.B and CG.C kernel benchmarks. This increase in performance was accomplished without changing a single line of code or recompiling the executable. We did not modify the allocation policy of the threads or used complicated algorithms to migrate pages whilst the application is still executing. We only observed the NUMA characteristics of the platform and tuned the placement of threads. Each thread allocated memory locally using the first touch policy and the result was a significant boost of performance. Furthermore, the performance of the eight threaded run is as fast as the run using sixteen threads, which could lead to saving a considerable amount of processor cycles.

The results of the MG.B and MG.C kernel benchmarks are not shown here. Comparisons of the best and worst runs for MG.B are presented in Figure 21.

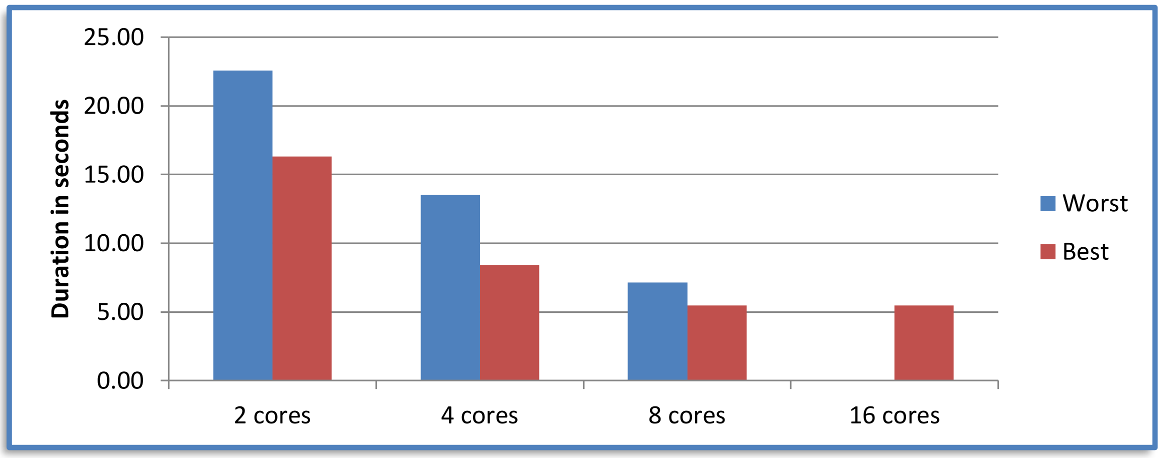

Figure 21 - MG.B comparison on X4600

However, the results are identical to the CG benchmarks. As discussed earlier the MG benchmark is bandwidth bound. Even though the two benchmarks stress different aspects of the memory subsystem (latency and bandwidth respectively), the placement of threads on particular nodes improves performance for both benchmarks.

However, and this is true for both benchmarks, as the number of threads increases the effect of placing threads to faster nodes diminishes. This is an effect of slower nodes being added to the mix. Thus experimenting with more than eight threads did not yield any performance improvements. However, the fastest eight threads run is as fast (or even faster in the case of the CG.C benchmark) as the sixteen threads run.

We also experimented with over-subscription of threads on the faster nodes. However, this resulted in worse performance than using some fast and some slow nodes. The reason behind this behaviour is that the OS scheduler is trying to be fair to all applications. This is especially true for desktop systems where interactivity of applications is usually judged by the response time. Therefore, the OS scheduler continuously awakes and puts threads to sleep. However, this behaviour causes an excess of context switches which take hundreds of processor cycles to complete. Furthermore, the cache lines and the TLB after a context switch are filled with data from the previous thread, which causes extra delays until the correct data are fetched from main memory to fill up the various cache levels and memory reorganisation fills up the TLB with the appropriate memory pages. Li et al. [30] have discussed and quantified the effects of task migration on multicore processor systems.

So far, we have used *taskset* to bind the execution of threads on specific processors. However as we have already showed Linux supports another tool, called *numactl*. This tool allows binding of threads to processors and enables the allocation of memory on specific nodes. It also enables allocating memory in interleave mode, where a round-robin algorithm is used to allocate memory on NUMA nodes. The allocation node specified by the *numactl* command overrides the internal memory allocation policy. It therefore allows researchers to experiment with placing memory on different nodes than executing threads or placing memory in interleaved fashion without modifying the source code of the benchmark.

The allocation of memory on specific nodes is accomplished by using the Linux kernel functionality of migrating pages to specific nodes (or even to a specific memory address). This functionality is a very powerful tool for programmers that are very confident that they know the memory access patterns of their applications. An application can use the page migration API to move pages whilst the application is running. Migrating pages takes time and is an expensive operation. If however the performance benefits of better memory locality outperform the initial cost of migrating a page then the total runtime should improve. A cost benefit analysis of the migration must be performed and the migration can take place if the application has a steady (or at least well-understood) memory access pattern.

The experiments were performed using the local allocation policy and the interleave method. The local allocation policy option does not take any arguments as input and it allocates memory on the local processor. The interleave method takes as arguments one or more node IDs and allocates memory on those nodes only using a round-robin algorithm. If not enough memory is available on any node the *numactl* tool fails and the execution of the benchmark is not started.

The results of the *numactl* tool experiments show that both tools, *numactl* or *taskset*, can achieve the best runtime scores. However using *numactl,* we have more options available and we can perform more experiments. Most of the extra combinations however yield worse performance. The worst performers are the outer nodes. However executing threads on outer nodes whilst allocating memory on inner nodes decreases performance even more. For instance, allocating memory on nodes 0 and 1 (outer nodes) and executing the four threads on the same nodes produces a runtime of 93 seconds. Allocating memory on nodes 2 and 3 (inner nodes) and executing the four threads on the same nodes produces a runtime of 75.97 seconds. However allocating memory on nodes 2 and 3 (inner nodes) using the interleave option and executing the four threads on nodes 0 and 1 (outer nodes) produces a runtime of 124 seconds. Therefore, if outer nodes have to be used, then the allocation policy should be local.

Using the *numactl* tool to allocate memory on one node only reduced performance. The option is called *membind* and takes as arguments node IDs. However, *numactl* allocates memory on the first supplied node until no free memory is available. It then moves on the list of provided node IDs. Using *numactl* and *membind,* we can allocate memory on a single node. Using this setup, the runtime results are consistently worse than any previous result. The reason is that the bandwidth that a single integrated memory controller can achieve is limited. Furthermore, the HT link can only transfer 4 GBs/sec in one direction. When two or more threads try to access the same memory that limit is easily reached. Even the inner nodes memory controllers are saturated when more than two threads try to read/write continuously on the same memory. Therefore, the best allocation policy is either local or interleaved. Figure 22 shows the results of various run of benchmark CG class B using four processors on different NUMA nodes for computation and different NUMA nodes for memory allocation. The columns labelled “(x,y) local” represent executions on four processors on nodes *x* and *y* where the allocation policy is local. Columns labelled “(x,y) interleave on (z,w)” represent executions on four processors on nodes *x* and *y* where the allocation policy is to interleave memory on nodes *z* and *w*.

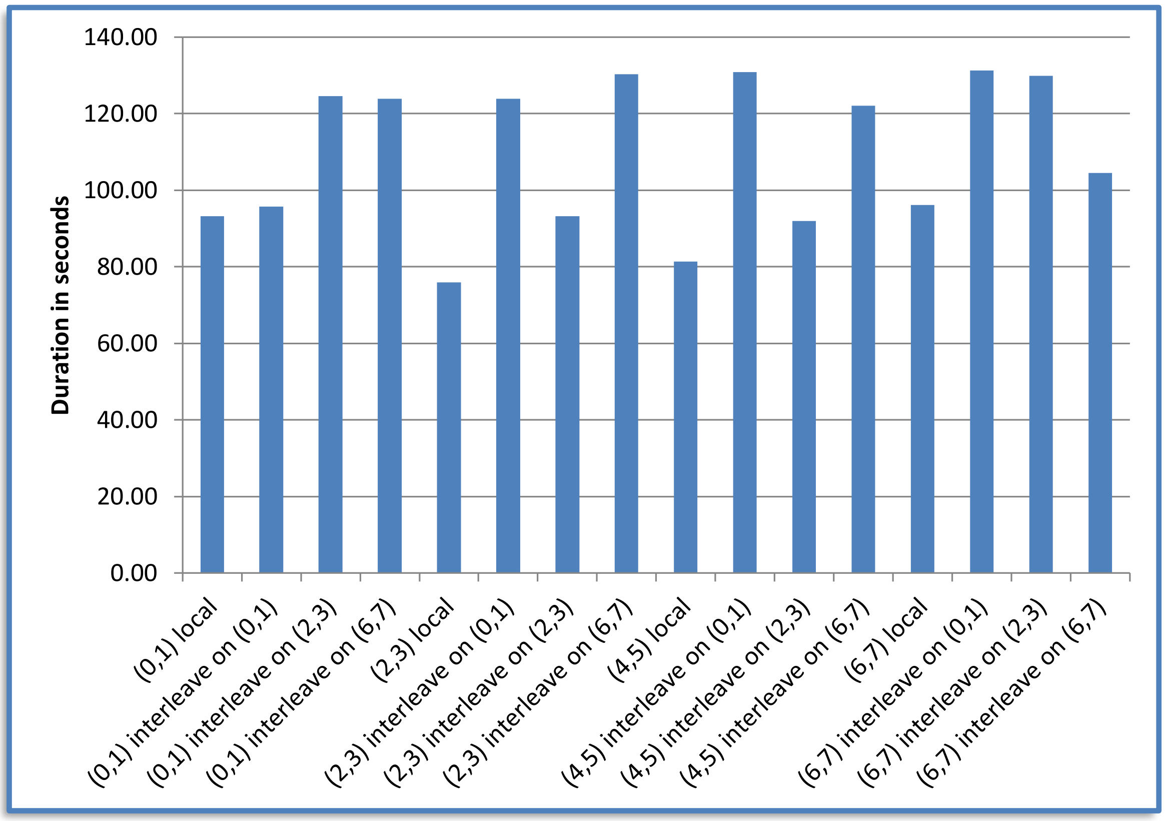

Figure 22 - CG (class B) run with numactl using four cores (two nodes) on X4600

In order to verify our analysis that memory location can have disastrous effects on performance we modified the source code of the CG kernel benchmark. The CG benchmark initialises data using parallel OpenMP directives. It avoids therefore the most catastrophic allocation of memory, which is to allocate every data structure on the first node. Our modification removed the parallel initialisation, forcing memory to be allocated on the first node.

Furthermore, we executed the modified version of the CG benchmark alongside an original version and an experiment that used *numactl* to force memory allocation on the first node. The columns labelled "LOCAL" and "ORIGINAL" depict the duration times of the modified and of the original version respectively. The columns labelled numactl use the *numactl* tool to force memory allocation on the first node. All numbers depict nodes and not actual processors. The results are shown in Figure 23.

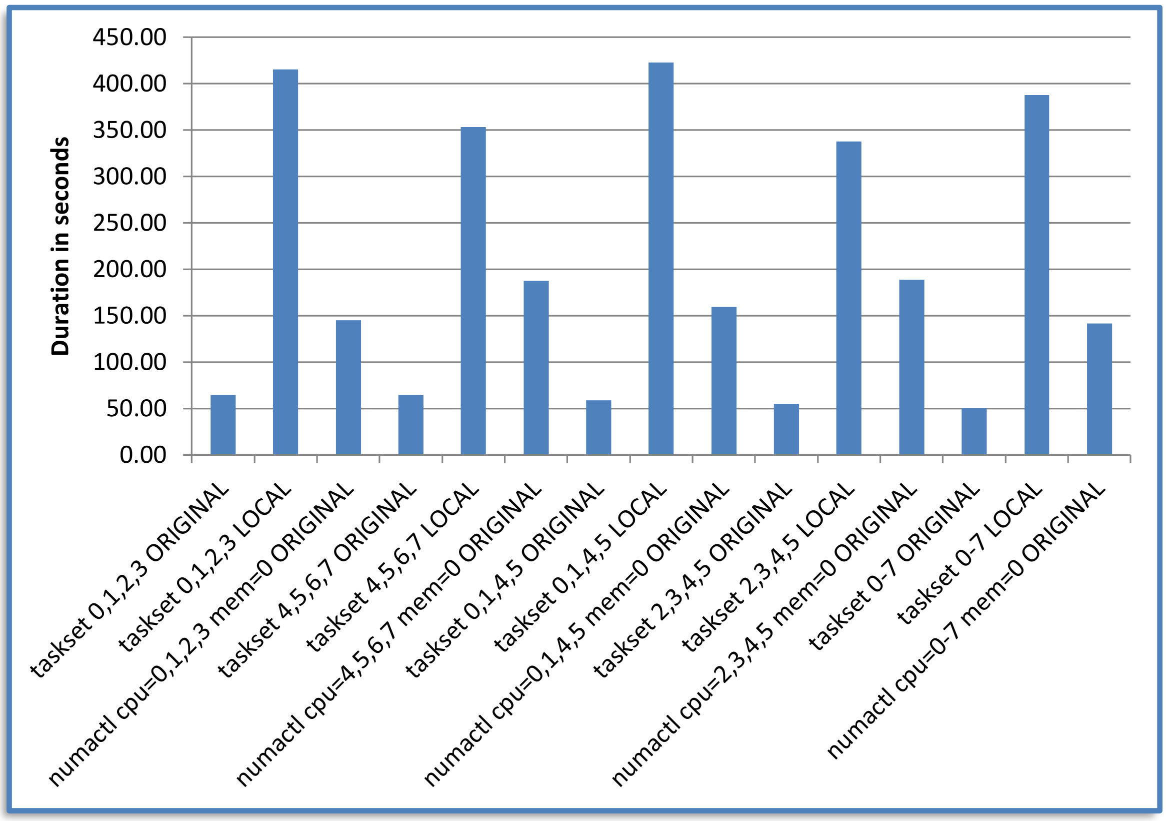

Figure 23 - Modified CG.B on X4600

The results show a dramatic drop in performance. Using the original version of the benchmark and eight threads the duration of the experiment was 64 seconds when we allocated the threads on the first four nodes. However, when we allocated memory on the first node using our modified version the duration of the experiment increased to 415 seconds, a 650% increase. On the other hand, when we used *numactl* to force memory allocation on the first node the experiment lasted for 145 seconds.

The difference of the modified version of the CG benchmark and the experiment using *numactl* is puzzling. n*umactl* is used to allocate memory on specific nodes, however the values returned from this experiment suggest that the allocation was not limited on node0. We used the *numastat* tool to collect memory allocation statistics before and after our experiments. n*umastat* reported various allocations that did not correspond to our memory allocation policies. We therefore wrote a command line script that copies the file */proc/<pid>/numa_maps* to our home directory, whilst the benchmark is still running. The file *numa_maps* stores the memory allocations of each process. We used a perl script created by J. Cole [49] to analyse the results. n*umastat* returned the results presented in Table 4, when we used the *taskset* command. The *taskset* command was used with the identifiers of the first eight processors. Therefore, we assume that memory would be allocated on the first four nodes. However, *numastat* reports that node4 allocated almost 50% of used pages.

	node0	node1	node2	node3	node4	node5	node6	node7
numa_hit	26,324	22,174	21,868	21,955	50,436	1,406	408	196
numa_miss	0	0	0	0	0	0	0	0
numa_foreign	0	0	0	0	0	0	0	0
interleave_hit	0	0	0	0	0	0	0	0
local_node	26,316	22,168	21,859	21,946	50,428	1,397	404	192
other_node	8	6	9	9	8	9	4	4

Table 4 - numastat results on X4600

However, the data that the *numa_maps* file reported were different. The data are available in Table 5. The data from the *numa_maps* file indicate that memory was allocated properly on the first four nodes. It seems that *numactl* reports per system memory allocations and not per process. One would assume that not many memory allocations would occur during the runtime of the benchmark. However, the Linux OS managed to allocate and deallocate 0.16 GBs of pages.

	node0	node1	node2	node3
Pages	22,091	21,762	21,800	21,865
Size in GBs	0.08	0.08	0.08	0.08

Table 5 - numa_maps results on X4600

The results of the *numa_maps* file for the experiments using our modified version of the CG benchmark prove that all memory is allocated on the first node (node0). According to *numastat,* memory was allocated on all eight nodes.

Furthermore, the results of the *numa_maps* file also prove that memory is allocated on the first node whilst we use the *numactl* command. However, when we use the *numactl* command the Linux OS allocates almost double memory pages than when the application is executed without the *numactl* command. A cause for the extra allocation could be the way *numactl* is implemented. *numactl* might allow the original allocations to occur. Then it might use the kernel API *move_pages()* to move the allocations from one node to another. This would explain why more allocations are taking place, since the *numa_maps* file would also include any inactive allocations.

However, we cannot explain the fact the duration of the benchmark is less when using *numactl* compared to the modified version. We modified the CG benchmark when it tries to initialise data and the initialisation is part of the timed portion of the execution. We did find that the *numastat* tool is not applicable for testing how much memory an application has used. However, *numactl* is used in many papers regarding NUMA. The proper tool is to parse the *numa_maps* file and use the data it provides. Furthermore, since memory allocation could change during the runtime of an application, the *numa_maps* file should be constantly checked for updates.

As we use different nodes, the experiments returned different duration values, however these values confirm our analysis. As we move execution of the threads onto the central nodes, we achieve better (lower) execution times. The last three columns refer to an experiment that uses all sixteen threads of the X4600 platform. The duration of the experiment when we assign memory on the first node is 387 seconds which is longer than the duration of the modified version when we use only eight threads, which is 337 seconds. The original version of the CG kernel benchmark requires 49 seconds to complete, when sixteen threads are being used. It seems that as the number of threads (and the number of nodes) increases, the congestion of the integrated memory controller on node0 is causing the slowdown of the application.

This experiment proves that allocating memory on only one node in a ccNUMA platform leads to poor performance results. However, we did not expect that the performance would increase by seven times when all sixteen processors are used.

5. Results on Cray XE6 – AMD Magny-Cours platform

This chapter describes the experiments and analyses the results of the AMD Magny-Cours platform. AMD Magny-Cours platforms are used as the basis of the HECToR HPC system. HECToR (High End Computing Terascale Resources) is the UK's national supercomputing service. The supercomputer is contained in 20 cabinets and comprises a total of 464 compute blades [31]. Each blade contains four compute nodes, each with two 12-core AMD Opteron 2.1GHz Magny Cours (Greyhound - 6172) processors. Each 12-core processor shares 16 Gb of memory.

Each AMD Opteron core has two dedicated levels of cache and a shared level. Level 1 cache consists of a 64 KB 2-way associative instruction cache and a 64 KB 2-way associative data cache. Level 2 cache is a 512 KB exclusive 16-way associative cache. Level 3 is a 6 MB cache. The Magny Cours processor uses the level 2 cache as a victim cache. A victim cache is used to store data elements that were replaced from the level 1 cache. It lies between level 1 cache and level 3 cache and it tries to reduce the number of conflict misses [27]. The level 3 cache is also used a victim cache for level 2 cache. However, a portion of level 3 cache is used to speed up the MOESI implementation. Therefore, the effective Level 3 cache size is 5 MB. The option of using 1 MB as HT Assist is available in the BIOS and can be modified. As the number of cores starts to increase the overhead of point-to-point communication for every cache miss is building up. Therefore, AMD is using 1 MB out of the 6 MB of level 3 cache to store the data states of level 1 and 2 caches. When a snoop message is received from another node, the message is not required to reach every core, instead a response is sent at the node level and the node then rearranges the data states. However, the rearrangement takes place internally; thus the overall communication is reduced. This technique is called HT Assist by AMD. The processor chip also has an integrated dual-channel DDR3 memory controller, which controls the main memory that is directly attached to the processor.

The Magny Cours processor has 4 16-bit links, clocked at 3.2 GHz. Each processor is connected via HT links with the others. A conceptual design of a single twelve-core AMD processor is shown in Figure 24 and a complete twenty-four core node is shown in Figure 25.

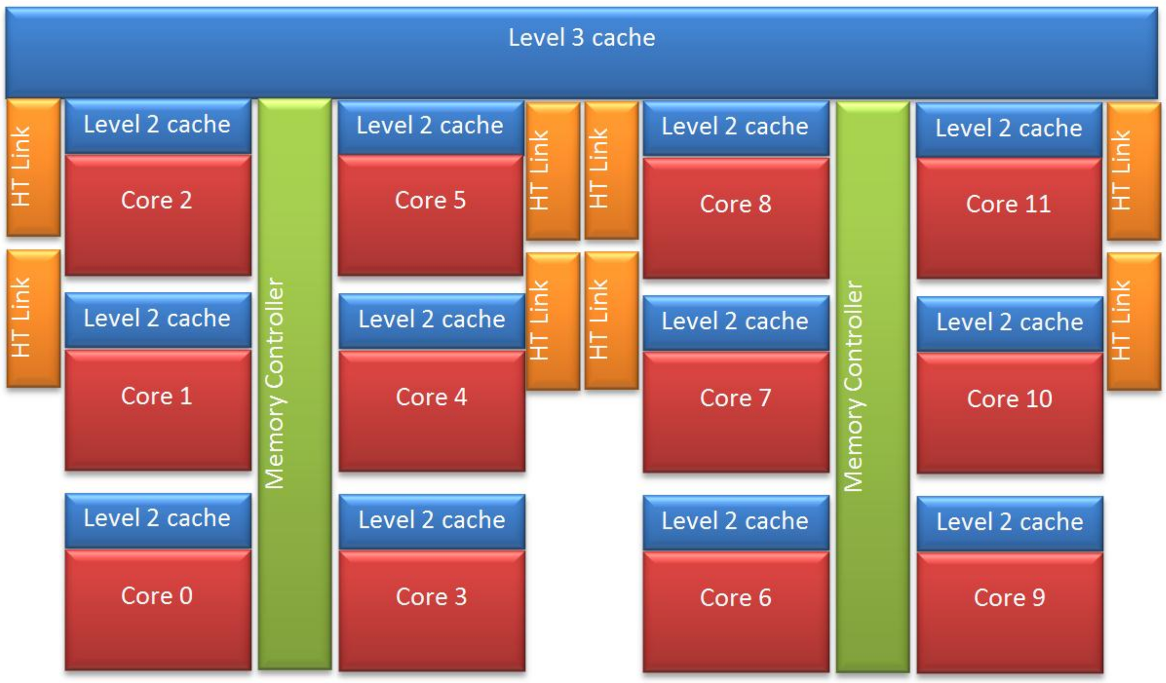

Figure 24 - A single twelve core AMD Magny Cours chip, from [32]

The Magny-Cours platform employs a modular design. The building blocks include the six core die, the shared LLC, four HyperTransport links and two DDR3 memory channels. These components are put together to create the twelve core chip and two chips are put together to create the Magny-Cours twenty-four chip multiprocessor (CMP). The Magny-Cours tries to excel in two different market domains. Packaging many cores in the same volume and in the same power envelope, it tries to dominate the data centre market. Data centres can easily replace four AMD Opteron platforms with one Magny-Cours. The Magny-Cours platform support hardware virtualization, thus data centres are able to allocate computing resources amongst twenty-four cores platforms, instead of sharing resources in six core platforms. Furthermore, AMD tries to win the HPC market by offering a platform with many cores, ideal for shared memory applications.

Each Magny-Cours processor employs many techniques in order to improve performance. It can execute instructions out of order. It can also fetch and decode up to three instructions during each cycle, as long as the instructions are in the instruction cache.

Each multichip module (MCM) holds two six core dies. Each MCM has four DDR3 channels, two DDR3 channels per six core die. The memory controller works along with the DRAM interface in order to provide concurrent DRAM request alongside local level 3 cache accesses. Therefore, if a level 3 cache miss occurs, the request to local DRAM has already been issued. This technique minimises latency due to level 3 cache misses. Finally, the MCM allows parallel DRAM and directory lookups in the LLC. This technique is similar to the previous one, however if a hit occurs in the directory lookup of any processor, the DRAM request is automatically cancelled.

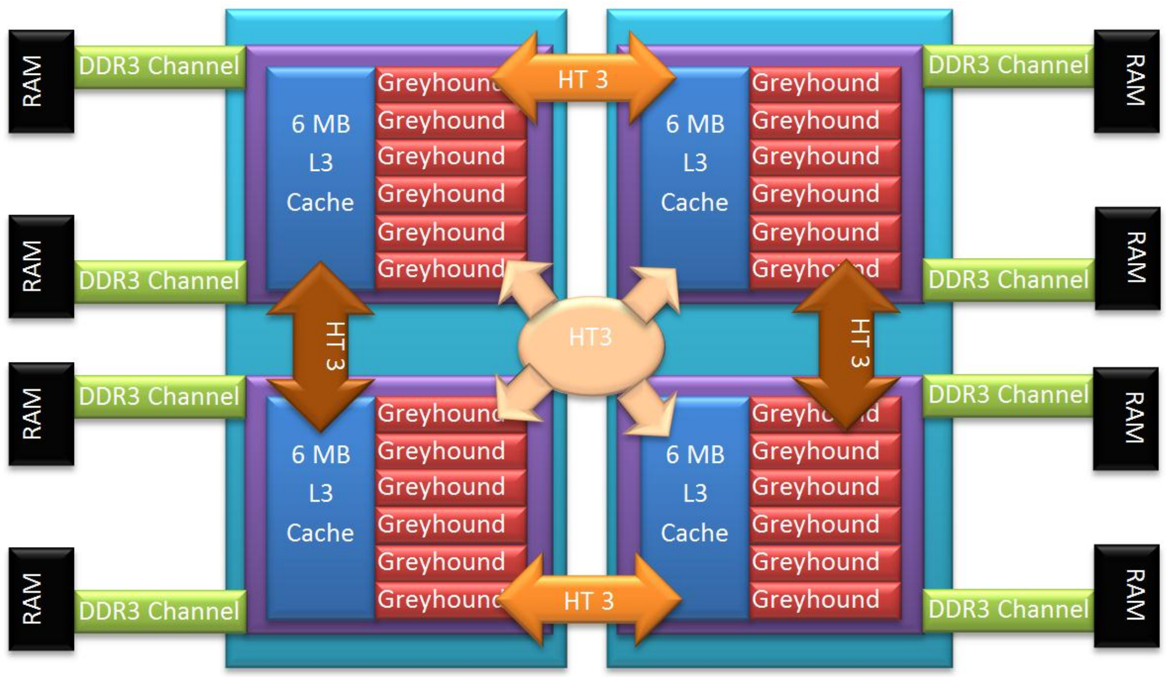

Figure 25 - A 24 core (node) AMD system, from [32]

As shown in Figure 25, the available bandwidth between the six core nodes varies. The HT bandwidth between two six core nodes on the same processor is 38.4 GB/s. The bandwidth between opposing six core nodes is a full 16-bit HT 3.1 link of 25.6 GB/s. Furthermore, the HT bandwidth between diagonally opposite six core nodes is a half HT 3.1 link, 8-bit link with 12.8 GB/s.

5.1 Cray XE6 – AMD Magny Cours Micro Benchmarks

The benchmarks described in Chapter 3 were run on HECToR. As mentioned earlier the HECToR system is build up from hundreds of AMD Magny-Cours blades. For the purposes of this investigation, only one node of twenty-four processors is required. Each HECToR node (which contains two twelve-core Magny-Cours chips) runs a special version of Linux, called Cray Linux Environment (CLE). CLE is built by Cray and acts as the operating system in high performance computing platforms. The main differences from other versions of Linux are that it does not interfere with the runtime of applications and it has a minimal CPU usage profile. Therefore, results from experiments run on HECToR are very uniform and easily reproducible.

Another irregularity of CLE is that it requires the use of the *aprun* command in order to use all 24 processors. a*prun* is a wrapper that sets the locality of memory and processor affinity based on command line arguments. If an application is executed without the use of the *aprun* command, it can only access the login node of HECToR and use only 6 processors. Therefore, the use of the *taskset* and *numactl* Linux tools is not required on HECToR. a*prun* automatically sets the appropriate processor bindings and memory locality options. The default allocation policy for the Magny-Cours platform is the first touch policy. The first touch policy allocates memory on the same node as the thread that requests the memory.

The first metric to be investigated is the read memory bandwidth. This metric will provide us with the raw local read memory bandwidth of each processor. As mentioned earlier the Magny-Cours platform has four NUMA nodes, each NUMA node has six processors. By measuring the local read memory bandwidth, we exclude any differences in read/write memory speed. Also by not modifying the array, we eliminate the performance impacts of the MOESI protocol. As in the previous benchmarks, an array is allocated on a memory segment controlled by a processor and a thread that runs on the same processor tries to access all the elements. The accessed element is copied in a temporary variable. In order to avoid the elimination of the temporary variable by the compiler, the variable is used in an *if*-test after the end of the loop. The duration of the loop is recorded and it is divided by the amount of copied data. Thus a MB/sec value is calculated. Compiler and processor optimizations will affect the runtime; however since these optimizations will take place on all processors the result will not be affected.

A ccNUMA platform views memory differently than an UMA platform. An UMA platform is able to access every part of global main memory without incurring any cost due to the distance of the processor requesting the memory access compared to the memory location. On a ccNUMA platform this is not true and an applications developer knows that accessing remote memory is more expensive than accessing local memory. This benchmark allocates and accesses only local memory, so we expect to see a constant uniform value, very similar to an UMA platform. Some variances are expected due to the cache coherence protocol or to OS activities.

By following the methodologies described in the previous chapter, we executed 3.1.1 Benchmark 1 five times. Another irregularity of the CLE environment is that it gets confused when compiler specific environment flags are present. Therefore it is essential to not use any PGI (or GNU gcc) environment option. In order to avoid the accidental usage of such variables the script file used was modified to specifically unset all PGI environment options. The syntax of the command was:

aprun –n 1 –d 1 –cc x bench.exe

x is the number of the processor that we measure. The benchmark was executed for an array of 2,000,000 elements or 32 MBs in size. The option "*–n 1*" means that only one process should be started. The option "*–d 1*" means that only thread should be created for that process and "*–cc x*" denotes the processor where the affinity should be set. The average value was used and is presented in Figure 26.

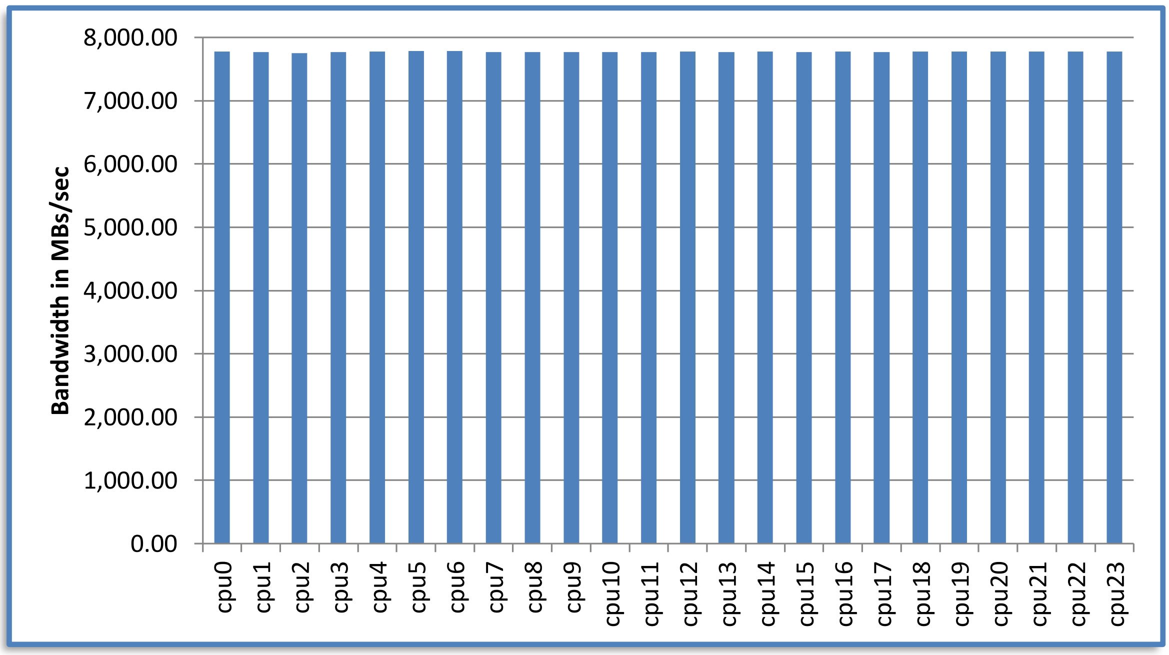

Figure 26 - Magny-Cours Read Bandwidth Measurements (32 MBs)

The graph shows a constant local read memory bandwidth, regardless of the processor that the benchmark is executed on. This consistency shows that the implementation of the NUMA memory model of this platform appears to function properly. The cache coherency protocol does not seem to affect the read memory bandwidth and the performance of the memory subsystem is not affected by the location of the memory, when only local memory is accessed. The results show that the read memory bandwidth benchmark is consistent and that the platform can achieve good performance when accessing local memory.

Furthermore, since no processor exhibits different performance levels, it seems that the CLE environment does not affect the memory bandwidth of applications. CLE stays out of the way of the running benchmark and does interfere at all.

Since the local read memory bandwidth benchmark did not produce any irregular results, a more interesting metric is the read bandwidth between two processors. The remote read memory bandwidth benchmark allocates memory on a single processor. It then reads the allocated memory using threads that are bound and execute on other processors. In order to bind the threads to specific processors we use the *aprun* tool. Furthermore the benchmark prints the hardware locality of the processor and memory location that is bound in over to verify the binding.

A ccNUMA platform exhibits variable bandwidth values when accessing remote memory. 3.1.1 Benchmark 1 will help us to quantify the performance hit that memory bandwidth will suffer if memory is allocated on a memory segment controlled by the first processor and is accessed by the other processors. Allocating memory on the first processor is a common problem for badly written multithreaded application. We hope that this benchmark will show us the performance impacts.

This metric will offer a glimpse of the NUMA effects on read memory bandwidth. In order to fix the processor affinity of the two threads involved in the benchmark we use the following command:

aprun –n 1 –d 2 –cc x,y bench.exe

x,y are the number of the processors that we measure. The benchmark was executed for an array of 2,000,000 elements or 32 MBs in size. The option "*–n 1*" means that only one process should be started. The option "*–d 2*" means that only two threads should be created for that process and "*–cc x,y*" denotes the processors where the affinity should be set. The benchmark was executed five times and the average value was used and is presented in Figure 27.

Memory is allocated in a memory segment controlled by the first node (node0) and the first processor (CPU0). Other threads, whose affinity is set on the other processors, then access the memory serially. The results show that the remote read bandwidth that processors physically located on node0 is the same as with the first processor. Therefore, all processors that are placed on the first node show the same memory bandwidth. Moreover, the value that this benchmark returns is slightly higher than the value that the local read memory bandwidth returned. This can be attributed to the fact that after the array is initialised, no cache clearing step is performed. Therefore, part of the array is still in the shared level 3 cache, which gives an extra boost to the performance of the local cores.

The memory bandwidth of the remote processors is 30% slower compared to the local memory access. This can be attributed to the cache coherency protocol, which sends probes across the HT links for every cache miss. The probes are sent for the first six processors, however the cache controller that holds the array data is found in the local cache; thus no data is actually send through the HT link. However when the array is accessed by remote processors, actual probes are send through the HT links slowing down the accesses. All NUMA regions require one hop to reach the other nodes; therefore, NUMA distance is always equal to one. However, the NUMA regions are connected through a combination of faster and slower HT links. The last six processors (CPU18 – CPU23, which are on node3) need to transfer data through a slower HT link. Thus their bandwidth values are 1% - 2% lower than the twelve processors (CPU6 – CPU17, which are on node1 and node2) which can use the faster HT links that are provided.

The small drop in the performance of the remote processors demonstrates that the Magny-Cours platform can handle remote memory accesses without exhibiting any irregular patterns.

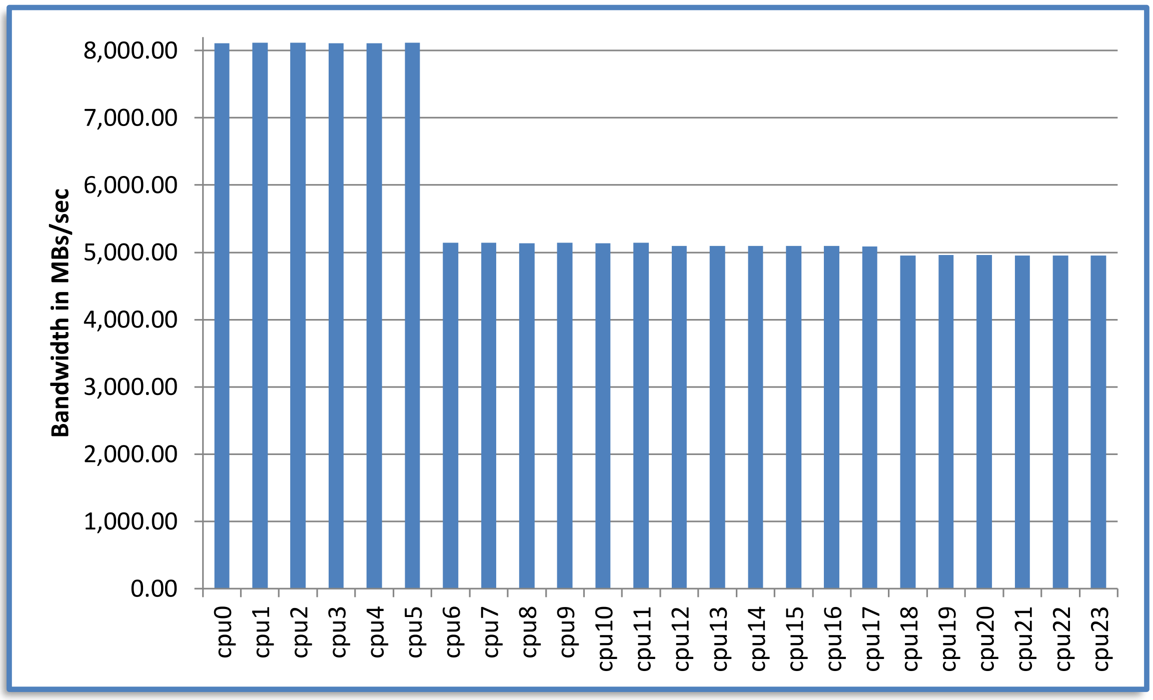

Figure 27 - Distributed Read Bandwidth on Magny-Cours (32 MBs)

Both bandwidth benchmarks returned expected results which collaborate our theories. The local bandwidth benchmark showed that each processor can access local memory without any drop in performance. The remote bandwidth benchmark revealed that remote memory bandwidth decreases by 30%, however there are no irregular patterns. Nevertheless the decrease in performance is expected due to the ccNUMA design. Since these two metrics did not expose any strange NUMA effects, we will investigate the effects of NUMA on local and remote memory latency.

The memory latency benchmark measures the time that a processor needs to complete a memory request. The memory request may lead to a local or remote cache, local or remote main memory. The location of memory is the significant factor that affects latency. Another factor that affects latency is the size of the requested memory. Small adjoining requests can sometimes trigger the prefetching algorithms of processors. If a processor detects that it tries to access continuous memory locations, it will try to pre-read and store in some level of cache data that lie ahead. If it guessed correctly the data will be in the faster cache memory even before the application has issued a request for them. This operation however gives a false impression for latency. Therefore our micro benchmark tries to fool the prefetching algorithms of modern processors by accessing memory using a pseudo-random algorithm and by touching only one element for every cache line. In order to execute the latency benchmark we use the following command:

aprun –n 1 –d 1 –cc x bench.exe

x is the number of the processor that we measure. The benchmark was executed for memory sizes up to 64 MBs in size. The option "*–n 1*" means that only one process should be started. The option "*–d 1*" means that only one thread should be created for that process and "*–cc x*" denotes the processor where the affinity should be set. The benchmark was executed five times and the average value was used. The results are presented in Figure 28.

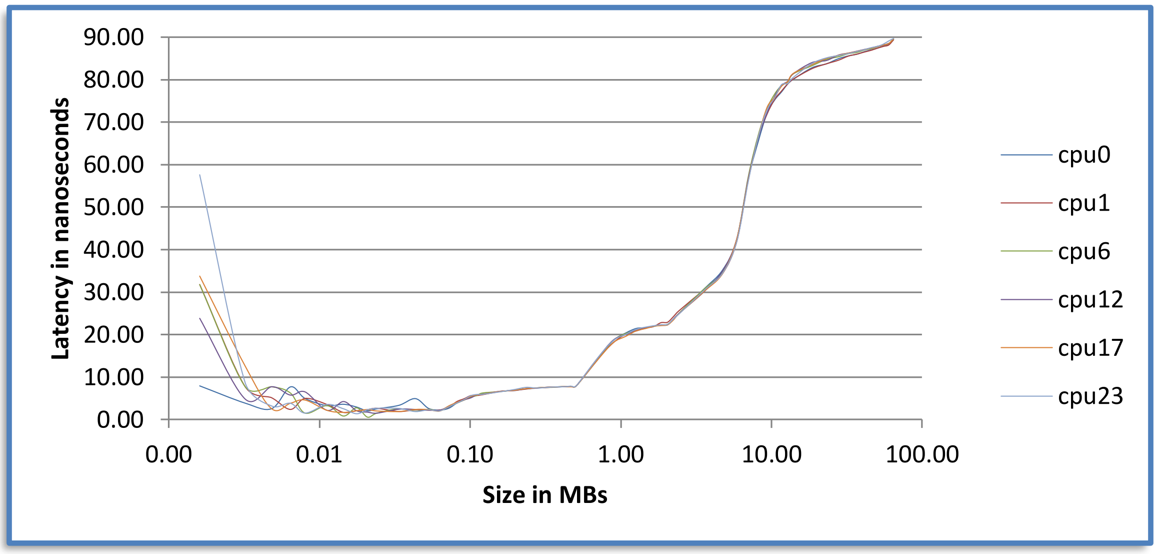

Figure 28 - Magny-Cours Latency Measurements

The size of the array (x-axis) is represented in logarithmic scale. For clarity and readability, we only present six processors; nevertheless all data points are the same across the twenty-four processors. The latency results show that the CLE environment is better suited for high performance applications than normal Linux versions, since it manages to avoid any irregular values on the first processor.

There are three clear steps in the previous figure. These steps coincide with the size of the three cache levels of the Magny-Cours platform. When the array is small enough to fit in the cache, the processor optimization routines prefetch the data and access times are low. However as the array increases in size and becomes larger than the cache, access times become higher and latency increases. Level 1 cache accesses are measured in the range of 2-3 nanoseconds. Level 2 cache accesses are measured in the range of 4-8 nanoseconds. Level 3 cache accesses are measured in the range of 20-28 nanoseconds. The values that we measure are very close to the values that AMD's design document for the Magny-Cours platform presents [33]. For very small array levels, the micro benchmarks return very high values. The reason of the very high latency values is that the loop count is very small. Thus, any wrong prediction made by the compiler regarding loop unrolling or data/instruction prefetching adds a significant percentage to the runtime of the benchmark. Prefetching data does not help, because the size of the array is a few cache lines and most of the prefetched data are useless, since we only access one element per cache line.

As stated earlier, each level 1 cache miss triggers the MOESI cache coherence protocol. The MOESI protocol sends signals to the other processors in order to find the requested data in a cache controller by another processor. In the local latency benchmark the data are always touched by the same processor. Therefore, the replies by all the other processors will be negative. However, the Magny-Cours can use up to 1 MB of the 6 MBs of level 3 cache in order to speed up this process. In essence, part of the level 3 cache is constantly updated with the status of every cache in the local chip. Thus when a snooping message arrive the integrated memory controller can very quickly figure out if the data belong to one of the local caches by looking at the directory which is kept in the 1 MB of cache. If the data are indeed present in a local cache, another snoop message is required in order to change the state value of that particular cache line according to the rules of the MOESI protocol.

The Mangy-Cours platform can selectively use 1 MB of level 3 cache as a cache directory (probe filter). A BIOS setting allows the use of the level 3 cache as a regular cache or a cache/probe filter. The probe filet functionality is also called HyperTransport Assist (HT Assist) and it helps to increase memory bandwidth and to decrease memory latency.

According to P. Conway et al. [33] the use of HT Assist increases memory bandwidth by 350% on a twenty-four core Magny-Cours system. Furthermore, it reduces latency to 40% for local RAM accesses and to 70% for remote RAM accesses (one hop away) compared to the same operation without HT Assist enabled. HT Assist is enabled by default on HECToR since it improves performance with a small sacrifice of 1 MB of level 3 cache.

The last results of our micro benchmarks suite are the distributed latency experiments. In order to verify if NUMA distance affects memory latency, we allocate memory on one node and then try to read all data elements from remote nodes. This benchmark will tell us if the number of hops that are required influences latency and it will also show us the performance impacts that a multithreaded application will suffer, if it allocates all memory on one node instead of allocating memory on all nodes.

The Magny-Cours platform is built with scalability in mind and only one hop is required to access all nodes. Therefore, we expect that the NUMA effects will be minimal on this platform. HT Assist is also enabled, which should reduce memory latencies. All memory for this benchmark was allocated on the first node and on the first processor (node0, CPU0).

In order to execute the latency benchmark we use the following command:

aprun –n 1 –d 2 –cc x,y bench.exe

x,y are the numbers of the processors that we measure. The benchmark was executed for memory sizes up to 64 MBs in size. The option "*–n 1*" means that only one process should be started. The option "*–d 2*" means that only two threads should be created for that process and "*–cc x,y*" denotes the processors where the affinity should be set. The benchmark was executed five times and the average value was used and the results are presented in Figure 29.

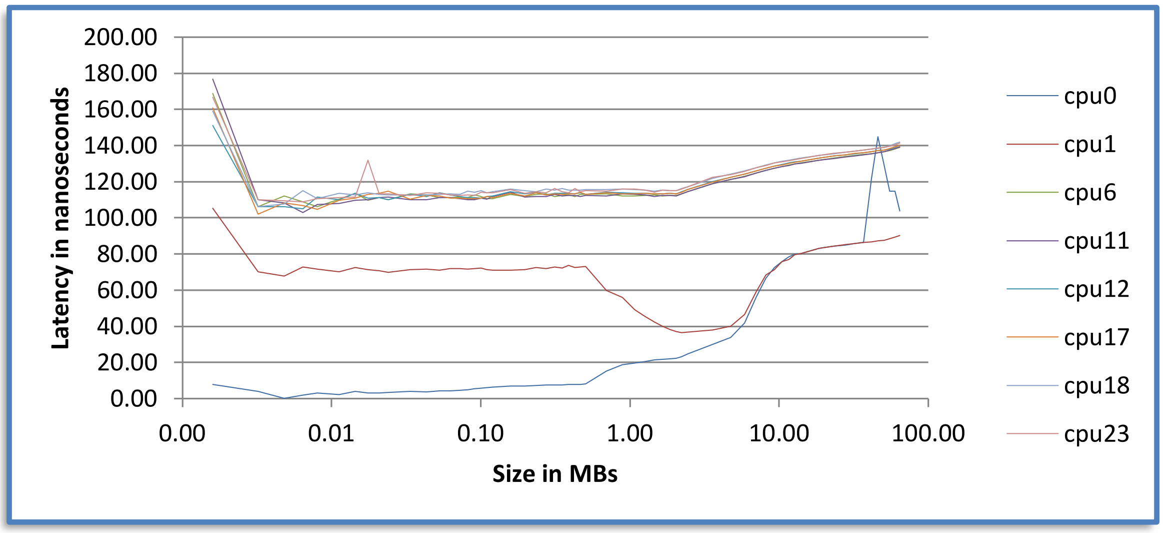

Figure 29 - Magny-Cours Distributed Latency Measurements

The distributed latency results illustrate that the Magny-Cours platform does not show any strange NUMA effects. Remote latency is slower than local however this was expected. Remote latency rose from 2-3 nanoseconds to 105 nanoseconds for remote level 1 cache accesses, from 4-8 nanoseconds to 110 nanoseconds for remote level 2 cache accesses and fro 20-28 nanoseconds to 115 nanoseconds for remote level 3 cache accesses. Latency for accessing remote main memory increased from 85 nanoseconds to 135 nanoseconds.

An interesting finding is that memory latency for CPU1 also rose. CPU1 is on the same node as CPU0. One would expect that processors that share the same silicon would be able to access data equally fast. However all processors on node0 suffer from the same decreased memory latency compared to CPU0. As we have previously discussed the Magny-Cours platform has three levels of cache. Each processor has a dedicated level 1 cache that is 64 KBs in size. It also has a dedicated level 2 cache that is 512 KBs in size. Furthermore the six processors that are on the same die, share a 6 MBs level 3 cache, where only 5 MBs are used as cache and 1 MB is used as probe filter (HT Assist). According to the design of the benchmark, memory is allocated on CPU0. However, we do not flush the caches. Therefore for small arrays the data remain in the level 1 cache (or level 2 cache) of CPU0. Since the data are not resident on CPU1, when CPU1 tries to access the data, it triggers a level 1 cache miss. This causes a MOESI snoop message to be sent from CPU1 to the other nodes and the memory controller of node0. Node0 tries to locate the data using the directory space of the level 3 cache. It then needs to move the data from the cache of CPU0 to the cache of CPU1 and in the process it needs to invalidate all moved cache lines and update the directory on the shared level 3 cache. All this operations cost time and the result is an increase of the local latency from 2-3 nanoseconds to 67 nanoseconds for remote level 1 cache accesses, from 4-8 nanoseconds to 71 nanoseconds for remote level 2 cache accesses and fro 20-28 nanoseconds to 35 nanoseconds for remote level 3 cache accesses. However accessing local main memory remained steady at 86 nanoseconds.

The increase for level 3 accesses is small and according to the graph latency seems to drop off as the size of the array increases. The reason is that as the size of the array increases beyond the level 2 cache size, some data are moved to level 3 cache. Since level 3 cache is a victim cache, when the array becomes larger than the level 2 cache, data are evicted to level 3 cache. Therefore, after CPU0 creates the array in memory, some data reside in levels 1, 2 and 3. So when CPU1 tries to access the data it gets a hit by accessing data that are in level 1 and 2 cache of CPU0, however all the data that are in the shared level 3 cache are free to use.

The final metric that we want to measure is contention over the HyperTransport links and on the integrated memory controller. In order to measure this NUMA effect, we allocate memory on the first processor and try to read from the same memory using all twenty-four processors. This experiment will cause many snoop messages to be sent along the processors. it will also test the performance of the IMC of the first processor. Since the Magny-Cours uses the HT Assist probe filter, we expect that the results will be similar to the previous experiments. Furthermore, we varied the size of the array in order to investigate if the size of the array affects the performance. The results are presented in Figure 30.

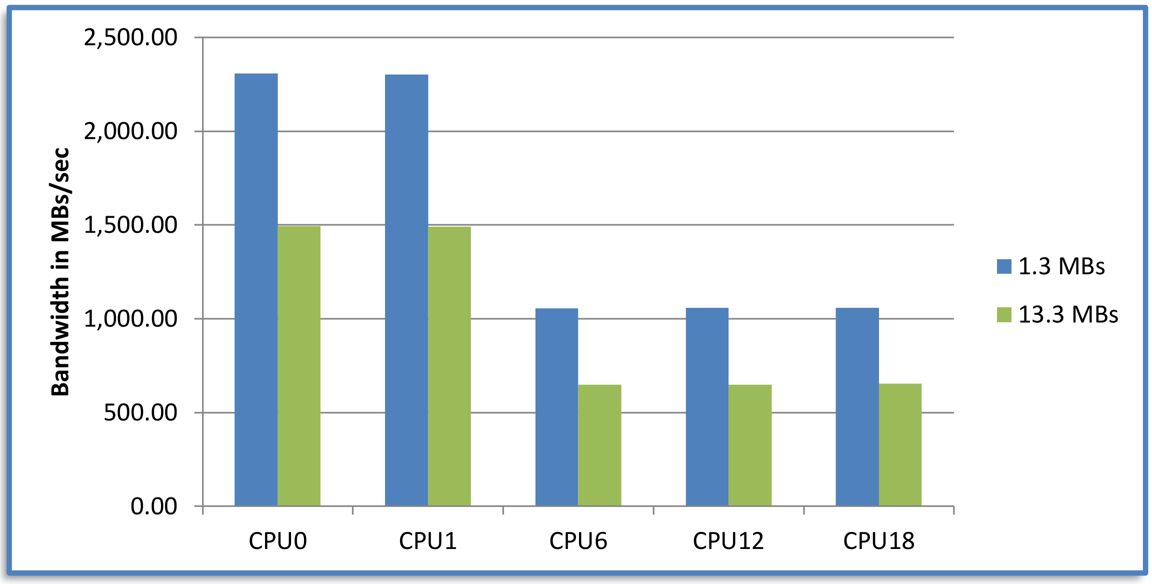

Figure 30 - Magny-Cours Contention Measurements

We used two different values in order to check the bandwidth results. It seems that the size of the array affects the overall bandwidth. For small arrays that can fit in the shared level 3 cache, bandwidth values are higher. However, when we increased the size of the array to 320 MBs (or 13.3 MBs per thread) the values stabilised.

Memory is allocated on the first processor (CPU0). We use the *aprun* command to set the affinity of the threads. It seems that the first processor is the fastest. This behaviour is expected, since the first six cores are accessing local memory. Furthermore, the first six processors share the same bandwidth, since they share the same IMC and the same level 3 cache. As we move away from the local six core die, the other processor feel the contention for the same memory location. The cache coherence protocol that the Magny-Cours uses is a broadcast protocol. Therefore, it is safe to assume that during the rush to access the same memory location, the memory controller of node0 will have a hard time deciding which request to serve first.

Performance drops from 7.7 GBs/sec for accessing local memory to 2.3 GBs/sec. Furthermore, remote read bandwidth drops from 5.1 GBs/sec to 0.6 GBs/sec, an 88% drop in performance. This benchmark tries to quantify the extreme scenario where all processors try to access the same location. This scenario could only happen if the application is designed to allocate memory on a single location, without using parallel directives to split the allocation and initialisation of memory. However, it is useful to observe how far performance can drop when many processors try to request access to the same memory location.

5.2 Cray XE6 – AMD Magny Cours Applications

The results of the previous benchmarks showed that the Magny-Cours platform is impacted by NUMA effects. However, the impacts are small and their causes are understood and are based on hardware limitations of the processor and the memory subsystem. The results of experiments with two kernel benchmarks from the NPB suite, CG and MG, are presented in this section. These benchmarks were used because they stress different characteristics of the memory subsystem. CG stresses memory latency, whereas MG stresses memory bandwidth.

The experiments are performed in order to validate our analysis and theories of the different levels of performance due to NUMA effects. For our experiments, we will use classes B and C. Class B problems are medium sized whereas class C are larger problems and take more time to compute. Both benchmarks are developed by NASA and are highly optimised. The initialisation of data is taking place in parallel; therefore, both benchmarks avoid allocating all memory on the first processor. We hope to show that allocating all data on one node is disastrous for performance and should be avoided at all costs.

The results of experiments using two, four and six processors and the CG.B benchmark are presented in Figure 31. Only a subset of results is presented. An exhaustive list of all possible runs was impossible to obtain. The *aprun* command is used in order to execute benchmarks on HECToR. Even though the *aprun* command has many configuration options [34], it lacks the functionality to place memory on a specific node. Therefore all our experiments will use the "-cc" option which binds threads to specific processors. The default allocation policy on the XE6 nodes is local allocation. We therefore anticipate that the allocation of memory will indeed take place on the local nodes. Since the overall memory footprint of the benchmarks is smaller than the available memory we expect that the allocated memory will not overfill to neighbouring nodes.

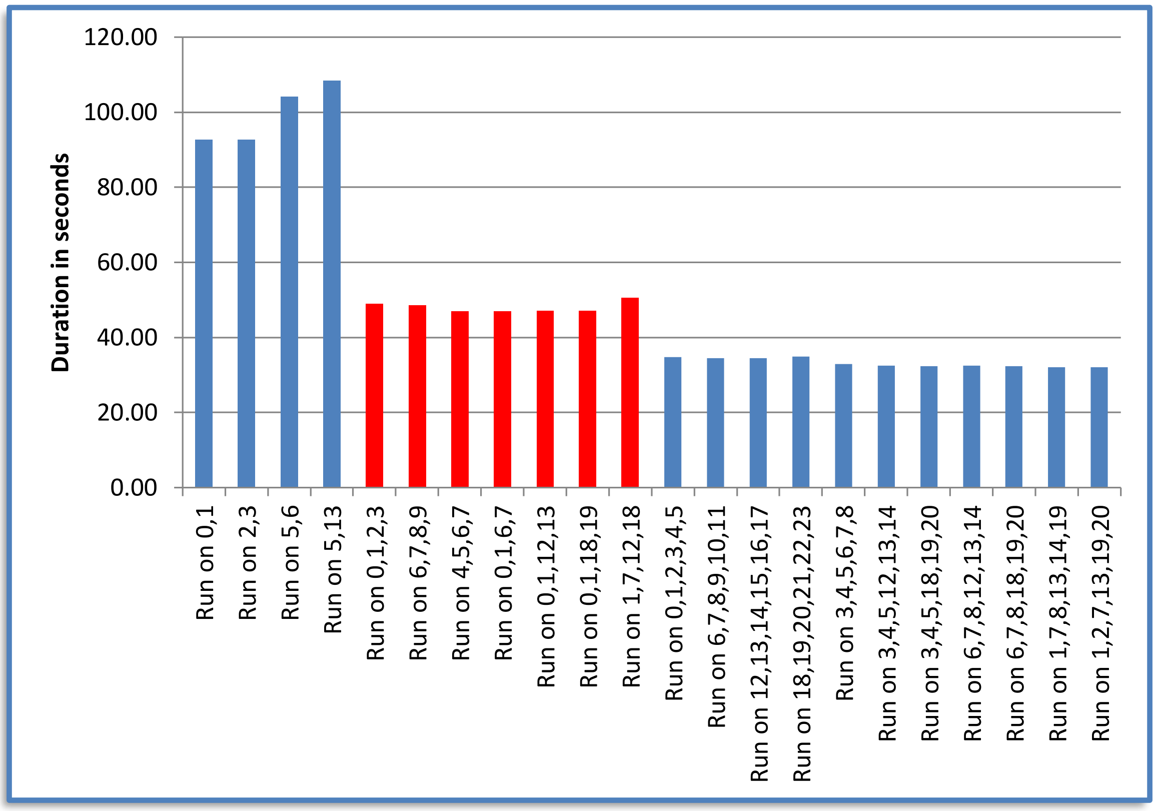

Figure 31 - CG (class B) using 2, 4 and 6 cores on Magny-Cours

The first four columns represent experiments using two threads. When two threads are used, the best allocation of threads (and memory) is on the same node. In order to force CLE to place the executing threads on specific processors we used the following command:

aprun –n 1 –d 2 –cc x,y ./cg.B.x

x,y are the numbers of the processors that we measure. The benchmark was executed five times. The option "*–n 1*" means that only one process should be started. The option "*–d 2*" means that only two threads should be created for that process and "*–cc x,y*" denotes the processors where the affinity should be set.

When we place the threads on different nodes (CPU5 is on node0 and CPU6 is on node1) the duration of the benchmark increases. The worst possible affinity setting is to use diagonally opposed nodes. These nodes use the slower HT link to communicate, thus the memory latency is higher. The use of the diagonal nodes increases the duration of the benchmark from 92 seconds to 108 seconds. This increase in duration constitutes a drop in performance by 19%.

However as we increase the number of threads to four, we notice that performance increases when we use more than one nodes. The next seven (red) columns show the results of our experiments using four threads. The activity of four threads that access main memory at the same time is enough to saturate the integrated memory controller and the two DDR3 memory channels of one node. By using two nodes, we effectively double the raw memory bandwidth. Even though we take a hit from increased latencies between the nodes, the CG.B benchmark shows better performance. However, the increase in performance is small. Duration increases from 46.9

seconds to 49.01, a drop of 5%. Even if we select the diagonal nodes to place the four threads performance of the CG.B benchmark rises from 46.9 for the best allocation to 47.1 for the allocation on diagonally opposite nodes. The worst possible allocation is to use four distinct nodes. The last red column shows the result of running the CG.B benchmark on a 1/1/1/1 configuration, using only one processor per node. Since the CG benchmark stresses memory latency, using four nodes increases memory latency and reduces the overall performance of the benchmark.

The next eleven columns show the results of experiments using six threads. We allocated the threads on the same node, using 3/3, 4/2, 1/2/2/1 and 2/1/1/2 configurations on various nodes. The performance of the CG.B benchmark does not vary much and the fastest run is at 31.5 seconds. The worst and best durations vary by 9%. The worst executions are when we allocate all threads on the same node. From the previous experiments, we know that even for latency controlled benchmarks, allocating on the same node hurts performance. The 2/1/1/2 configuration, where we allocate two threads on the first node, 1 on the second and third nodes and two threads on the fourth node reduces duration to 32.1 seconds. The 1/2/2/1 and 2/1/1/2 configuration differ by less than 1%, therefore we can assume both of them perform equally well. However, the best allocation is a more sensible 2/2/2 configuration that reduces duration to 31.5 seconds. By using the correct number of nodes, we manage to use the increased bandwidth and to reduce latency. Using more nodes, causes latency to rise and using fewer nodes causes a saturation of the available bandwidth.

Figure 32 shows the results of our experiments using CG.B on eight, twelve, eighteen and twenty-four processors. The first eight bars show the results of executing CG.B using eight threads. When we use eight threads, the worst possible allocation is to fully subscribe one node and use two processors from the next. Using a 4/4 configuration we improve performance by 5%, compared to a serial allocation of threads. When we use a 2/2/2/2 configuration we improve performance by 9% compared to the original duration. As we showed earlier, more than two threads of the CG benchmark can easily saturate the memory controller of a node. Furthermore, intra-node latencies (which the CG benchmark tests) are superior to inter-node latencies. By using multiple nodes, we spread the cost and all nodes use both their memory channels at full capacity. We also reduce memory latencies, since intra-node latencies (which the CG benchmark tests) are superior to inter-node latencies.

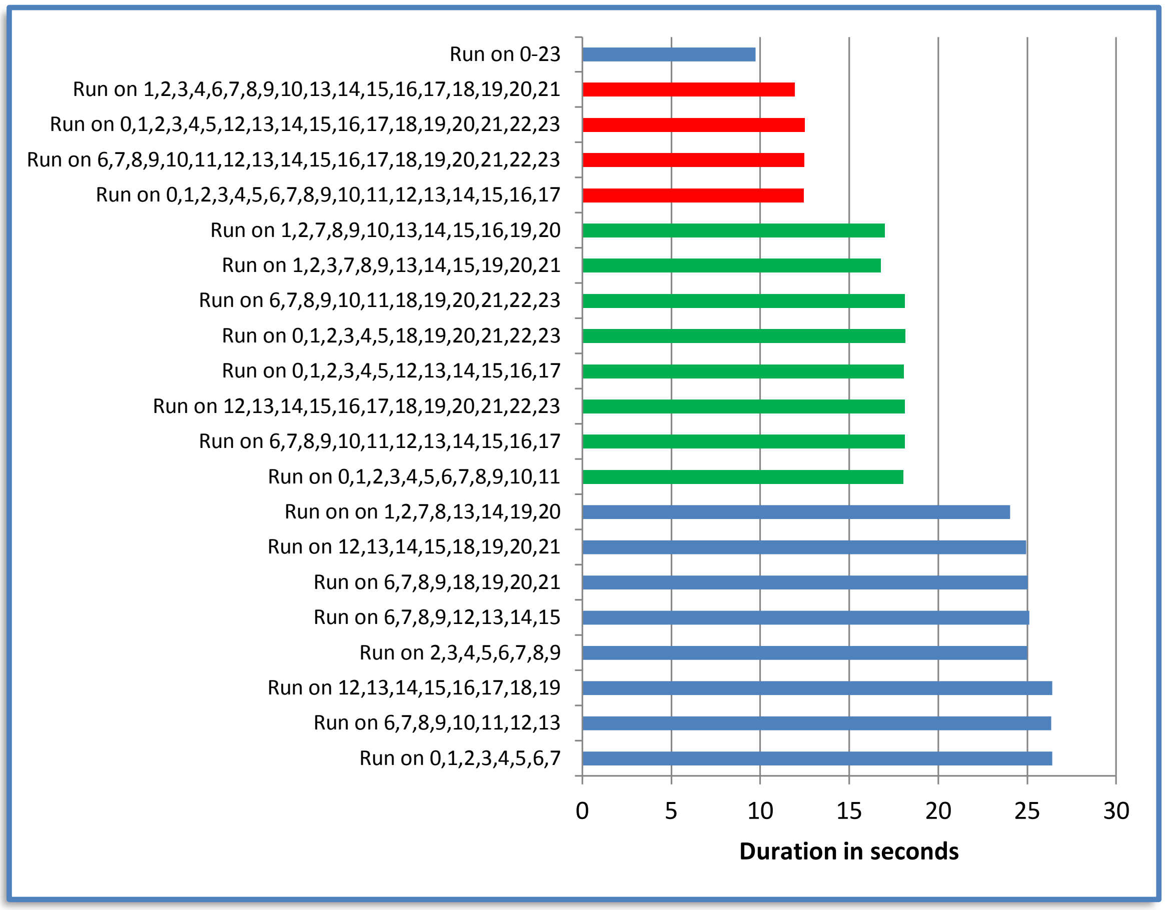

Figure 32 - CG (class B) using 8, 12, 18 and 24 cores on Magny-Cours

The twelve thread experiment also performs better when we under subscribe nodes. The 6/6 configuration runs for 18.05 and 18.12 seconds. When we use neighbouring nodes, the benchmark runs for 18.05 seconds and when we use the diagonal nodes, the benchmark runs for 18.12 seconds. However when we use a 2/4/4/2 configuration we manage to lower the benchmark duration to 17 seconds. When we use a more traditional 3/3/3/3 configuration we achieve maximum performance, with a duration of 16.77 seconds. Therefore, by under subscribing the threads we managed to increase performance by 7.5%.

The eighteen thread experiment only sees a small increase in performance. However, we do not have many options on allocating threads. When we fully subscribe three out of four nodes, the benchmark runs for 12.46 seconds. When we use a 4/5/5/4 configuration we manage to reduce the duration to 11.92 seconds, a small increase in performance of 4.6%.

As more processors are used, the benefit of assigning threads onto processors diminishes. Figure 33 clearly shows that the performance improvements steadily decrease, as we increase the number of threads.

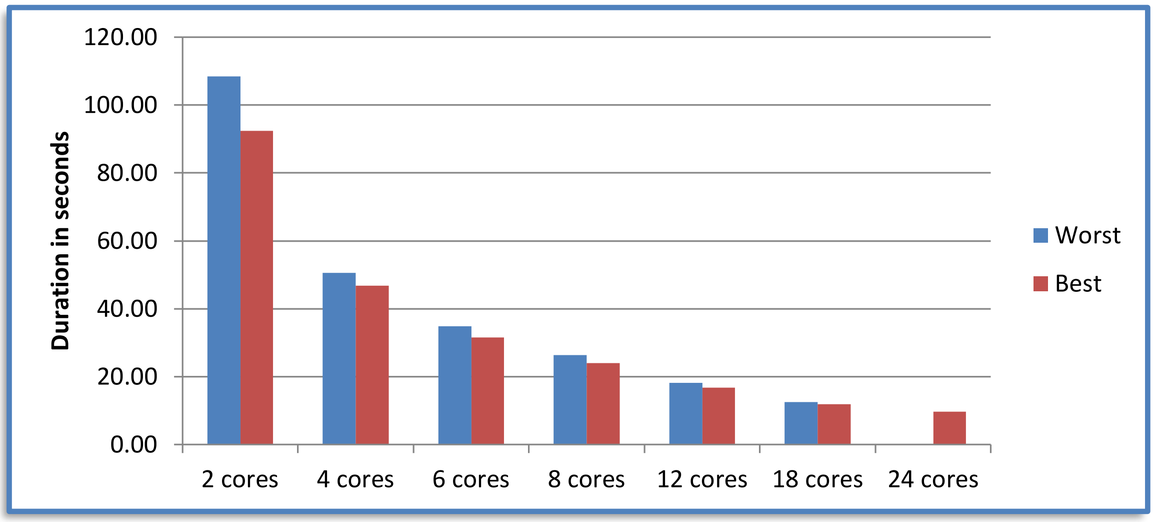

Figure 33 - CG.B comparison on Magny-Cours

The results for the CG.C benchmark are similar to CG.B. When we use four thread the 1/1/1/1 configuration results in a 159.6 seconds run, whereas a fully subscribed node results in a 147.8 seconds run and a 2/2 configuration gives as the best performer which is a 140 seconds run.

The six threads experiments perform badly on fully subscribed nodes with durations of 108.8 seconds. When we use 3/3 configurations we improve performance and reduce duration to 96.5 seconds. However, when we use more than two nodes with 1/2/2/1 or 2/1/1/2 configurations we increase duration to 99.8 seconds. Finally, the best run was achieved with a more sensible 2/2/2 configuration that reduced duration to 94.5 seconds, improving performance by 13%.

As we use more than eight threads, the configurations that work better are the same as the ones for the CG.B benchmark. Therefore, for eight threads we increase performance by 12.8% when we use a 2/2/2/2 allocation. Twelve threads results are improved by 11.4% when we use a 3/3/3/3 configuration. Finally, the eighteen threads results are improved by 5.1% by using a 4/5/5/4 configuration.

Figure 34 shows the best and worst thread placements on various experimental runs. As the number of threads increases, the performance benefits decrease.

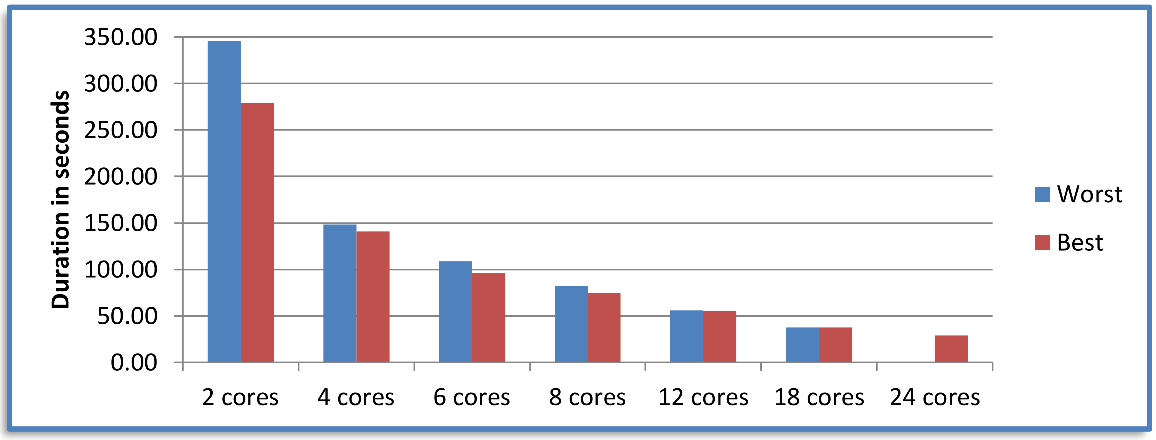

Figure 34 - CG.C comparison on Magny-Cours

The results of experiments using two, four and six processors and the MG.B benchmark are presented in Figure 35. Only a subset of results is presented. An exhaustive list of all possible runs was impossible to obtain. From the results for the CG benchmarks, we expect that for fewer threads the best placement should be on the same node. However, we expect that as the number of threads is increased, the threads should be placed on different nodes in order to fully utilise the two memory channels that are present on each six-core die.

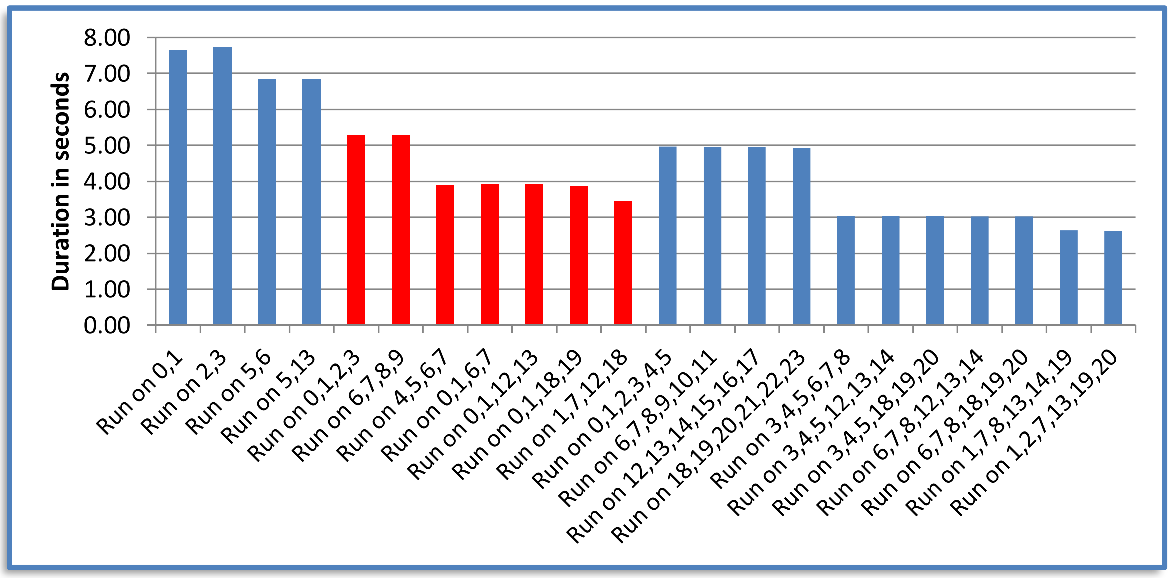

Figure 35 - MG (class B) using 2, 4 and 6 cores on Magny-Cours

The results of the MG.B benchmark show that for two threads the best placement is on different nodes. This seems to contradict our theories and is the exact opposite result compared to the CG.B benchmark that we investigated previously. When we place the two threads on the same node, the duration of the benchmark lasts for 7.7 seconds compared to 6.8 seconds for placing on different nodes.

The same is true for the four threads experiments, which are shown as red columns. When we place the four threads on the same node, the duration is 5.3 seconds compared to 3.9 seconds for placing the threads on different nodes in a 2/2 configuration. However, if we place the threads on four different nodes in a 1/1/1/1 configuration we achieve the best performance with a drop in duration to 3.45 seconds. The performance improvement the four threads MG.B experiment is 34%.

The MG kernel benchmark is memory bandwidth intensive. It seems that only one thread of the MG benchmark fully utilises the two DDR3 memory channels of the Magny-Cours platform. This finding is collaborated by the fact that when using two threads the faster execution occurs when the two threads are placed on neighbouring nodes. By spreading the threads on multiple nodes, we manage to multiply the raw bandwidth throughput of the platform. The trade-off between remote nodes (higher bandwidth) and same node (lower latency) favours the higher bandwidth. The six thread experiments also support this explanation, since the remote placement of threads in a 3/3 configuration yields a 39% improvement in runtime, from 4.9 seconds to 3 seconds. However, when we place the threads in an unconventional 1/2/2/1 or 2/1/1/2 configuration we manage to reduce duration to 2.62 seconds with a performance improvement of 47%. Using a 2/2/2 configuration produced the same results as the 1/2/2/1 or 2/1/1/2 configurations.

The results of experiments using eight, twelve, eighteen and twenty-four threads are presented in Figure 36. The placement of the eight threads measure performance improvements up to 28%. By using a 4/4 configuration in splitting the threads instead of a 6/2 we manage to drop the benchmark duration from 3.7 seconds to 2.6 seconds. However a 2/2/2/2 configuration managed to reduce duration to 1.98 seconds, yielding performance improvements of 46%.

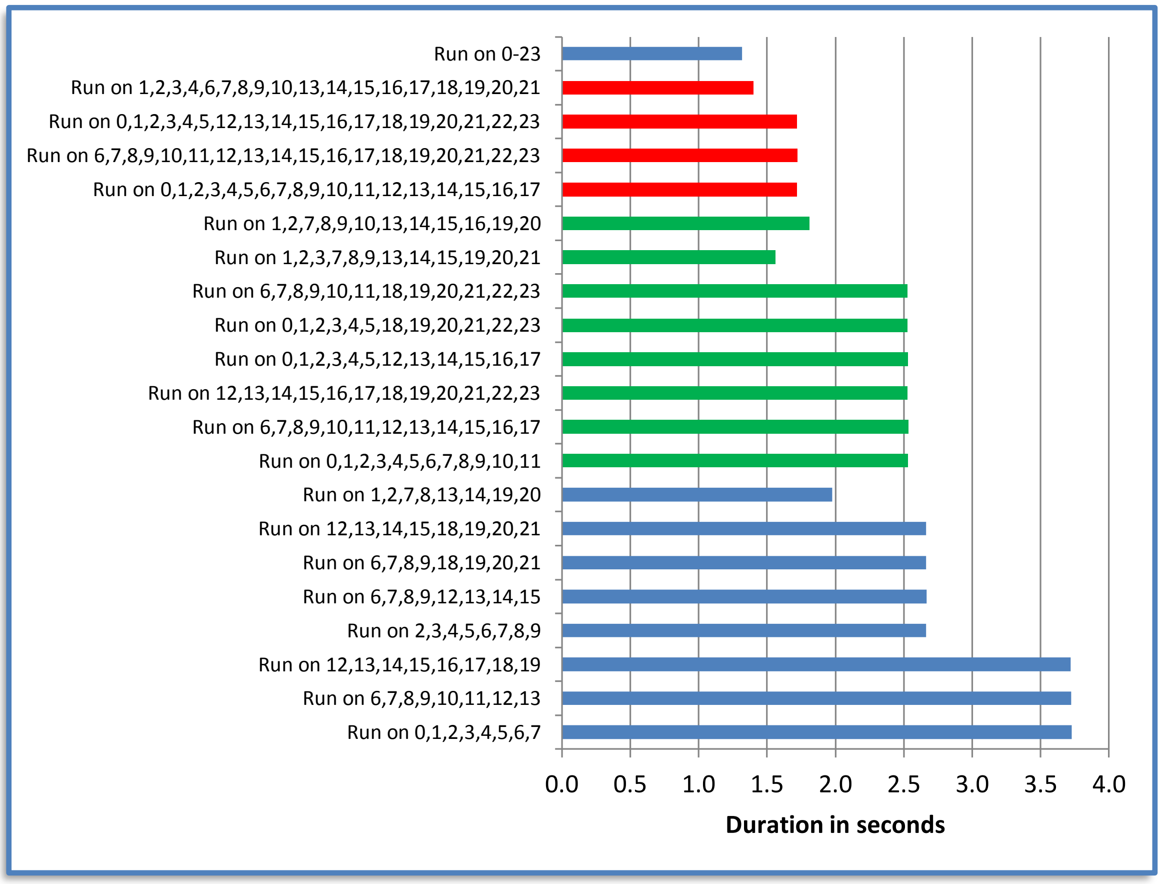

Figure 36 - MG (class B) using 8, 12, 18 and 24 cores on Magny-Cours

The twelve thread experiment exhibited an almost constant performance when we used a 6/6 configuration. The benchmark timed durations of 2.5 seconds. However when we divided the threads in a 3/3/3/3 configuration we managed to reduce duration to 1.56 seconds and increase performance by 38%.

The eighteen thread experiment also exhibited a constant duration regardless of the placement of the threads. We used the sensible approach of fully subscribing three nodes with threads, a 6/6/6 configuration. The result was a steady value of 1.72 seconds for the duration. However when we used a 4/5/5/4 configuration for the subscription we reduced duration to 1.4 seconds increasing performance to 19%.

The performance improvements of the MG kernel benchmark are impressive. The actual cause of the improvements is the fact that the MG benchmark is memory-bandwidth intensive. We used the same thread placement for the CG benchmark and even though performance increased, the gains were not as high. In some cases, especially for fewer threads, performance decreased. Therefore, before modifying the thread placement policy, the performance characteristics of the application must be studied and be well understood. As the number of threads increases, the best configurations start to converge.

The MG.C benchmark shows the same improvement characteristics as we modify the allocation of the threads. Performance improvements do not reach values as high as 47%, however we manage to improve performance by 31% when using four threads, 36% when using six threads, 34% for eight threads, 27% for twelve threads and 11% for eighteen threads.

The next two figures (Figure 37 and Figure 38) show the best and worst executions of MG.B and MG.C on the Magny-Cours platform. The improvements in performance are greater than the CG benchmarks; however, the same trend is evident. As the number of threads reaches the number of available processors, performance stabilises and thread allocation becomes a smaller factor.

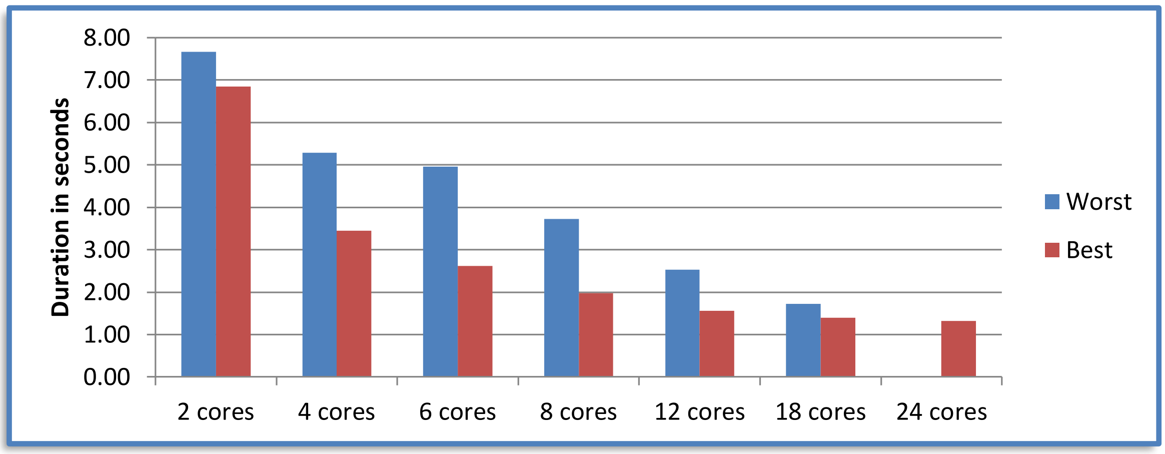

Figure 37 - MG.B comparison on Magny-Cours

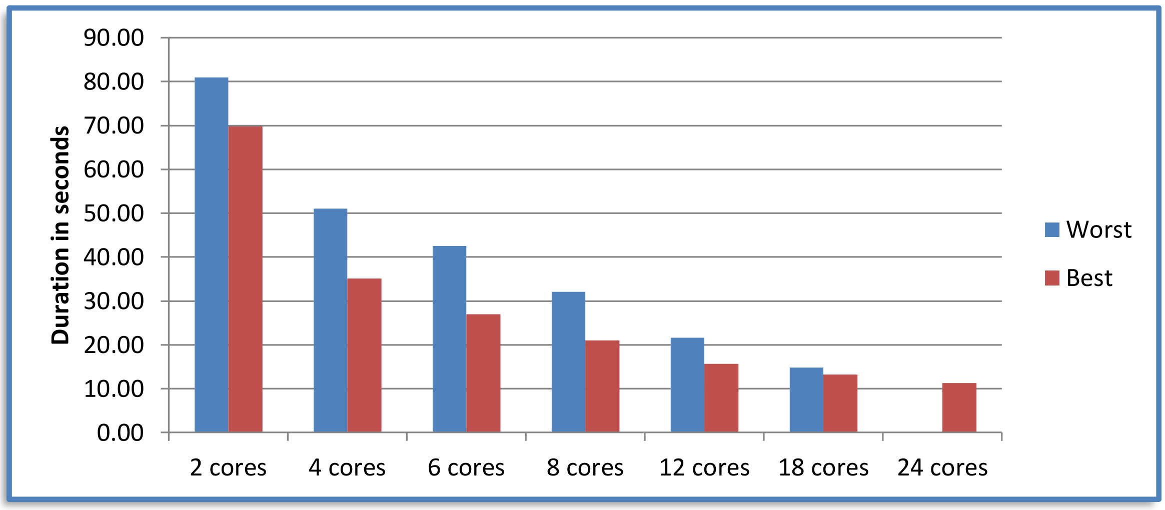

Figure 38 - MG.C comparison on Magny-Cours

Unfortunately, the *aprun* tool does not allow us to restrict memory allocation on a specific node. Therefore, we turn to our modified version of the CG benchmark. In order to measure the effects of allocating memory on a single node when twenty-four threads are used, we modified the source code of the CG benchmark. The CG benchmark runs an untimed initialisation run before the actual benchmark starts to execute. This run is used to allocate memory and to set the initial values. Due to the first touch policy, which is the default policy of the Magny-Cours platform, memory will be allocated on the node that executes the allocating thread. We removed all OpenMP directives from the initialisation run, therefore the main thread, which is bound to the first processor, performs every allocation and first touch.

The results of the modified version compared to the original version are presented in Figure 39. The columns labelled "LOCAL" and "ORIGINAL" depict the duration times of the modified and of the original version respectively.

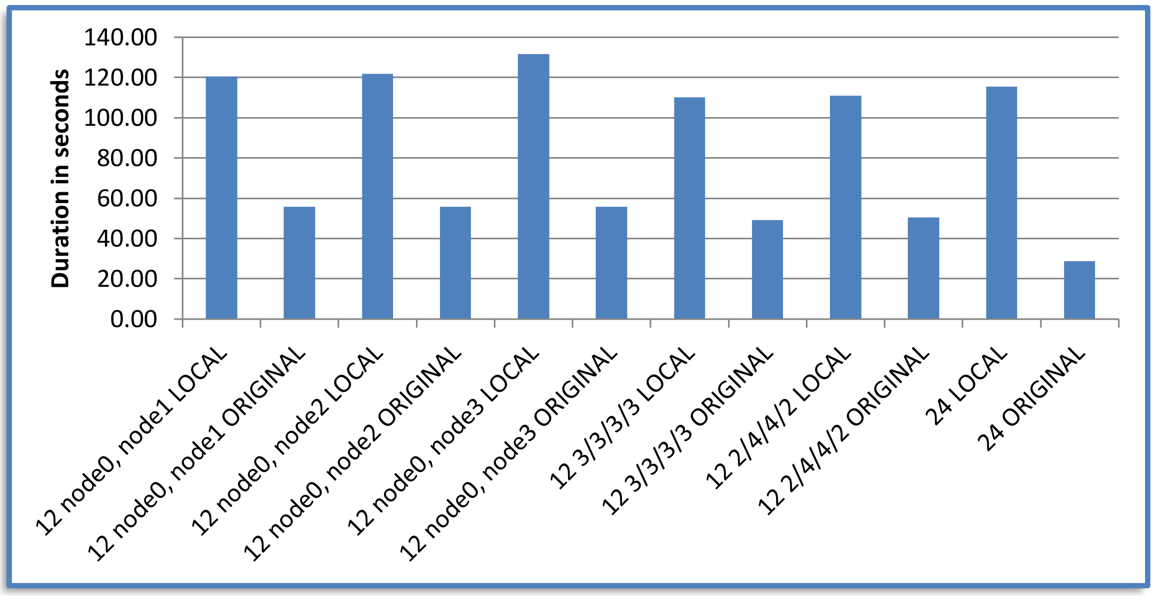

Figure 39 - Modified CG.C on Magny-Cours

We used a 6/6, 3/3/3/3 and 2/4/4/2 configurations for the twelve thread experiments and a fully subscribed twenty-four configuration for the last two experiments. The results show how far performance can drop when a bad decision regarding memory allocation is made. The original CG benchmark manages to complete in 55 seconds in the 6/6 configuration and in 49 seconds in the 3/3/3/3 configuration. However, the modified version requires 110 seconds for the best run and 131 seconds for the worst run. The worst execution is using the diagonal nodes to run the threads, whereas memory is allocated on the first node. The diagonal nodes use slower HT interconnect links and a 10% decrease in performance is achieved. The difference of performance is a drop of more than 100%.

However when the full twenty-four threads are used performance takes a larger hit. The original version requires 28.6 seconds, whereas the modified version needs 115 seconds. The modified version runs 4 times slower than the original. The cause of this slowdown is the saturation of the integrated memory controller of the first node. As we have seen earlier, the integrated memory controller is saturated when more than two threads of the CG benchmark try to concurrently access memory. Therefore, the memory controller cannot keep up with twenty-four threads requesting data at the same time. These experiments show that the memory allocation policy on a platform that has many processors is very critical and a small mistake can affect performance.

6. Results on Intel Nehalem E5504 platform

This chapter describes the experiments and analyses the results of the Intel Nehalem platform. The Intel Xeon E5504 platform consists of two Intel Xeon Gainestown processors (E5504). Each processor features 4 cores. Therefore, the total available cores of the platform are eight [35]. Each processor has 3 GBs of 800 MHz, DDR3 memory per core, thus 12 GBs per processor and 24 GBs total. Overall memory bandwidth is 19.2 GB/sec [36], since the Nehalem platform uses three DDR3 channels per processor.

Each Intel Xeon Gainestown core has two dedicated levels of cache and a shared level. Level 1 cache consists of a 32 KB 2-way associative instruction cache and a 32 KB 2-way associative data cache. Level 2 cache is a 256 KB exclusive 16-way associative cache. Level 3 cache is a 4 MB cache, shared by a core, or 4 threads. The level 3 cache is an inclusive cache, which means that the level 3 cache stores all data that levels 1 and 2 store plus cache lines that were evicted from higher cache levels. The cache lines are 64 bytes wide and are common for both L1 and L2 caches. The Xeon processor uses the level 2 cache as a victim cache. A victim cache is used to store data elements that were replaced from the level 1 cache. It lies between level 1 cache and main memory and it tries to reduce the number of conflict misses [27]. The processor chip also has an integrated dual-channel DDR3 SDRAM memory controller, which controls the main memory that is directly attached to the processor.

As stated earlier Intel processors use QPI links as a communication medium for ccNUMA implementation. The Xeon Gainestown processor has 2 QPI links, clocked at 3.2 GHz and capable of 25.6 GB/sec bandwidth per link. One QPI link connects the two processors whereas the other connects each processor with the 5520 hub interconnect. Traffic that needs to be routed to I/O goes through the hub.

A complete eight core node is shown in Figure 40**.**

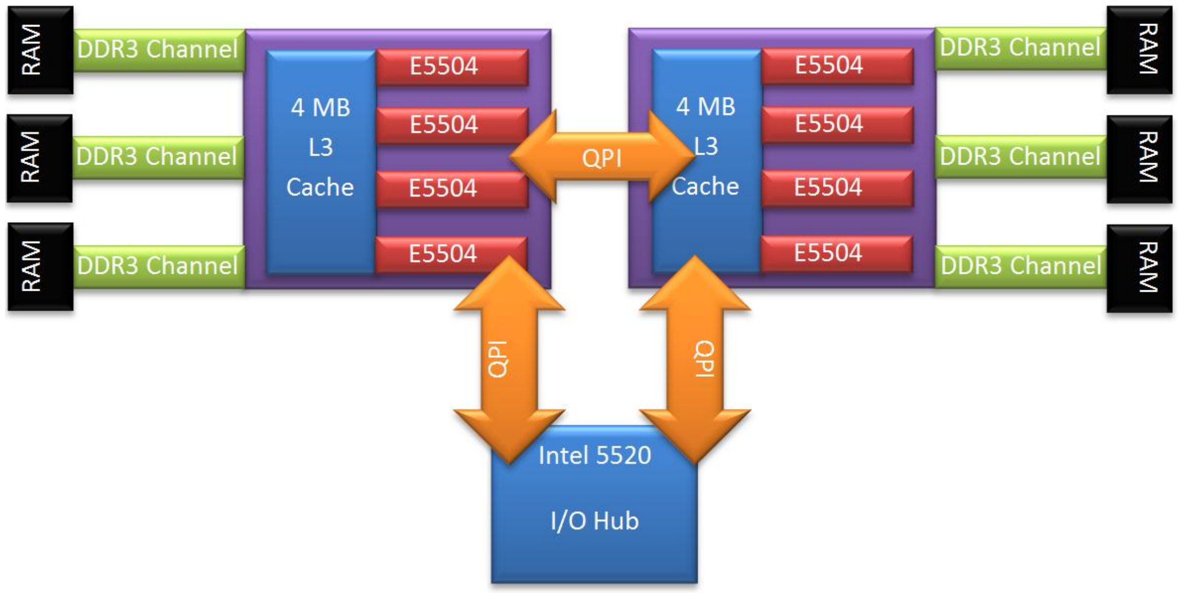

Figure 40 - Nehalem 8 core platform, from [36]

The Nehalem series of processors is based on a modular design. The components of the Nehalem design are the processor cores (which include a level 1 and level 2 cache), the integrated memory controller (IMC), the shared LLC (which varies in size) and the QPI controller. Intel can create different configurations using these components. The different configurations range from high-powered desktops to servers, and can range from four to sixteen processor cores.

As with all modern processors, the Nehalem uses branch prediction to improve performance. Nehalem processors use branch prediction in order to detect loops and save instructions in buffers. Thus, they manage to improve performance, by reducing the amount of instructions that require decoding.

Out of order execution enables the Nehalem processors to speculatively execute instructions in order to use the deep pipeline of the processor. If the processor executes the correct instructions, performance improves. If however the wrong instructions are executed, the processor needs to revert the result and try again.

Seven new instructions are included in Nehalem based processors. These instructions follow the single-instruction, multiple-data (SIMD) paradigm and are used to improve performance of multimedia and other applications.

Finally, the Nehalem architecture uses a more granular approach to power management. The new Power Control Unit (PCU) module allows each core to receive the amount of voltage that is required. Therefore, some cores may run at full voltage, whereas others can move to a sleep state in order to consume power.

6.1 Intel Xeon E5504 Micro Benchmarks

The micro benchmarks described in Chapter 3 were run on the two available Intel Xeon E5504 platforms. The results from the two platforms were almost identical. In order to present fewer graphs and to avoid duplicates the results of a single platform are presented. However, while using micro benchmarks not written in assembly language small variations of the results are expected, even when measuring the same hardware platform. The operating system plays a role as well. The Intel Xeon platform uses a vanilla Linux OS, which is not designed for long duration high performance computing applications. Therefore, simple OS functionalities (like disk bookkeeping, swap file reorganisation, signal and interrupt handling, etc.) have an impact on the results that the micro benchmarks report. As stated earlier each benchmark is executed five times and the average value is used. This approach reduces the impact of obvious outliers.

The Linux commands *taskset* and *numactl* were used in order to execute the micro benchmarks. The default memory allocation policy of the Linux OS is the first touch policy. The first touch policy allocates memory on the NUMA node that the memory request was submitted. There are cases where this allocation fails (due to insufficient free memory space). The memory subsystem tries to allocate memory on the next nodes, until it manages to allocate all the requested memory. Using the *taskset* and *numactl* commands, we set the processor affinity. Therefore, Linux has to allocate memory on the same node. The *numactl* command offers extra functionality. It allows setting the processor affinity of an application and the memory affinity.

In order to investigate the NUMA effects on performance, we need to measure and quantify two characteristics of the memory subsystem. These characteristics are memory bandwidth and memory latency. We need to understand how NUMA affects these two characteristics in respect to processor and memory locality. The first metric to be investigated is memory bandwidth. Measuring the read memory bandwidth will provide us with the first data of any performance differences of the NUMA nodes. The Intel platform has only two NUMA nodes, the least between the available platforms. Furthermore, the Intel platform uses the faster DDR3 memory controller, with three channels available for communicating with main memory. By measuring the read memory bandwidth, we exclude any differences in read/write memory speed. Also by not modifying the array, we eliminate the performance impacts of the MESIF protocol. As in the previous benchmarks, an array is allocated on a memory segment controlled by a processor and a thread that runs on the same processor tries to access all the elements. The accessed element is copied in a temporary variable. In order to avoid the elimination of the temporary variable by the compiler, the variable is used in an *if*-test after the end of the loop. The duration of the loop is recorded and it is divided by the amount of copied data. Thus, a MB/sec value is calculated. Compiler and processor optimizations will affect the runtime; however since these optimizations will take place on all processors the result will not be affected.

By following the methodologies described in the previous chapter, we executed 3.1.1 Benchmark 1 five times. Before the execution, the *MP_BLIST* environment variable was set to using the following command:

export MP_BLIST=x

x is the number of the processor that we measure. *x* is a value in the range 0 to 7, where 0 is the first processor and 7 is the last processor. The *taskset* command is also used in order to bind and set the executable to a specific processor. The syntax of the command was:

taskset –c x bench.exe

x is the number of the processor that we measure. The benchmark was executed for an array of 2,000,000 elements or 32 MBs in size. The average value was used and is presented in Figure 41.

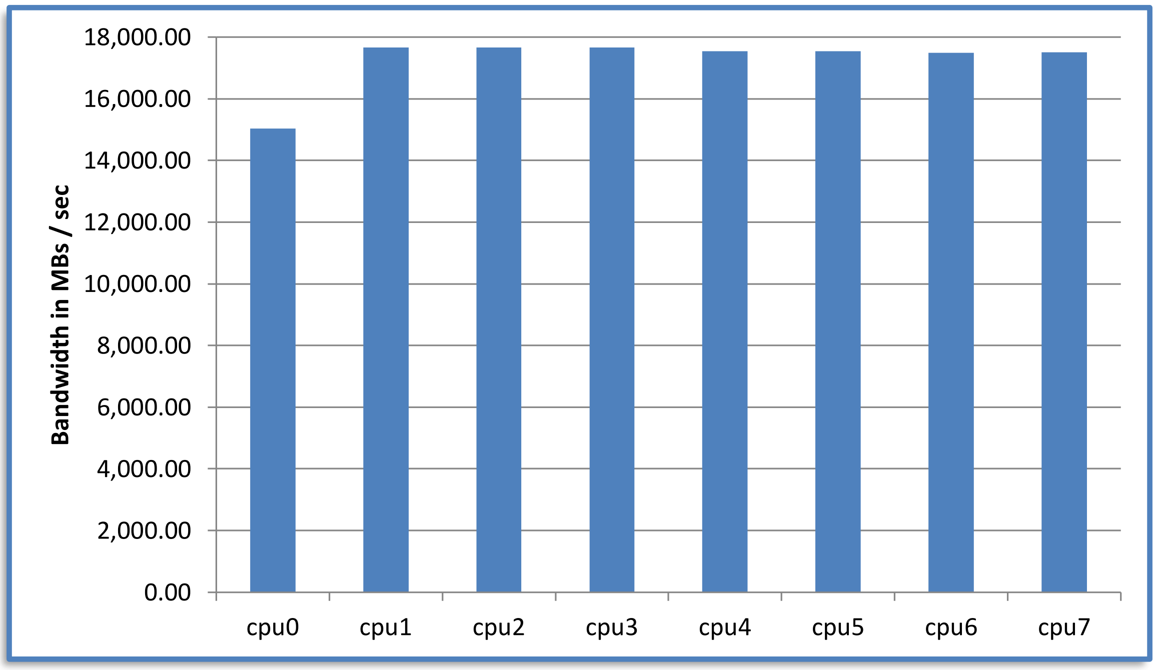

Figure 41 - Read Bandwidth Measurements on E5504 (32 MBs)

The graph shows that all processors achieve the same read memory performance, except CPU0. The implementation of the cache coherence model on this platform appears to function properly and it does not introduce any irregular bandwidth patterns. The performance of the memory subsystem is not affected by the location of the memory, when only local memory is accessed. The almost 15% reduced performance of CPU0 can be attributed to the fact that the first processor is loaded with OS functionalities.

Since the local read memory bandwidth benchmark did not return any unbalanced results, a more interesting metric that needs investigation is the read memory bandwidth when two processors are involved. This benchmark allocates memory on a single node. It then reads the allocated memory using threads that execute on other processors. Binding the threads to specific processors is achieved by using the Linux tool *taskset*. The benchmark also prints the processor affinity and the memory locality in order to verify that *taskset* has placed the thread on the proper processor. We also use the PGI *MP_BIND* and *MP_BLIST* environment variables to ensure that a thread is not migrated to another processor.

The variability of remote memory bandwidth is a characteristic of the ccNUMA memory model. An application written for a ccNUMA platform should be designed in order to take advantage of the NUMA effects. 3.1.1 Benchmark 1 will quantify the effects of accessing remote memory. It will also show the drop of performance, when accessing local and remote memory. The Nehalem platform has only two NUMA nodes; therefore, we do not expect any noticeable variances in the results. By following the methodologies described in the previous chapter, we executed the first benchmark 5 times. Before the execution, the *MP_BLIST* environment variable was set to using the following command:

export MP_BLIST=x,y

x,y are the number of the processors that we measure. *x,y* are in the range 0 to 7, where 0 is the first processor and 7 is the last processor. *x* is the processor that allocates the data array and *y* is the processor that access the array. The *taskset* command is also used in order to bind and set the executable to a specific processor set. The syntax of the command was:

taskset –c x,y bench.exe

x,y are the number of the processors that we measure. The benchmark was executed for an array of 2,000,000 elements or 32 MBs in size. The average value was used and is presented in Figure 42.

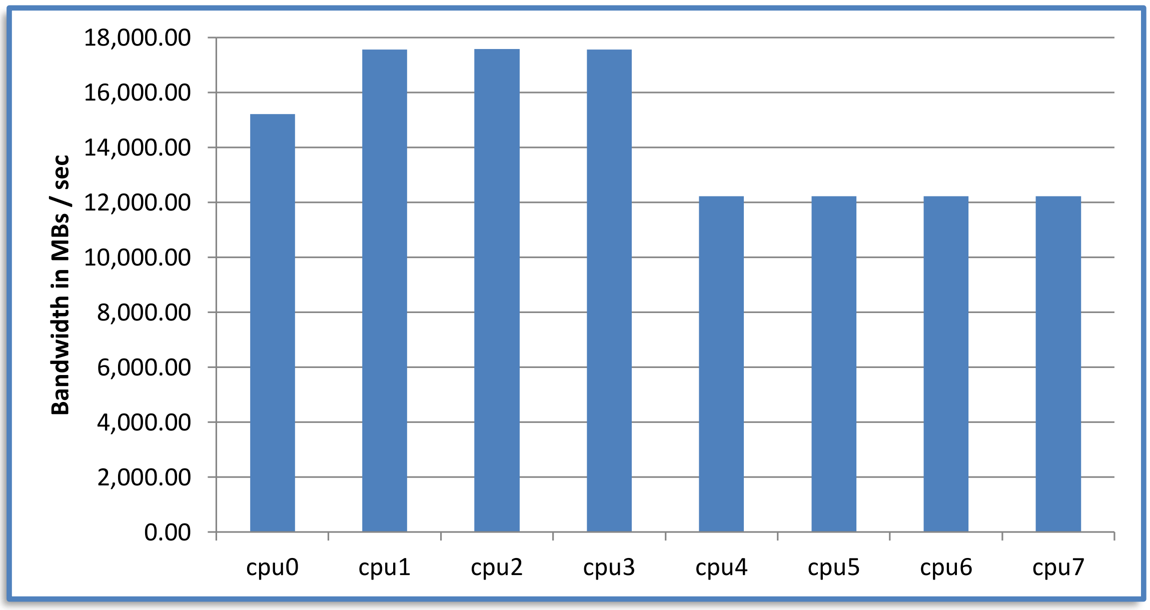

Figure 42 - Distributed Read Bandwidth on E5504 (32 MBs)

Memory is allocated on the first processor (CPU0) and is then accessed serially by threads executing on the other processors. The results show that the effective read memory bandwidth for CPU0 – CPU3 is the same as in the previous local allocation (Figure 41). Since CPU0 – CPU3 share the same chip all memory transfers take place locally with little to no extra cost. The only cost is the extra MESIF snoop messages that are sent to the remote cache controller. However, the delay caused by the probes is already factored in the previous benchmark.

On the other hand, the remote processors (CPU4 – CPU7) have to transfer memory over the QPI link. Thus, the read memory bandwidth of the remote processors is limited by the QPI interface bandwidth and is reduced by almost 30%. Since the E5504 platform only has two NUMA nodes, the distance is fixed and all remote processors show the same performance. The bandwidth of the

QPI interconnect is 12.8 GBs/sec per direction. Therefore, the values that the remote processors reach are very close to the theoretical maximum of the link. As we mentioned earlier we only measure the remote read memory bandwidth in order to gain a high overview of the NUMA system.

Both bandwidth benchmarks returned data that fit and collaborate with our theories. The local bandwidth benchmark showed that on a finely tuned ccNUMA platform, access to local memory does not suffer from the cache coherence protocol. The remote bandwidth benchmark revealed that remote memory bandwidth decreases by 30%. However the achieved throughput is very close to the unidirectional limit of the QPI interconnect. Furthermore, a decrease of remote bandwidth is expected, since data need to transfer through two memory controllers and an external point-to-point bus. Since these two metrics did not expose any strange NUMA effects, we will investigate the effects of NUMA on local and remote memory latency.

Memory latency is another factor that affects performance of a platform. Memory latency is the time that a processor needs to complete a memory request. The memory request may ask for a local or remote cache (L1, L2 or L3), local or remote main memory. The memory location controls the value of the memory latency. Memory that is located further away from the requesting processor should result in a higher latency result. Since Linux uses the first touch policy, we know that memory will be allocated on the node that the executing thread is running.

By following the methodologies described in the previous chapter, we executed the latency 3.1.3 Benchmark 3 five times. Before the execution, the *MP_BLIST* environment variable was set to using the following command:

export MP_BLIST=x

x is the number of the processor that we measure. *x* is a value in the range 0 to 7, where 0 is the first processor and 7 is the last processor. The *taskset* command is also used in order to bind and set the executable to a specific processor. The syntax of the command was:

taskset –c x bench.exe

x is the number of the processor that we measure. The benchmark was executed for memory sizes up to 64 MBs in size. The average value was used and the results are presented in Figure 43.

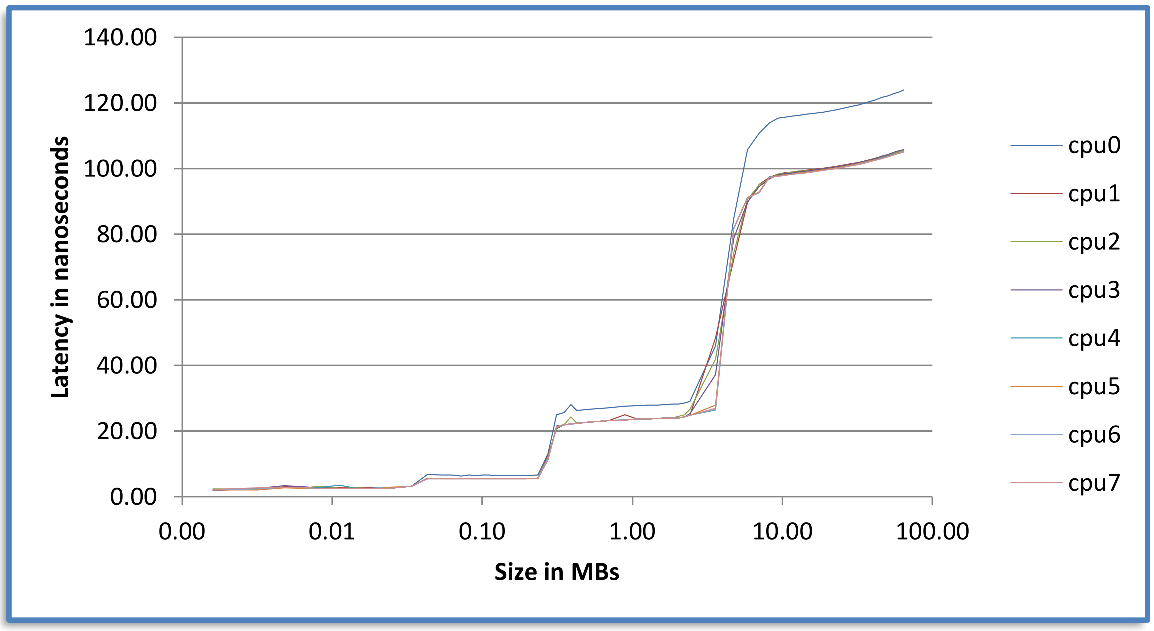

Figure 43 - Nehalem Latency Measurements

The size of the array (x-axis) is represented in logarithmic scale. All processors perform equally well, except from CPU0. As stated earlier this is expected due to OS functionalities. The graph reveals three clear steps. These steps overlap with the size of the three cache levels of the Intel Xeon platform. When the array is small enough to fit in the cache, the processor optimization routines prefetch the data and access times are low. Principally we measure the latency of level 1 cache. However as the array increases in size and becomes larger than the cache, access times become higher and latency increases. Level 1 cache accesses are measured in the range of 2-3 nanoseconds. Level 2 cache accesses are measured in the range of 4-6 nanoseconds. Level 3 cache accesses are measured in the range of 22-25 nanoseconds. The values that we measure are very close to the values that Intel has published for the Nehalem processor.

The latency micro benchmark is designed to make prefetching data very hard. The array is constructed in a pseudo-random arrangement in order to avoid prefetching. Furthermore only one element for every cache line is accessed, so even for small array almost all memory requests cause level 1 cache misses. As stated earlier, each level 1 cache miss is followed by a series of signals sent to the other processors in order to find the missing data in the cache memory of another processor. The signals follow the rules of the MESIF protocol and if the data are found, then the cache controllers initiate a direct memory copy through the QPI link. Since there are only two NUMA nodes, communication costs are reduced.

The previous results show that the Intel Xeon platform can handle and it scales very well when accessing local memory. Unfortunately, a similar single node Intel Xeon platform was not available to compare latency results without the extra latency costs of the MESIF protocol.

In order to verify if NUMA distance affects memory latency, we use the distributed latency benchmark. 3.1.4 Benchmark 4 allocates memory on one node and it tries to read all data elements from threads that execute on remote nodes. Since the Nehalem platform has only two NUMA nodes, we expect that the results will be similar to the local latency benchmark with an added latency cost for remote processors. Memory was allocated on the last processor (node1, cpu7) in order to avoid noise from the operating system. The benchmark was executed five times and it followed the methodologies described in the previous chapter. The PGI environment variable *MP_BLIST* was set to:

export MP_BLIST=x,y

x,y are the number of the processors that we measure. *x,y* are in the range 0 to 7, where 0 is the first processor and 7 is the last processor. *x* is the processor that allocates the data array and is equal to 7 and *y* is the processor that access the array. The *taskset* command is also used in order to bind and set the executable to a specific processor set. The syntax of the command was:

taskset –c x,y bench.exe

x,y are the number of the processors that we measure. The benchmark was executed for memory sizes up to 64 MBs in size. The average value was used and is presented in Figure 44.

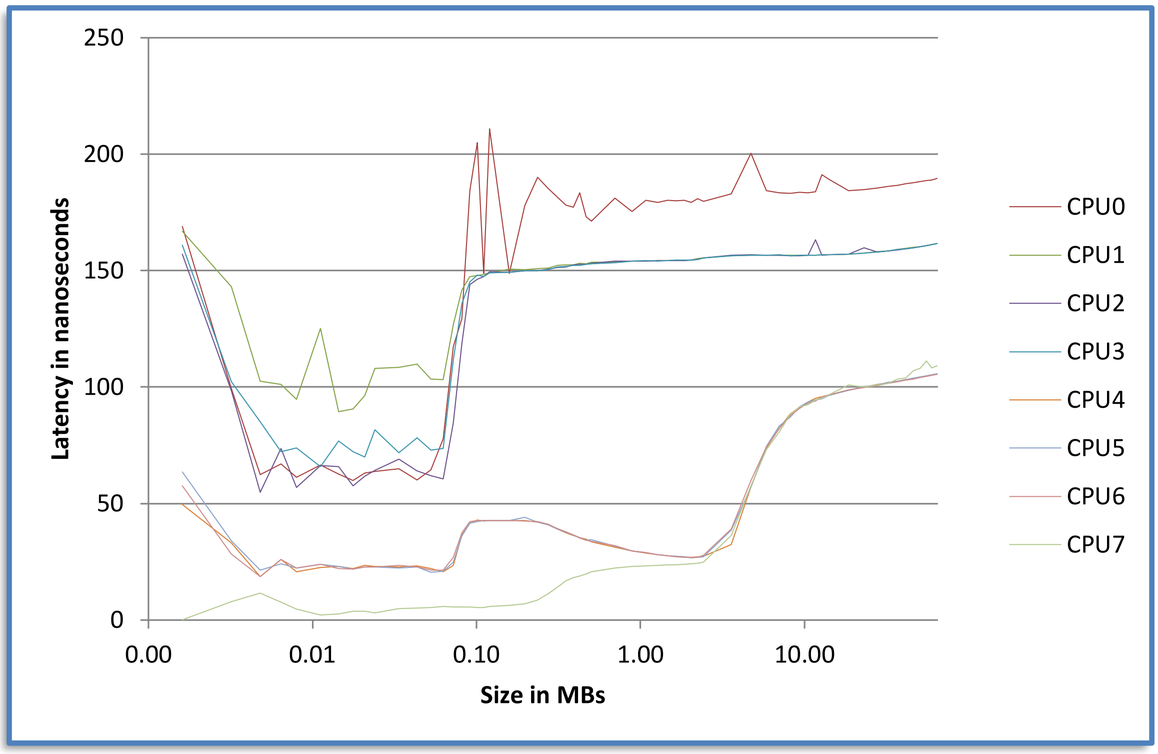

Figure 44 - Nehalem Distributed Latency Measurements

The size of the array (x-axis) is represented in logarithmic scale. The locality of processors can be seen clearly in the previous figure. The four remote processors are grouped together in the upper half of the graph. The first processor (CPU0) stands out, as the data points are not consistent. The experiment was repeated five times and the average values were used, but the erratic behaviour of CPU0 is still evident. Furthermore, latency is very high for small arrays. This can behaviour can be attributed to the fact that for very small arrays (couple of cache lines long) every wrong prediction regarding loop unrolling or prefetching data/instructions could cause a significant amount of extra time to the benchmark.

The four remote processors show an increase in latency. For arrays that are smaller than level 1 cache latency has risen from 1-3 nanoseconds to 65 nanoseconds. For arrays that are smaller than level 2 cache latency has risen from 4-6 nanoseconds to 140 nanoseconds. For larger arrays latency has risen from 22-25 nanoseconds to 150 nanoseconds. For reference, local memory latency from the previous experiment was around 105 nanoseconds. Therefore, memory latency for level 2 cache accesses for remote processors is slower than accessing local memory.

However, the increase of remote latency is not as great as we expected. Furthermore, it seems that the latency for level 1 and level 2 cache arrays is nearly as low as latency of local processors. As we have discussed earlier the Nehalem features a shared level 3 cache with a size of 4 MBs per node. This cache is inclusive, which means that it contains the data of lower levels of cache (L1 and L2) and extra data that the processors of the node think that might be used in the near future. Therefore, any MESIF snoop message that arrives on a particular node can be processed very quickly, since only one level of cache needs to be checked. If the data exist on level 3 cache they are bound to be present on levels 1 or 2. Thus, latency of remote level 1 and level 2 caches are identical and very low. The snoop messages do not actual reach the lower levels of cache; instead, they are processed at a higher level and only need to check one level of cache.

For local processors latency performance is better. For arrays that are smaller than level 1 cache latency has risen from 1-3 nanoseconds to 25 nanoseconds. For arrays that are smaller than level 2 cache latency has risen from 4-6 nanoseconds to 45 nanoseconds. For larger arrays, latency has risen from 22-25 nanoseconds to 30 nanoseconds. Local memory latency remains at a respectable 105 nanoseconds.

Even though the latency values for the local processors are very close to their local values from the previous experiment, an inconsistency shows up in the graph. It seems that latency is reaching a peak around the time the array size is larger than level 1 cache size. It then starts to decrease slowly until the time that the size of the array becomes equal to the level 3 cache size. It then increases again as the array size becomes larger than the level 3 cache size.

This behaviour is contradictory to previous experiments. However if we take a closer look at the various cache levels of the Intel Xeon the following theory explains this behaviour. As we have already seen the level 2 cache of the Xeon acts as a victim cache for level 1. Furthermore, each processor has a separate 32 KBs level 1 cache and a separate 256 KBs level 2 cache. As the array is smaller than the level 1 cache size latency is low. This is expected, as the array fits into level 1 cache and cache memory access is very fast. However because each processor has a separate level 1 cache, for every level 1 cache miss, the requesting processor must send a MESIF snoop message. The snoop message will be intercepted by the local and remote level 3 caches. However, by the time that the message arrives to the remote level 3 cache the local level 3 cache would have enough time to serve the request and also to send a cancellation MESIF message to inform the other nodes. The local level 3 cache would check its data and locate the requested cache line. It would then serve the cache line to the requesting processor and update the state on the level 3 and to the appropriate lower cache level. These operations take approximately 22 nanoseconds to complete. According to the literature [37] and to documents published by Intel these operations take approximately 13 ns. The different values can be attributed to the fact that we use a high level language (C), whereas the authors of the sources used low level assembly in order to control the state of specific cache lines. However according to the same sources the same latency should be observed for level 2 cache accesses. However, in our benchmarks the latency value for local level 2 cache access is 42 nanoseconds. Since we do not know the state of each cache line, we do not know the sequence of events that the MESIF protocol needs to take. In addition, we do not know which cache lines have to be invalidated.

As the size of the array gets even larger latency becomes to fall. If we consider that the array is constructed in a pseudo-random fashion to avoid prefetching, one can assume that the level 2 cache is not utilized fully. Therefore, as the array gets larger the requesting processor needs to invalidate lines only in level 3 cache. This behaviour will lead to lower values for latency, since fewer operations would be required in order to fetch data from CPU7.

However as the array becomes larger than level 3 cache size, the delay of retrieving data from RAM dominates and latency starts to rise again. This behaviour is not seen when remote processors request data from CPU7 because the QPI interconnect links are slower than the cache access speed.

The final metric that we quantify is contention over the QPI interface and of the integrated memory controller. Contention can be simulated by allocating all memory on the first node and accessing the same memory from all other processors. This experiment would cause a high surge of traffic on the QPI interconnect and many snoop messages on the IMC of the first node. We also varied the size of the allocated memory in order to investigate if the size of the array affects performance. The results are presented in Figure 45.

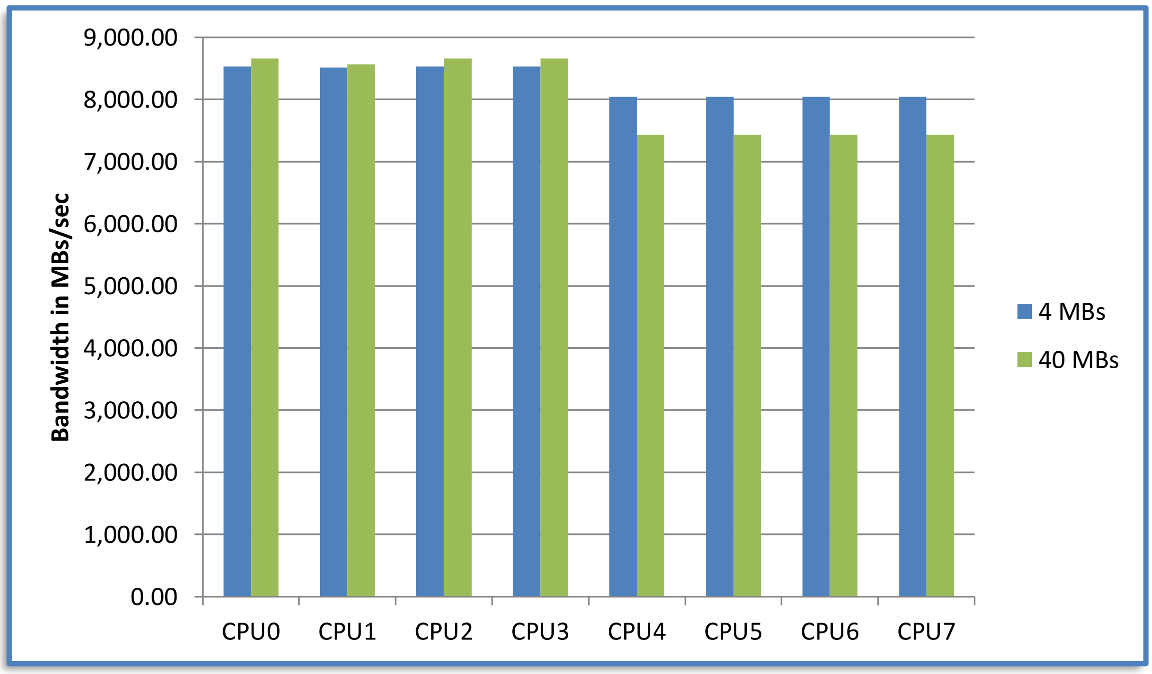

Figure 45 - Nehalem Contention Measurements

The results show that the size of memory does affect performance. However, this is the case because the first experiment used an array size that is less than the LLC. Therefore, we also measure the effects of level 3 cache on performance. It seems that if the array can fit in the LLC we reduce performance by 1% on the local nodes and improve performance by 10%. As we have discussed earlier, the Nehalem can access remote level 3 cache faster than local memory. Therefore, the results of the remote nodes are expected to be better when the array can fit in the LLC. However, the drop in performance for the local nodes is interesting. We can attribute the drop in performance in the slow update rate of the inclusive level 3 cache. As we have discussed earlier, an update to an inclusive cache is slower since a line is not evicted from a lower level cache to a higher level cache. Instead, both need to be updated. The update process happens in parallel; however, the time that the slowest update (to the LLC) takes is the one that controls the duration of the event. Therefore, when we use an array larger than the LLC, we force constant updates to all levels of cache and this can cause the drop in performance on the local node.

Performance for local nodes drops from 17 GBs/sec to 8.6 GBs/sec for local read bandwidth and from 12 GBs/sec to 7.4 GBs/sec for remote read bandwidth. We observe a large drop in performance, which is caused due to contention on the local integrated memory controller.

6.2 Intel Xeon E5504 Applications

The results of the micro benchmarks showed that the basic assumptions of this thesis are valid and that the location of memory affects performance. The results of the distributed latency micro benchmark showed that the effects of very fast cache are cancelled by accessing memory that is located in a remote chip. Even accessing memory that is located in the local chip increases the latency by a factor of 10 for cache level 1 and cache level 2 accesses. Level 3 cache latency is not affected, since level 3 cache is shared amongst processors of the same chip. The bandwidth micro benchmarks showed that the bandwidth of each processor is the same, with small variations for CPU0. However when remote memory is accessed the attainable read memory bandwidth is reduced by 30%.

Nevertheless, the micro benchmarks do not represent a typical application. Therefore, the CG and MG benchmarks were executed in order to confirm that our understanding of the Nehalem memory subsystem is sound. The *taskset* command is used to restrict execution on specific processors. The command used was:

taskset –c x-y ./cg.B.x

x,y are in the range of 0-7 and represent the available processors. The results of this experiment are presented in Figure 46 and lower values are better.

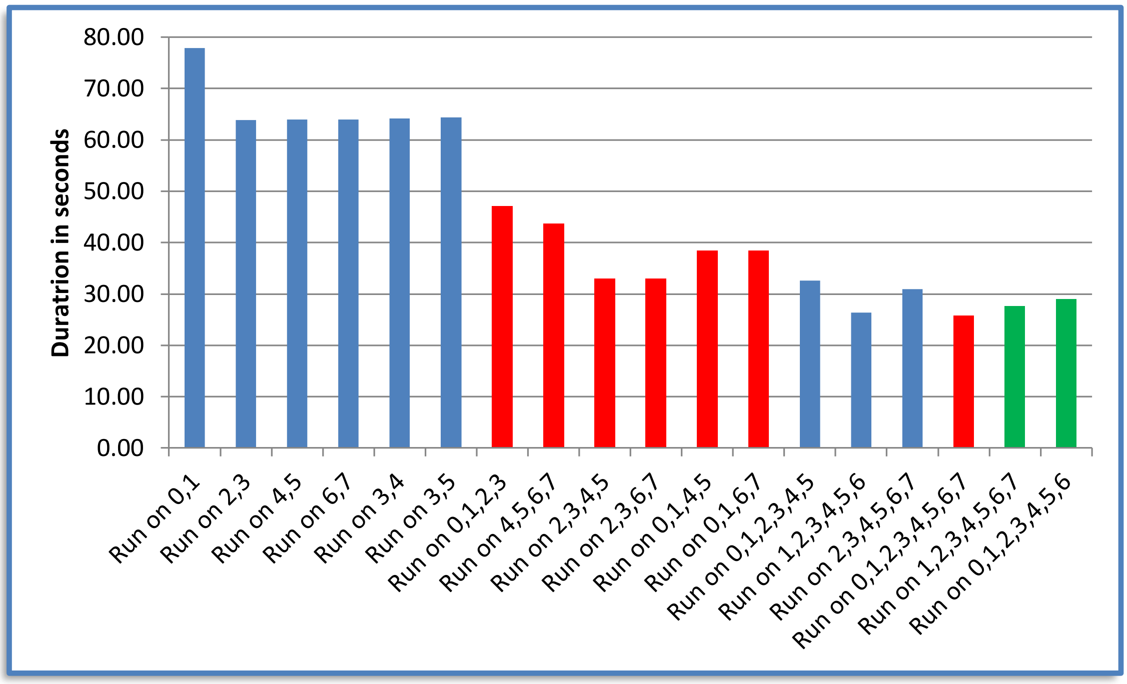

Figure 46 - CG (class B) using 2,4, 6, 7 and 8 cores on Nehalem

The results of the CG benchmark show that in order to achieve best performance, the benchmark should be executed using processors from both nodes. The CG benchmark tests communication latency. As mentioned earlier each node has three DDR3 memory channels that are used to access main memory and has very low memory latency between the nodes, thanks to the inclusive level 3 cache. The utilisation of both nodes enables the Nehalem memory subsystem to achieve the best possible memory bandwidth, as the six DDR3 channels are fully used by the two memory controllers. In addition the low latency between the nodes allows the CG benchmark to perform better.

Even when two threads are used, the performance of the benchmark is not affected by the location of the threads. The only slow result is achieved when we use the first processor. However, we believe that the OS system is responsible for the slow performance when using the first processor. As we increase the number of threads, it is clear that the best durations are achieved by dividing the threads amongst the two nodes. Since the Nehalem shows very fast remote latency values, diving the threads allows us to use the six available DDR3 memory channels. The best placement policy for four threads is a 2/2 configuration. The same is true for the six thread experiments, where the 3/3 configuration returned the best duration times.

The difference in the performance of the benchmark is a reduction of the runtime by 33%. Therefore, unlike other platforms, it seems that the Nehalem platform shows improved performance when the workload is divided between the nodes. Consequently, the performance of a latency centred application can be increased by approximately 33% with the careful selection of executing processors, when only four processors are used.

The next figures (Figure 47 and Figure 48) show the results of the CG class B and class C benchmarks when two, four, six and eight cores are used. The results show that as the number of cores increases the performance benefits decrease. It also shows that that when using very few cores the performance difference on memory and thread locality are smaller than when using cores in the medium range. Finally, the best allocation when using six cores performs as good as using eight cores on the Intel platform. However, upon inspecting the data for CG class C it seems that the duration of the benchmark when it uses two cores can be divided into two groups. The fast group has an average duration of 177 seconds. However, the slow group has an average duration of 351 seconds. This value is double than the average and is not affected by the processors being used. We did not find a cause on slower performance of some iterations of the CG class C benchmark.

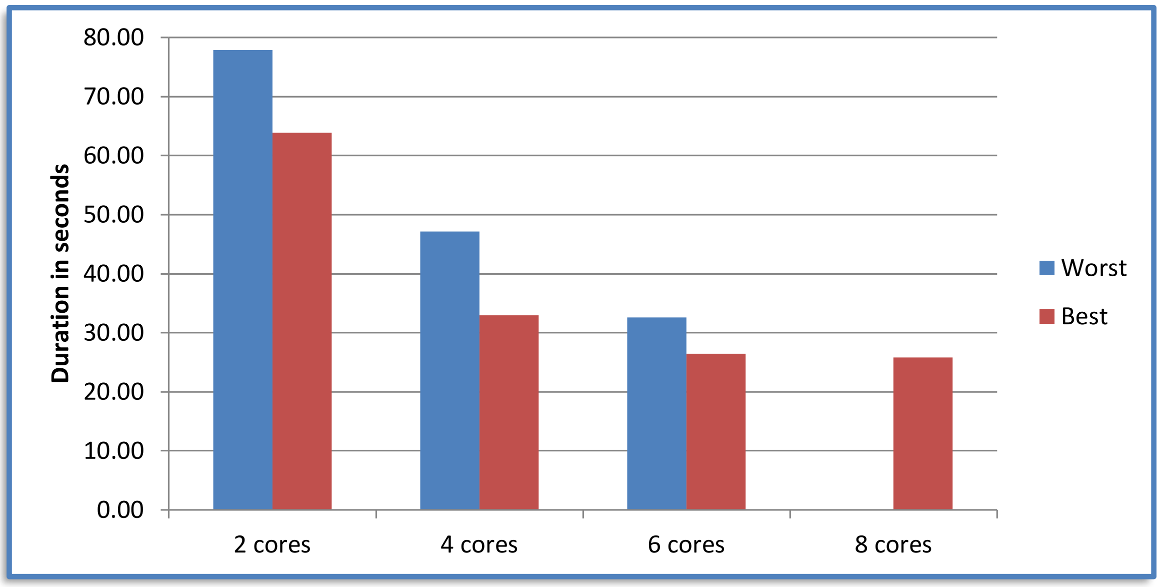

Figure 47 - CG.B comparison on Nehalem

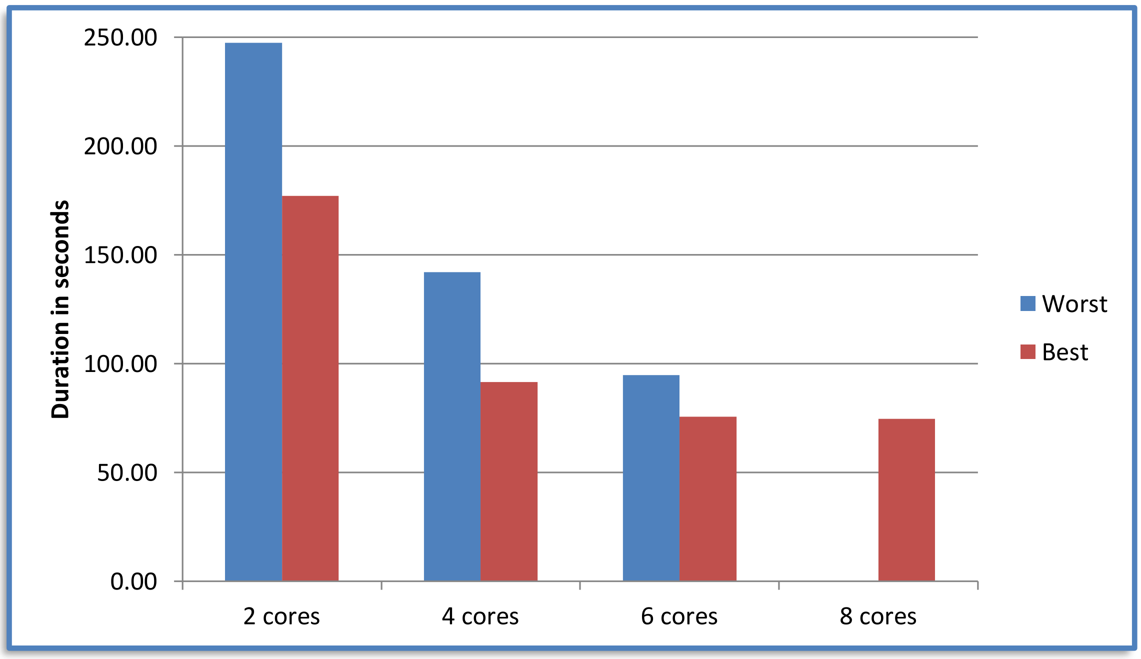

Figure 48 - CG.C comparison on Nehalem

As discussed in previous chapters the MG benchmark is memory-bandwidth intensive. We expect that the results of the MG benchmark follow the same trend as the CG benchmark. Indeed the results of the MG class B benchmark are very similar to the CG benchmark. The results are presented in Figure 49.

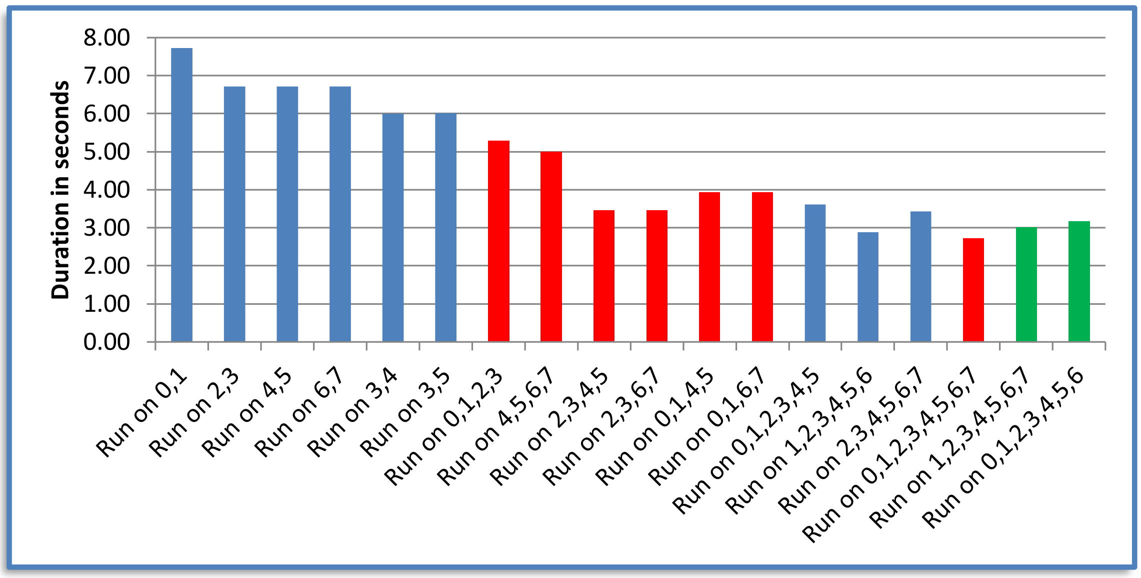

Figure 49 - MG (class B) using 2, 4, 6, 7 and 8 cores on Nehalem

The strategies that yielded better results for the CG benchmark are also applicable for the MG benchmark. However, on the MG benchmark even for the experiment with two threads is beneficial to divide the two threads on two different nodes. Since the MG benchmark stresses memory bandwidth, using two nodes enables the application to use the six DDR3 memory channels that are available. As we increase the number of threads, we observe that the benchmark performs best when we equally divide the threads amongst the NUMA nodes.

The comparison results of using two, four, six and eight threads are presented as Figure 50. However, the results of MG class C are not consistent with the previous benchmarks. Figure 51 shows the results of MG class C on two, four, six and eight cores run on the Nehalem platform.

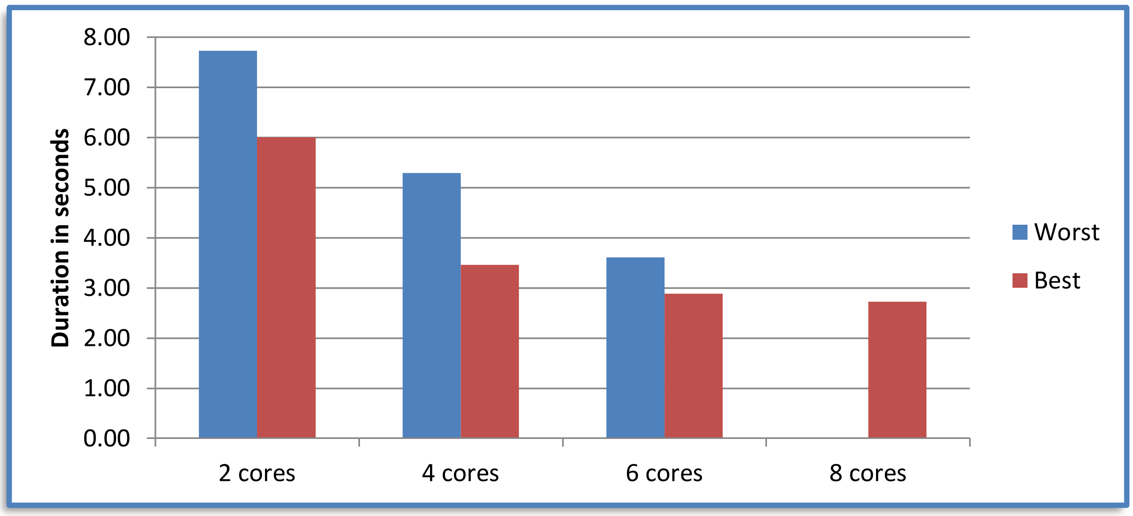

Figure 50 - MG.B comparison on Nehalem

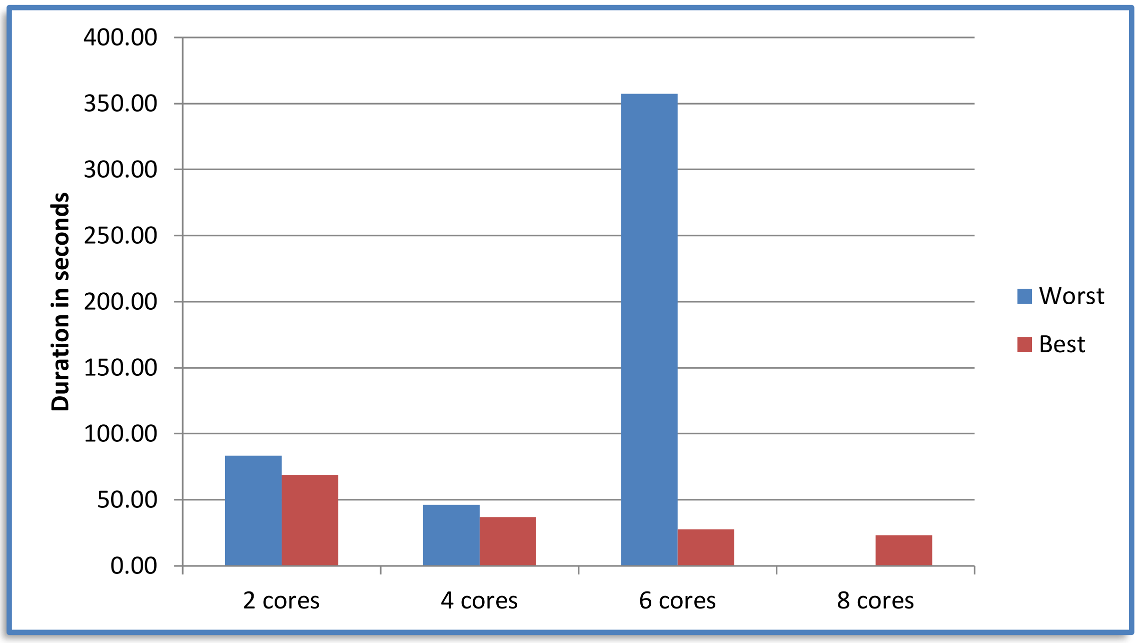

Figure 51 - MG.C comparison on Nehalem

The runtime of the MG.C benchmark for two, four and eight cores is following a normal trend. However when six cores are used the runtime of the benchmark increases by 1,200% for the worst-case scenario. The best-case scenario runs for 27 seconds and cores 1 to 6 are used (three from one node and three from another). The worst-case scenarios run for 350 and 357 seconds and take place when four cores are used from one node and two cores from the other (a 4/2 configuration).

According to P. S. McCormick et al. [38], CPU0 and CPU7 exhibit lower performance on the Nehalem platform. The reason behind the reduced performance is that CPU7 is used to map timer interrupts. According to [38] "The kernel in the Intel system is not tickless and therefore there are numerous timer interrupts during the course of benchmark execution". Furthermore, CPU0 is tasked with handling network interrupts. Whenever a network message is received the hardware network component sends an interrupt to CPU0. That processor needs to put on hold any activity that it is doing in order to handle the network interrupt. They also believe that the lower performance of the first processor is due to operating system activities.

These irregular results for both MG.C and CG.C prompted us to check the data more thoroughly. The abnormal results could be replicated as long as four processors where used on one node and two on the other.

The NUMA API for Linux provides a tool that shows the amount of memory in pages that has been allocated locally and remotely. It also displays the number of allocations that where intended for a particular node and succeeded there as numa_hit. numa_miss records how many allocations were allocated on a different node than the intended one. numa_foreign records the number of allocation that were intended for another node, but ended up on the current node. interleave_hit is the number of interleave policy allocations that were intended for a specific node and succeeded there. local_node is incremented when a process running on the node allocated memory on the same node and other_node is incremented when a process running on another node allocated memory on that node. The tool is called *numastat*. The data of *numastat* that we collected before and after each experiment, are presented in Table 6 and Table 7 with the average values for the fast and slow runs:

	Slow run			Fast run		
	node0	node1	**SUM**	node0	node1	**SUM**
numa_hit	323,368	97,379	**420,747**	209,189	119,175	**328,364**
numa_miss	0	0	**0**	0	0	**0**
numa_foreign	0	0	**0**	0	0	**0**
interleave_hit	46	50	**96**	31	25	**56**
local_node	323,316	97,281	**420,596**	209,134	119,153	**328,287**
remote_node	53	98	**151**	55	22	**77**

Table 6 - CG.C numastat average output

	Slow run			Fast run		
	node0	node1	**SUM**	node0	node1	**SUM**
numa_hit	610,747	456,363	**1,067,110**	413,898	476,564	**890,462**
numa_miss	0	0	**0**	0	0	**0**
numa_foreign	0	0	**0**	0	0	**0**
interleave_hit	47	54	**102**	0	0	**0**
local_node	610,699	456,265	**1,066,964**	413,896	476,557	**890,453**
remote_node	48	97	**145**	2	7	**9**

Table 7 - MG.C numastat average output

It seems that the slow runs allocate more NUMA memory than the fast runs. CG.C allocates almost 30% more NUMA memory, whereas MG.C allocates 20% more. Furthermore, the slow runs allocate all the extra NUMA memory on node0. Both benchmarks allocate 50% more NUMA memory on node0 than the fast benchmarks.

These values could explain the increase of the duration of CG.C that doubles in the slow runs. However, MG.C runs 13 times slower than in the fast run.

However Z. Majo and T. R. Gross [39] explain in their paper how the Nehalem memory subsystem works. Apart from the components already mentioned, the Nehalem subsystem is equipped with two Global Queue(s) (GQ). The purpose of the GQ is to arbitrate all memory requests that try to access local or remote caches and local or remote RAM, thus it helps to implement the cache coherence protocol of the Nehalem platform. The GQ has different ports to handle all possible memory access requests.

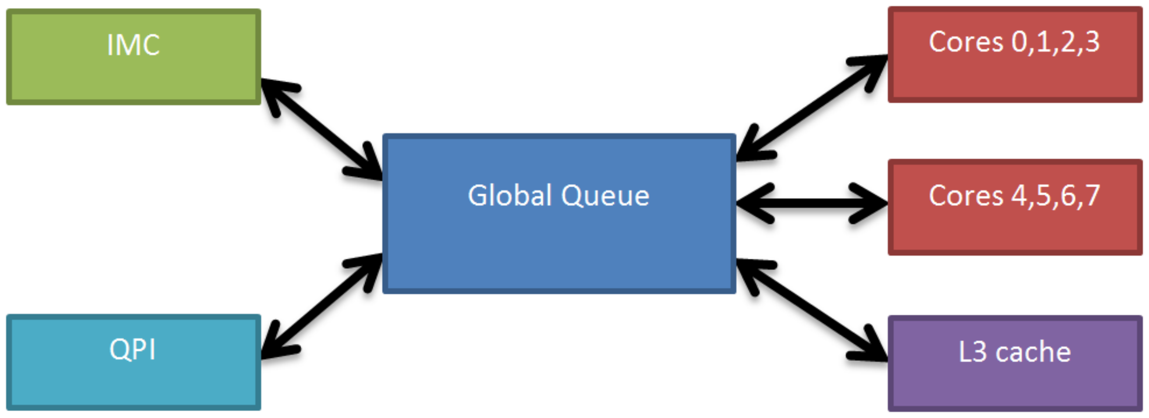

Figure 52 - Global Queue, from [39]

Therefore, the fairness of the GQ algorithm is very important, since it can cause different parts of the memory subsystem to stall waiting for their turn in accessing a memory component. According to their findings when the GQ is fully occupied, its sharing algorithm favours execution of remote processes. It is therefore unfair towards local cores. In their conclusion, they note that when many or when all the cores are active, data locality is not always the solution, as “the bandwidth limits of the memory controllers and the fairness of the arbitration between local and remote accesses are also important.”

Furthermore, Jin et al. [40] in their paper, which investigated resource contention in multicore architectures, discovered that the Nehalem platform suffers from extreme contention of the LLC and the on-chip memory controller when running the MG benchmark. However, they do not reach any conclusions on the root cause of the contention.

As we have previously seen, *numastat* does not report statistics per process. Instead, statistics of the whole system are returned. Therefore, we used the *numa_maps* file that holds data per process in order to investigate the performance issue of the MG benchmark. Using the perl script created by J. Cole [49] we analysed the results, which are presented in Table 8.

	Slow run			Fast run		
	node0	node1	**SUM**	node0	node1	**SUM**
Pages (Size in GBs)	580,480 (2.21)	292,447 (1.12)	**872,927 (3.33)**	438,865 (1.67)	434,062 (1.67)	**872,927 (3.33)**
Active			**7 (0.00)**			**872,688 (3.33)**
Anon			**872,723 (3.33)**			**872,723 (3.33)**
dirty			**872,723 (3.33)**			**872,723 (3.33)**
MAPMAX			**192 (0.00)**			**192 (0.00)**
Mapped			**206**			**206 (0.00)**

Table 8 - numa_maps results on Nehalem

According to the *numa_maps* file, both benchmarks allocate the same amount of memory. This correlates with our understanding of how the *taskset* command operates on applications. Furthermore, the amount of pages allocated on each node agrees with the *taskset* options that we used. The slow runs used the option "-c 0,1,2,3,4,5" which binds more threads on the first node (node0), whereas the fast runs used the option "-c 1,2,3,4,5,6" which binds threads evenly across the nodes. We observe that the allocation of pages agrees with the binding of the threads.

However, there is a difference in the value of active pages. According to the Linux memory documentation, the Virtual Memory subsytem (VM) uses the following classification system to describe the status of a memory page:

- Free. This indicates that a page is free and available for allocation.
- Active. This indicates that a page is actively in use by either the kernel or a userspace process.
- Inactive Dirty. This page is no longer referenced by the process that allocated it. It is a candidate for removal.

The *anon* value is the amount of memory that the MG benchmark allocates when it solves a class C problem and is allocated as the heap. The difference between the *anon* value and the pages value is the amount of memory that shared libraries are using. It seems that a very small percentage of the allocated pages are active when performance is low. On the other hand, when performance is normal, almost all the allocated pages are in the active list. We think that the cause of the low performance is that the static memory allocation of the MG class C benchmark is greater than 2 GBs. Linux divides memory in a low and a high area. The high memory area is only addressable via virtual addresses, even for a 64-bit system. When we allocate memory in 3/3 configuration the maximum memory, per node, lies below the 2 GB limit. However, when we use a 4/2 or 2/4 configuration, one node has to allocate more than to 2 GBs of memory.

In order to validate our theory, we modified the Makefile of the NPB benchmark suite and added the option "-mcmodel=medium". This option allows the generated application to allocate and access memory segments larger than 2 GBs of memory. This option is missing from the official Makefile templates that the NPB team distributes. We then managed to compile the MG class C using the PGI compiler. When we executed the application, runtimes returned to normal values. The 3/3 configuration was still faster, however the 4/2 configurations managed to complete the benchmark in 29.2 seconds compared to 23.6 seconds of the 3/3 setup.

Chapter 7

7. Conclusion

This chapter presents our conclusions for every platform and try to compare their performance. Finally, it will present some suggestions for future work.

Our main aim was to measure and quantify how the ccNUMA architecture affects performance. We used several micro benchmarks to measure the low-level performance characteristics of each platform. Furthermore, we used two kernel benchmarks from the NPB suite to measure and quantify performance on a platform level.

The following factors affect memory performance, and in association the performance of a platform:

- The state of the cache line (as dictated by the five states of the cache-coherence protocol)
- The action of the processor that requests the memory (read/write)
- The location of the requested memory (local or remote L1, L2, L3, main memory)
- The destination of the requested memory (local L1, L2, L3, main memory)

Our micro benchmarks measured and quantified the last three factors. However, only micro benchmarks written in assembly language could define the state of individual cache lines. Therefore, our results include some uncertainty that shows up as small variances of our results compared to the literature.

We did not investigate the importance of shared libraries on performance. The design of our benchmarks did not allow us to measure any performance effects due to shared or static linking. Applications compiled for a Linux environment can use a shared or static link with libraries. A shared link adds a small header to the application. The operating system loader (ld.so for Linux operating systems) tries to find all the linked libraries before the application executes. If a required library is not found, execution is aborted. When all libraries are found they are loaded before the application, and the memory offset of the application is modified in order to find external references. Shared linking allows the system operator to update common libraries without re-compiling any applications. A static link adds all the compiled code of the library to the application. Therefore, the size of the application increases as we link against more libraries. Furthermore, if we require to use an updated version of a library we need to re-compile the application. On the other hand, we do not need to distribute several libraries alongside our application. However, on a ccNUMA platform, loading a shared library only on one node can cause performance issues, since the shared library is loaded only on one NUMA node. Kay et al. [41], [42] have showed that a call to a shared library takes longer to complete as the NUMA distance increases.

However, we believe that design decisions regarding thread and memory allocations affect performance more gravely than link preferences. On the other hand, according to Mytkowicz et al. [43] the order of linking or the size of the environment variables on the shell that executes the compilation process can cause differences in performance up to 10%. Therefore, a simple performance improvement is to rearrange the order of linked libraries and measure the performance of the application. Experiments run on the upgraded file system platform for the HLC by Thuressona et al. [44] revealed that the use of shared or static linking has a minimal impact on performance.

During our investigation, we executed and quantified numerous experiments. We used our micro benchmarks and the industry standard NPB benchmark suite.

7.1 X4600

The X4600 is the oldest amongst the platforms that we tested. Our experiments showed that the X4600 exhibits very irregular performance patterns. The inner nodes are faster than the outer nodes. We have tried to offer some explanations regarding this phenomenon. Unfortunately, other researchers used the X4600 platforms and we could not gain super user access. Furthermore, we could not patch the Linux kernel in order to install support for hardware profiling. Therefore, we cannot offer quantitative data that support our explanations.

Apart from the irregular performance, which manifests as higher bandwidth and lower latency for the inner nodes compared to the outer nodes, the X4600 shows significant NUMA effects. The reason behind the significant NUMA effects is that the X4600 has eight NUMA nodes. Access to remote memory becomes significantly slower as NUMA distance increases, and the X4600 needs a maximum of three hops in order to access all available memory. Likewise, the HT interconnect links use a slower version of the HyperTransport protocol. The X4600 platform uses AMD Opteron processors that are not fitted with a level 3 cache memory. The lack of a level 3 cache memory increases memory latency, due to the fact that more snoop messages are required in order to implement the cache-coherence protocol. As an example, both the Magny-Cours and Nehalem platforms use the LLC as a cache line directory to speed up performance [33], [45].

On top of the lack of a level 3 cache, the X4600 uses the slower DDR2 memory unlike the other platforms that use the faster DDR3. Furthermore, the X4600 platform has only one memory channel, unlike the Magny-Cours that has two channels or the Nehalem that has three channels.

For the reasons mentioned above, if an application does not allocate memory properly the X4600 is less forgiving than the other platforms. The X4600 exhibits the widest spreads in performance. Since the causes of performance drop are understood, the solution is evident. Based on our experiments the best allocation policy is locally. However, we did not fully understand the results of the contention benchmark. The first processors (CPU0 and CPU1) can access memory faster than others can. We attribute the increase in bandwidth to the fact that node0 holds all kernel and shared libraries. Therefore, the local processors have an advantage when performing system calls. However, this does not explain the performance boost. Unfortunately, we could not login to the nodes with a interactive shell in order to collect further data. Moreover, we could not login as the superuser and install a kernel with performance counter support. In all cases, local allocation performed better than other policies. In some experiments, the duration of the NPB benchmarks were reduced by 35%-40%.

For example, our modified version of the CG benchmark showed a seven-fold reduction in performance. The actual results were 50 seconds for the original version and 387 for the modified version.

Since the best allocation policy is local allocation, the availability of eight NUMA nodes complicates the proper allocation of memory. The other platforms have two or four NUMA nodes. Therefore, experiments can be executed in order to find the best allocation policy for specific applications. However, the presence of eight NUMA nodes makes the allocation process more prone to errors. Each node has only 4 GBs of memory, so when a NUMA node has no free memory, the memory allocator moves to the next NUMA node until it can allocate all requested memory. This behaviour causes performance drops even when we explicitly try to allocate memory on a specific NUMA node. A solution is to use the *numactl* tool with the *membind* option. n*umactl* will fail if not enough memory is free on the requesting node. Therefore, using *numactl* we can enforce memory allocation and be informed if the allocation fails.

Once the disadvantages of the X4600 platform were identified, we were able to use them to improve performance. We discovered that the default allocation of the Linux OS is not always the best. It seems that the Linux kernel installed on our X4600 platforms did not properly set the processor affinity on the benchmarks. Thus, different duration times were recorded for every experiment. When we used the *taskset* tool to set the affinity manually, we observed that the durations of the experiments started to converge.

7.2 Magny-Cours

The Magny-Cours platform is the latest, as in 2011, multi socket/multicore AMD offering. The Mangy-Cours platform is used by the latest Cray powered HPC systems. Our experiments showed that the Magny-Cours platform does not show any uneven performance issues. The performance was stable across the NUMA nodes, showing minimal NUMA effects. As with all ccNUMA systems, accessing remote memory comes with a cost. However, the design decisions of the AMD engineers and the software design of the CLE OS of the HECToR system manage to hide the performance effects.

The CLE OS is a specially purposed built Linux OS version. It is a stripped down version of a normal Linux OS. Any module than can reduce performance or cause performance variances has been removed or tuned. Cray maintains the CLE OS and the performance stability of HECToR is based upon CLE.

The AMD Magny-Cours platform is fitted with three levels of cache. The LLC is a 6 MBs cache that is shared between the six processors of each die. However, only 5 MBs are used as cache and 1 MB is used as storage for a cache directory (probe filter). The probe filter, or HT Assist, is used to speed up the processing of cache-coherence snoop messages. According to Conway et al. [33] the HT Assist feature can increase bandwidth by 350% and reduce latency to 70% on a twenty-four processor configuration. Furthermore, the Magny-Cours uses two memory channels of fast DDR3 compared to a single channel of DDR2 for the X4600. However, the Nehalem is using three DDR3 memory channels to communicate with the integrated memory controller.

We have measured local and remote bandwidth using the micro benchmarks and the results show that the Magny-Cours can handle local and remote bandwidth without incurring huge NUMA costs. Remote bandwidth is reduced, but the reduction is constant across all the nodes. Furthermore, the local and remote latency benchmarks showed that local latency is extremely fast, faster than the Nehalem for level 1 and level 2 caches. Remote latency is higher than local latency; however, processors on the same NUMA node use the shared level 3 cache in order to minimise the NUMA effects. Processors located on remote NUMA nodes feel the full cost of transferring memory using an HT link. Unfortunately, we were not able to reboot a node, in order to disable the HT Assist feature and measure the results. The contention benchmark revealed that when all HT links are used and all processors rush to access the same memory location, remote bandwidth is reduced by 88%. This effect of placing memory on the wrong node can have serious consequences on performance.

The experimental data showed that processor binding could have a large impact on performance. Performance of the NPB benchmarks varied. Two factors were identified that caused this variance:

- Locality of memory
- Benchmark type (bandwidth or latency controlled)

For bandwidth intensive applications (like the MG benchmark), we should spread the executing threads amongst the nodes. Using more NUMA nodes seems counterintuitive. However, using more NUMA nodes we increase the number of memory controllers that can access main memory. Each memory controller has two extra DDR3 channels to offer. Thus, the cost of the extra latency that we incur is fully compensated by the increase in raw bandwidth. This finding stands against common logic, that dictates that memory should be located as close as possible. However, it has been collaborated with experiments from two to eighteen threads. Therefore, under-subscription favours bandwidth intensive applications on a Magny-Cours platform.

For latency intensive applications (like the CG benchmark), we must place the executing threads as close as possible. Latency increases as we use more NUMA nodes, because memory accesses need to travel through an HT link. Therefore, for latency controlled applications, we should practise the standard ccNUMA training of fully subscribing nodes.

An application can change from being bandwidth to latency intensive during a single run. Therefore, it is very hard to properly select an allocation policy for fast changing application. If on the other hand, we can find out the type of the application, through experimenting or through source code inspection, then the proper allocation policy can improve performance by up to 28%.

The Magny-Cours platform does not kill performance if the wrong allocation policy is used. However, performance drops noticeably when we do not observe the characteristics of the platform. For example, our modified version of the CG benchmark showed a four-fold reduction in performance. The actual results were 28 seconds for the original version and 115 for the modified version. This result shows that the extreme case were we allocate all memory on the first node decreases performance. However, the performance drop is not as extreme as on the X4600, which decreases performance by seven times.

7.3 Nehalem

The Nehalem E5500 series is not the latest processor that Intel has produced. The newer, faster and benefiting from more threads (due to HyperThreading and more cores per processor) E7000 series processor is already available on the market. However, the E5500 is still being used in HPC platforms built by SGI. Our experiments showed that the Nehalem platform does not show any uneven performance issues. The advantage of the Nehalem platform, compared to the other platforms, is the use of fewer cores and fewer NUMA nodes. Intel does not produce any platforms that use more than two sockets. The use of only two nodes reduces the effects of NUMA.

We tested the Nehalem platform whilst it was running a vanilla Linux operating system. It would be desirable to use an operating system designed from the ground up for HPC applications. However, it was not possible to modify the two Nehalem platforms that we used.

The Intel Nehalem platform is fitter with three levels of cache. The LLC is a 4 MBs cache that is shared between the four cores of each processor. The LLC is an inclusive cache, which contains the data of all lower levels of cache of the four processors in addition to extra data that are stored in the LLC. The inclusive design of the LLC allows very fast reaction times when a snoop message arrives. Since the LLC is inclusive, the memory controller is not required to search each lower level cache in order to find if a cache line contains the requested data. An inclusive cache is slower than a victim cache, since every update of a lower level cache needs to be propagated to the LLC. However, in the Nehalem these operations happen in parallel. The Nehalem uses three DDR3 memory channels to communicate with memory. It has the highest raw memory bandwidth amongst the tested platforms.

The results from the local and remote bandwidth experiments using the micro benchmarks revealed that even for a small platform NUMA effects are visible. Local read bandwidth was measured at 17 GBs/sec, whereas remote read bandwidth was 12 Gbs/sec. This reduction is attributed to the QPI interconnect rather than the cache coherence protocol. The measured remote bandwidth is very close to the limit of the QPI interconnect, proving that the Nehalem platform can make good use of the extra bandwidth. Furthermore, the remote reduction in read memory bandwidth is evenly distributed. However, the fact that the Nehalem has only two NUMA nodes, is helping the platform to better cope with remote memory accesses.

The local and remote latency benchmarks showed that memory latency in the Nehalem platform is very low. Local latency is comparable with the AMD Magny-Cours. Memory latency to the LLC is a bit higher, however the design of the two last levels of cache are not the same. Remote latency gets a boost with the use of the inclusive LLC. We observed a large increase in memory latency on the other platforms. The extreme case is the X4600, where access of remote caches is as expensive as accessing remote main memory. The Magny-Cours performed better, however accessing remote caches incurred a considerable cost. The Nehalem was able to access remote caches almost as fast as accessing the local LLC. This proves that the QPI interconnect and the memory controller of the Nehalem can handle remote memory access.

Since the platform used only two NUMA nodes, we were not able to run many experiments and scenarios. All our experiments demonstrated that the best placement policy of threads is on different NUMA nodes. Since the Nehalem exhibits very low memory latency and can also provide very high memory bandwidth using the six available DDR3 memory channels, consequently splitting the threads on two nodes improves performance. This finding seems to contradict the view that on a ccNUMA platform memory locality improves performance. It has been shown that performance increases as we increase the distance of threads and memory. However, performance increases in platforms like the Nehalem and Magny-Cours. In the X4600, increased distance means reduced performance. Therefore, the details of each platform must be studied before applying a processor or memory affinity policy.

It seems that we have stumbled upon a bug in the implementation of the medium memory model by the GNU FORTRAN compiler. We were not able to compile the MG class C benchmark using the PGI compiler and the Makefile provided by the NPB team. We used the GNU FORTRAN compiler, which was able to compile the benchmark. However, when we executed the benchmark we discovered that the performance was very slow when we used a 4/2 configuration. We concluded that the root cause of the slow performance is medium model memory implementation of the GNU FORTRAN compiler. It seems that a bug in the implementation causes severe page swapping and increase the duration by seven times. The bug is triggered when a static memory allocation greater than 2 GBs is performed on one node. We modified the Makefile of the PGI compiler, in order to enable the compilation of an application that can allocate and access static allocations greater than 2 GBs. When we executed the PGI compiled MG class C benchmark the duration was significantly decreased.

7.4 Further Research

Unfortunately, we were not able to find out the cause of the different performance behaviour of the inner and outer nodes of the X4600 platform. We were unable to login as the superuser in order to collect BIOS and ACPI data, which we think that could have helped the investigation. Furthermore, we were unable to patch the kernel of the system in order to install hardware performance counters.

We measured and quantified read memory bandwidth. Due to time limitations, we did not measure write memory bandwidth. Since most applications read and write during the duration of their execution, write memory bandwidth could provide further NUMA effects that we did not consider.

Finally, we only considered two NPB benchmarks. A further study could include other NPB benchmarks or real scientific applications. Analysing performance behaviours of a real scientific application can be challenging. However, using the findings of this thesis, one can postulate on the performance differences of a real scientific application due to NUMA effects.

References

1. Moore's law - http://en.wikipedia.org/wiki/Moore%27s_law (accessed on 29/05/2011)
2. Scientific Programming – OpenMP - Volume 11 Issue 2, April 2003 - http://portal.acm.org/citation.cfm?id=1240072&picked=prox (accessed on 09/07/2011)
3. Symmetric multiprocessing (SMP) - http://en.wikipedia.org/wiki/Symmetric_multiprocessing - (accessed on 10/07/2011)
4. W. Stallings – "Computer Organization and Architecture: Designing for Performance" – ISBN: 978-0130812940
5. Asymmetric multiprocessing (AMP) - http://en.wikipedia.org/wiki/Asymmetric_multiprocessing - (accessed on 10/07/2011)
6. Asymmetrical Multiprocessor (AMP) Software Systems - http://labs.verisigninc.com/projects/amp-software.html - (accessed on 10/07/2011)
7. ASMP vs SMP - http://ohlandl.ipv7.net/CPU/ASMP_SMP.html - (accessed on 10/07/2011)
8. Nihar R. Mahapatra, Balakrishna Venkatrao – "The Processor-Memory bottleneck: Problems and Solutions" - http://epic.hpi.uni-potsdam.de/pub/Home/TrendsAndConceptsII2010/HW_Trends_The_Processor-Memory_bottleneck___Problems_and_Solutions.pdf - (accessed on 13/07/2011)
9. NUMA FAQ - http://lse.sourceforge.net/numa/faq/ - (accessed on 10/07/2011)
10. Non-Uniform Memory Access (NUMA) - http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access - (accessed on 10/07/2011)
11. Scheduling (computing) - http://en.wikipedia.org/wiki/Scheduling_(computing) - (accessed on 13/07/2011)
12. Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl - "Affinity-On-Next-Touch: An Extension to the Linux Kernel for NUMA Architectures" - ISBN:3-642-14389-X 978-3-642-14389-2 – 2009
13. OpenMP - http://en.wikipedia.org/wiki/OpenMP - (accessed on 13/07/2011)
14. MOESI protocol - http://en.wikipedia.org/wiki/MOESI_protocol - (accessed on 13/07/2011)
15. Understanding the detailed Architecture of AMD's 64 bit Core - http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html#3.18 - (accessed on 13/07/2011)
16. AMD64 Architecture Programmer's Manual Volume 2: System Programming - Publication No. 24593 – Revision 3.17 – Date June 2010
17. MESIF protocol - http://en.wikipedia.org/wiki/MESIF_protocol - (accessed on 13/07/2011)
18. Intel Xeon Processor 7500 Series Datasheet, Volume 2 – Order Number: 323341-001 - March 2010 - http://www.intel.com/Assets/PDF/datasheet/323341.pdf - (accessed on 13/07/2011)
19. An Introduction to the Intel QuickPath Interconnect - January 2009 – Document Number: 320412-001US - http://www.intel.com/technology/quickpath/introduction.pdf - (accessed on 13/07/2011)
20. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga - "THE NAS PARALLEL BENCHMARKS", http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf - (accessed on 26/8/2011)
21. H. Jin, M. Frumkin and J. Yan - "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance", http://www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf - (accessed on 26/8/2011)

22. Who Killed the Virtual Case File? - http://spectrum.ieee.org/computing/software/who-killed-the-virtual-case-file (accessed on 06/07/2011)
23. Study: 68 percent of IT projects fail - http://www.zdnet.com/blog/projectfailures/study-68-percent-of-it-projects-fail/1175?tag=col1;post-7627 (accessed on 06/07/2011)
24. Software Project Failure Costs Billions. Better Estimation & Planning Can Help - http://www.galorath.com/wp/software-project-failure-costs-billions-better-estimation-planning-can-help.php (accessed on 06/07/2011)
25. dlopen - http://en.wikipedia.org/wiki/Dynamic_loading (accessed on 29/05/2011)
26. Sun Fire X4600 M2 Server Architecture - White Paper - June 2008 - http://download.oracle.com/docs/cd/E19121-01/sf.x4600/
27. CPU cache - http://en.wikipedia.org/wiki/CPU_cache - (accessed on 13/07/2011)
28. Abdullah Kayi, Edward Kornkven, Tarek El-Ghazawi, Samy Al-Bahra, Gregory B. Newby, “Performance Evaluation of Clusters with ccNUMA Nodes - A Case Study”, Dept. of Electrical and Computer Engineering, The George Washington University, 978-0-7695-3352-0/08, 2008, IEEE, DOI 10.1109/HPCC.2008.111, p. 320
29. “Performance Guidelines for AMD Athlon™ 64 and AMD Opteron™ ccNUMA Multiprocessor Systems”, Publication no: 40555, Revision 3.00, June 2006, Advanced Micro Devices, Inc.
30. T. Li, D. Baumberger, and S. Hahn. “Efficient And Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin”. In PPoPP ’09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 65–74, New York, NY, USA,2009. ACM.
31. HECToR - http://www.hector.ac.uk/service/hardware/ - (accessed on 13/07/2011)
32. Jeff Larkin, Alan Minga – “Maximizing Application Performance on the Cray XE6” - http://www.slideshare.net/jefflarkin/hpcmpug2011-cray-tutorial - (accessed on 23/07/2011)
33. P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, B. Hughes - “Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor”, 10.1109/MM.2010.31, IEEE Computer Society
34. “Workload Management and Application Placement for the Cray Linux Environment”, Running Applications, S–2496–31, Cray.
35. Intel® Xeon® Processor E5504 - http://ark.intel.com/products/40711/Intel-Xeon-Processor-E5504-(4M-Cache-2_00-GHz-4_80-GTs-Intel-QPI) - (accessed on 23/07/2011)
36. Intel® Xeon® Processor 5600/5500 Series with the Intel® 5520 Chipset Overview - http://ark.intel.com/products/40711/Intel-Xeon-Processor-E5504-(4M-Cache-2_00-GHz-4_80-GTs-Intel-QPI) - (accessed on 26/07/2011)
37. D. Hackenberg, D. Molka, W. E. Nagel – “Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems”, Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01062 Dresden, Germany, ACM 978-1-60558-798-1/09/12, MICRO’09, December 12–16, 2009, New York, NY, USA
38. P. S. McCormick, R. K. Braithwaite, Wu-chun Feng - “Empirical Memory-Access Cost Models in Multicore NUMA Architectures”, LA-UR-11-10315
39. Z. Majo and T. R. Gross - “Memory System Performance in a NUMA Multicore Multiprocessor”, 2011 ACM 978-1-4503-0773-4/11/05, presented on SYSTOR’11.
40. H. Jin, R. Hood, J. Chang, J. Djomehri, D. Jespersen, K. Taylor, R. Biswas and P. Mehrotra - “Characterizing Application Performance Sensitivity to Resource Contention in Multicore

Architectures", NAS Technical Report NAS-09-002, November 2009, NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000

41. Abdullah Kayi, Yiyi Yao, Tarek El-Ghazawi, Greg Newby, "Experimental Evaluation of Emerging Multi-core Architectures", The George Washington University, Dept. of Electrical and Computer Engineering, 1-4244-0910-1, 2007, IEEE
42. Abdullah Kayi, Edward Kornkven, Tarek El-Ghazawi, Greg Newby, "Application Performance Tuning for Clusters with ccNUMA Nodes", Dept. of Electrical and Computer Engineering, The George Washington University, 978-0-7695-3193-9/08, 2008, IEEE, DOI 10.1109/CSE.2008.46, p. 245
43. T. Mytkowicz, A. Diwan, M. Hauswirth, P. F. Sweeney - "Producing Wrong Data Without Doing Anything Obviously Wrong!", ASPLOS'09, March 7–11, 2009, Washington, DC, USA, ACM 978-1-60558-215-3/09/03
44. A. Thuressona, N. Neufeldb - "Optimizing HLT code for run-time efficiency", LHCb-PUB-2010-017, Issue: 1, Lund, Sweden, CERN, PH, November 4, 2010.
45. D. Molka, D. Hackenberg, R. Schöne and M. S. Müller - "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System", Center for Information Services and High Performance Computing (ZIH), 1089-795X/09, DOI 10.1109/PACT.2009.22, pp. 261
46. dlopen - http://www.gnu.org/software/hello/manual/gnulib/dlopen.html#dlopen (accessed on 05/07/2011)
47. dlopen POSIX - http://pubs.opengroup.org/onlinepubs/9699919799/functions/dlopen.html (accessed on 05/07/2011)
48. Comma Separated Values - http://en.wikipedia.org/wiki/Comma-separated_values (accessed on 05/07/2011)
49. J. Cole perl script - http://jcole.us/blog/files/numa-maps-summary.pl (accessed on 07/09/2011)

Appendix A – Benchmark Suite Software Design

A.3.100 Requirement 100: “Enhancing the code of the benchmarks should be an easy task.”

In order to facilitate future work each benchmark (or test) is written in a separate source file. Each test is completely autonomous and requires only the presence of the common header files (see requirement 120 for a description of the header files) which define and implement common functions. Each test must have a function called *run()*, which displays the benchmark details [*printDesc()*], initialises the data arrays [*init()*], loads the appropriate data [*distribute()*], executes the benchmark [*execute()*] and writes the result in a file for post processing. The function *run()* must exist. If it does not exist then the loader executable will exit with an error. The other functions can exist or their functionality can be merged into other functions.

A demo code will be created which can be used as a pattern for future benchmarks.

A.3.110 Requirement 110: “The length (lines of code) for each benchmark should be limited.”

Using this design the length of each test is approximately 130 lines of code and the total length of the file is around 200 lines.

A.3.120 Requirement 120: “The benchmarks should make use of common functions to reduce the length of code and to reduce errors.”

In order to reduce the length of the files and to reduce errors, common functions will be moved to common files. The common files are *commonHeader.h*, *benchmark.h* and *defines.h*.

- *defines.h* contains global definitions used in all benchmarks. Variables like total number of threads and number of iterations are defined in this file. The values of these variables can also be defined during compilation.
- *commonHeader.h* contains the function definitions for the common functions. The common functions are presented in detail in the next table.
- *benchmark.h* contains the function definitions for the four main functions. These definitions could be moved to the file *commonHeader.h*.

The common functions used by the benchmark suite are the following:

Function Name	**Function Description**
double [or hrtime_t] getmytime()	This function returns either a *double* (or an *hrtime_t* for Solaris) which is used as a timestamp. It is used to find the current time, in order to keep accurate timestamps.
void printTime(double [or hrtime_t] time)	This function prints out the time. It takes as an argument a *double* (or an *hrtime_t* for Solaris). It is used to print out the time in both Linux and Solaris configurations.
void calculatePrintResults(..)	This function calculates and prints the duration of each benchmark step. It takes as arguments an array which contains the various time lapses of

	each benchmark, the number of used threads and the number of bytes transferred. It is used after the end of the benchmarks to calculate and print out the data. It also formats the data using the appropriate method.
char* getDescription()	This function returns a char* with the description of the benchmark currently being executed. It is used with the *setDescription()* function.
void setDescription(const char*)	This function sets the description of the benchmark currently being executed. It is used with the *getDescription()* function.
void writeToFile(..)	This function writes the array of timestamps to an output file. It takes as arguments the name of the output file, an array which contains the various time lapses of each benchmark, the number of used threads, the number of bytes transferred and the iteration number of each test. It also formats the data using the appropriate method.

Table 9 - Common functions

A.3.130 Requirement 130: "The user must be able to execute all benchmarks, a subset of benchmarks or a single benchmark."

In order to reduce used CPU cycles the user should be able to execute a subset or a single benchmark. In order to implement this requirement, the application program creates a list of available benchmarks. Using a white/black user defined list the application program only executes the required benchmarks. The black/white list requires more effort for implementation; it however provides added flexibility for the end user. The black/white list solution is selected because the user might execute only 2 out of 10 benchmarks in which case he will use the white list. If the user needs to execute 8 out of 10 benchmarks he will use the black list.

A.3.140 Requirement 140: "The user should be able to check how many benchmarks are available."

The user should be able to check how many benchmarks are available before running a benchmark session. The application program will use a command line switch (-l for list) to display the number of available benchmarks.

A.3.141 Requirement 141: "The user should be able to display the descriptions of the available benchmarks."

In addition to displaying the number of available benchmarks the user should be able to display a short (254 characters long) description of each benchmark. The application program will use a command line switch (-ld for list description) to display the number of available benchmarks and their description.

A.3.200 Requirement 200: "The benchmarks must use portable code."

The use of proprietary functions or libraries is not permitted. The source code for the benchmark suite must compile and execute properly in various software configurations. Some of the configurations might have reduced functionality and missing libraries. The code will follow the C standard as much as possible. The only elaborate functionality used is *dlopen()*[25]. *dlopen()* is used to dynamically load the benchmark library file. According to the GNU site [46], the *libgnu* functionality supports the use of this function. Furthermore according to the Open Group site [47], the POSIX standard defines this function. Since *libgnu* is available in many platforms and most Unix variants are POSIX compliant, we are confident that the use of the *dlopen()* functionality is portable.

A.3.210 Requirement 210: "The benchmark source files must have some comments which explain the purpose and the output of the benchmark."

The use of comments in the source files are used to meaningfully describe the functionality of the code. Due to the small file size (see requirement 110) and with the use of comments the functionality of the code can be easily understood.

A.3.300 Requirement 300: "The benchmarks must measure and quantify the NUMA effects."

The benchmarks are used as a tool to measure and quantify the NUMA effects on performance. Therefore, the benchmarks measure a particular scenario collecting relevant data. The scenarios differ in the following attributes:

- data distribution. Data are either distributed across the nodes or are just positioned on one node.
- data transfer. Data are transferred either in parallel or in serial. Parallel transfer means that all the threads access (read/write) data at the same time, thus the memory subsystem of the platform is stressed and results may not be easily replicated. Serial transfer means that each thread accesses (reads/writes) data one at a time, thus the memory subsystem of the platform should be able to cope with the transfer.
- data size. Smaller sized data fit easily in the L1 cache of the node, thus local accesses are considerably faster than remote. Medium sized data fit easily in the L2 cache of the node. Larger sized data do not fit in L2 cache, thus they are paged out in main memory.

A.3.310 Requirement 310: "At least one benchmark to measure latency/bandwidth must be written."

In order to measure and quantify the NUMA effects at least one benchmark is required to measure bandwidth/latency. However, in order to fully understand the dependencies and complexities of each hardware platform more than one benchmarks are required. For a detailed description of benchmarks refer to 3.1 Benchmarks.

A.3.400 Requirement 400: "The benchmarks should be compiled using a central mechanism (Makefile)."

In order to facilitate the usage of the benchmark suite, an automated central mechanism is used for compilation. A *Makefile* file is introduced. The *Makefile* creates compilation profiles for GNU gcc and PGI pgcc. The *Makefile* also enables basic optimization flags.

If no target is used, the benchmark libraries and the application program are compiled. This is the default behaviour and is used to compile the whole suite. The benchmarks are compiled as shared libraries which are dynamically loaded by the main application program.

If the *test* target is used, then the benchmarks are compiled as separate executables. This is only used for testing. The main application program is not compile and separate static executables are compiled, one for each benchmark, which can be run individually to test their functionality.

A.3.410 Requirement 410: "The user should be able to compile a single benchmark or compile all benchmarks."

The *Makefile* can be used to compile a single benchmark. If no benchmark target is used, then the whole benchmark suite is compiled. In order to compile a single benchmark, the user must call the *Makefile* using the name of the benchmark.

A.3.411 Requirement 411: "The compilation process could only compile changed files and ignore unchanged files."

In order to speed up the compilation process only changed source code files should be compiled. However if a header file is changed, then all benchmarks require recompilation due to the modular and shared design of the code.

A.3.420 Requirement 420: "An automake functionality could be used in order to check the existence of required libraries and compilers, before compilation starts."

The various Unix operating systems allow the use of the automake functionality. automake automatically parses a configuration file and tests the host environment for the required libraries and functionality. If all tests pass the *Makefile* is created automatically. Otherwise an error is issued describing the missing library or functionality. The automake functionality is not installed by default on every Unix operating system. Furthermore since the benchmark suite only uses standard and portable code should be able to compile and execute under all Unix environments. If there is enough time an automake configuration file will be delivered, however a working *Makefile* will also be avaialbe.

A.3.500 Requirement 500: "The benchmarks must output the results in a properly formatted text file."

The benchmark suite will be used to collect data from various experiments. Storing the data in order to analyse them is a critical functionality of the suite. Therefore the collected data will be saved in a file using the comma separated values format (*csv*) [48]. The *csv* format is used as the files can then be imported to various spread sheet applications for post processing.

A.3.510 Requirement 510: "If more than one benchmark is being executed, then the output file should be written to disk as soon as possible in order to avoid any data losses."

In case the application program crashes, each benchmark result could be saved on file as soon as possible. Even though the chance of a program crash is small, due to vigorous and extensive testing, each library is responsible in saving the results of each benchmark test on file. Thus if a benchmark crashes the results collected by the previous benchmarks are saved.

A.3.520 Requirement 520: "The format of the output file could be user configurable."

The default option is to use the comma separated values format (*csv*) (see requirement 500). However a user may have a particular application which accepts a different delimiter. Thus the option of using a different delimiter should be supported. Since the *csv* format is widely used, this requirement will only be implemented if a specific use case for a different delimiter is demonstrated by an end user.

Appendix B – Micro benchmarks usage

For Linux environments the following variables must be set:

```
export OMP_NUM_THREADS=X
export PSC_OMP_AFFINITY=TRUE
export PSC_OMP_AFFINITY_GLOBAL=TRUE
export OMP_WAIT_POLICY=ACTIVE
export MP_BIND=yes
export MP_BLIST=Y
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./
```

Figure 53 - Linux environment variables

The number of threads that we want to use should replace the *X* value. Furthermore, the identifiers of the processors that we want to measure should replace the *Y* value.

The benchmark should be executed using the *takset* command:

```
taskset –c Y ./bench.exe
```

Figure 54 - Linux execution

The same value of *MP_BLIST* should replace the *Y* during the execution. The benchmark will automatically load and execute all benchmarks found on the current directory.

For CLE environments the following variables must be set:

```
#!/bin/bash
#PBS -N newname
#PBS -l mppwidth=24
#PBS -l mppnppn=24
#PBS -l walltime=00:05:00
#PBS -A d04
#PBS -l ncpus=24

cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=X
unset PSC_OMP_AFFINITY
unset PSC_OMP_AFFINITY_GLOBAL
unset OMP_WAIT_POLICY
unset MP_BIND
unset MP_BLIST
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./:/opt/gcc/4.5.3/snos/lib64/
```

Figure 55 - CLE environment variables

The number of threads that we want to use should replace the *X* value.

The benchmark should be executed using the *aprun* command:

```
aprun –n 1 –d X –cc Y ./bench.exe
```

Figure 56 - CLE execution

The number of threads that we want to use should replace the *X* value. Furthermore, the identifiers of the processors that we want to measure should replace the *Y* value. The benchmark will automatically load and execute all benchmarks found on the current directory.