Master Thesis

December 2009

Master in Computer Engineering

Inter-motherboard Memory Scheduling

Author: **Mónica Serrano Gómez**

Advisors: **Julio Sahuquillo Borrás**
Houcine Hassan Mohamed

Contents

Chapter 1

Introduction

1.1 Parallel Computer Architectures

The demand for even more computer power to deal with high-performance computing has been continuously increasing during the last decades. Message-passing based systems have been the commonly used approach; however, these systems not only lead to high latencies when implemented in very large machines (e. g., BlueGene/P can be scaled to an 884,736-processor[1]) but also this approach difficults the programming of parallel applications.

Because of both reasons, industry has moved to shared memory systems for small to medium number of processors, which can be classified in two main categories. In the first category, *Symmetric shared-Memory Multiprocessors* (SMP) are relatively expensive and they do not scale to large sizes (i. e., larger than 32 nodes) since they use a common shared bus to access to main memory. Projects working on shared memory with coherent cache, like the NumaChip by Dolphin Interconnect Solutions [2], suffer from limited scalability introduced by the coherence protocol. Thus, a major concern is that the access to remote memory become affordable, and efficient both regarding to latency and price. In the second category, *Distributed Shared Memory* (DSM) [3] provide a virtual address space shared among processes running on loosely coupled processors.

Both kind of shared-memory architectures offer advantages such as ease of programming and portability achieved through the shared memory programming paradigm. However, DSMs also include the lower cost of distributed-memory machines, and higher scalability than SMPs.

On the other hand, current technology constraints have moved chip manufacturers from complex cores to simpler multicore based processors [4]. In this context, it is common to find several multicores sharing the same motherboard. Furthermore, it is expected that motherboards include a number of processors at least one order of magnitude higher in the near future. Their affordable price and their potential computational power when running parallel workloads, has lead the industry to use these motherboards as clusters to implement parallel computers.

In summary, it is common to find a parallel machine consisting of a set of motherboards connected by an interconnection network. In such a system, each motherboard can be seen as a block of a cluster. Unfortunately, the shared memory space that can be seen by a processor is limited to the available memory inside the motherboard. We refer to this kind of machine, i.e., a cluster of DSMs, as the original machine.

This work is a part of a wider research project working in a real cluster of DSMs machine where the main goals are: i) to extend the HyperTransport protocol in order to allow a processor to access to the memory modules in another motherboards (i. e., to enlarge the shared memory space); ii)to devise new memory scheduling algorithms targeted to the new machine.

Compared to the original, our approach offers the benefit of being able to have extra memory available from other motherboards. Besides, most applications take advantage of having more memory resources but do not need more computing nodes. So they can perform without memory coherence.[5, 6], which is the key to achieve scalability. The most expensive part– coherent cache among motherboards– could be removed while keeping the main function: shared memory.

In other words, the aim of this project is to enlarge the shared memory space and handle it in an efficient way. However, computational resources management is out of the scope of this work. Nevertheless, this drawback will be mitigated as the number of processors per board is expected to increase in the near future.

1.2 Motivation

Large scientific parallel applications demand large amounts of memory space. Current parallel computing platforms schedule jobs without fully knowing their memory requirements. This leads to uneven memory allocation in which some nodes are overloaded. This, in turn, leads to disk paging, which is expensive in the context of scientific parallel computing.

Data intensive applications, such as data mining and ad hoc query processing databases, are considered very important for massively parallel processors, as well as conventional scientific calculations. Thus, investigating the feasibility of data intensive applications on a PC cluster is meaningful. Association rule mining, one of the best-known problems in data mining, differs from conventional scientific calculations in its usage of main memory. It allocates many small data areas in main memory, and the number of those areas suddenly grows enormously during execution. As a result, the contents of memory must be swapped out if the requirement for memory space exceeds the real memory size. However, because the size of each data area is rather small and the elements are accessed almost at random, swapping out to a storage device must degrade the performance severely.

A straightforward solution to the previously commented problem is to oversize RAM memory in the nodes; however, this solution may be prohibitive as RAM memory is one of the most expensive resources on high performance computers. On the contrary, the solution proposed in this work is able to perform without extra resources because it takes advantage from the spare memory in some of the other boards. In fact, our approach could even do better than oversizing, with regard to performance. See the example below:

> Take a node with an amount of memory X, in which an application requiring an amount Z, $X < Z$, is running. Let's analyze the implications which would have each of the two choices above. Increasing the node's local memory up to Y, being $Y < Z$, would imply to borrow the memory exceeding Y from hard disk. On the other hand, just taking Z-X from another board (which does not need it) would avoid accessing to hard disk, whose latency is several orders of magnitude greater than accessing to RAM memory.

IBM z series [7] and HP Integrity Superdome [8] mainframes are examples of shared-memory machines with an amount of memory that can be as large as two Terabytes, and thus expensive. This research is focused on making the right modifications on the system in order to be able to lend/ borrow RAM memory among motherboards, in an efficient way. Starting from what has been stated up to this point, we aim at taking advantage of free memory on certain computers in order to increase the available memory for a given application located on another motherboard; saving the cost of producing machines with oversized memory.

Furthermore, the results of this work lead us to conclude that by using simple hardware (just a couple of counters), the scheduler could access to the required information in order to dynamically schedule the memory modules budget among the applications so that the highest performance for the whole system is provided.

1.3 Magnitude of the Work

This wide scope research requires from several groups to carry out the whole project. There are people from different universities (Universidad Politï¿½cnica de Valencia, Universidad del Valencia, University of Heidelberg) which are taking part in its development. Each working group deals with a set of tasks which range from the physical to the logical layer.

1.3.1 Industry Application

The described idea is currently carried out on a real machine (prototype) detailed in the next chapter. The main goal behind this implementation apart from simulation is to allow us to perform ”demos” in order to bring the attention of machine manufacturers. HyperTransport [9] is currently the lowest latency, highest bandwidth openly licensed standard communication technology for chip-to-chip and board-to-board interconnects. AMD Opteron uses HyperTransport to interconnect the processors in a motherboard.

Because of the simplicity and broad application of our mechanism, a product able to be introduced in commercial systems could be offered, with the subsequent interest for companies worldwide. Regarding to the devised HyperTransport extensions, they keep compatibility with the standard.

1.3.2 Standing of the Research Group

Universidad Politï¿½cnica de Valencia has become an academic member of *HyperTransport Consortium* with special conditions to take part in discussions of *Technical Working Group* (TWG).

Besides, *Advanced Technology Group* (ATG) inside HyperTransport Consortium has been set up. It is formed by:

- The most important researching academic groups on the ground of interconnection networks:
  - Mannheim University
  - Georgia Institute of Technology
  - Simula Research Laboratory
- Engineers from the most representative enterprises of HyperTransport Consortium
- TWG President
- Leader: Parallel Architecture Group Director.

Chapter 2

Proposed Approach

This chapter is focused on describing the proposed approach. This description is structured in three main parts: real system, motherboards connection and memory scheduling.

2.1 Real System

The tasks of this project are carried out on a cluster prototype which is equipped with HyperTransport technology. The main characteristics of this system are listed below.

2.1.1 System Characteristics

Manufacturer: Supermicro
Format: Rack
Number of nodes: 64 nodes with 16 cores Opteron 2.0 GHz
Interconnection between nodes: Gigabit Ethernet, HT with High Node count extensions by F.O.
Local Hard Disk: 250GB Sata2
RAM Memory: 16GB
Operating System: RHEL5.1

2.1.2 Standard HyperTransport

HyperTransport is used by AMD Opteron [10] to interconnect the processors in a motherboard. In these systems, each processor requires to know where a memory request must be fordwarded. This is achieved by including in each processor a set of registers configured at the initialization phase that reflect the system physical memory distribution. In this way, when a processor issues a load or store operation related to a given memory location, the processor compares the requested address with those registers, and then forwards the memory operation to the memory controller handling that memory address, provided by the previous comparison. Forwarding the memory operation involves the generation of a HyperTransport message.

2.2 Connecting Motherboards To Access Remote Memory

A process must be able to see not only the memory on its board but also other board's, so a new hardware component has been devised to make it possible. The so-called RMC (Remote Memory Controller) will implement the required functionality. The system described at Section 2.1.2 is the basis upon which the technology that enables the access to remote memory will be designed. The new component will be seen by the processors in the motherboard as a new memory controller. However, the RMC will not be a typical memory controller as it has no memory banks directly connected to it, otherwise it relies on the memory banks installed in other nodes in the cluster. To enable the RMC functionality, the registers mentioned above must be reconfigured so that some of the memory accesses (i. e., those accessing to memory located in other motherboard) are directly forwarded to the RMC, that will convert those accesses into remote accesses. RMC will have a regular HyperTransport interface to the local node and a High Node Count HyperTransport [11] interface to the rest of the cluster and it will be attached to the motherboard of the nodes in the cluster by means of HTX compatible cards designed by University of Heidelberg [12].

2.3 Memory Scheduling

2.3.1 Background

In the literature, different research papers dealing with remote memory allocation mostly related to memory swap can be found. Oleszkiewicz et al. propose a peer-to-peer solution called parallel network RAM [13]. This approach avoids the use of disk and better utilizes available RAM resources in a cluster. This approach reduces the computational, communication and synchronization overhead typically involved in parallel applications. Shuang et al. design a remote paging system for remote memory utilization in InfiniBand clusters [14]. They present the design and implementation of a high performance networking block device (HPBD) over InfiniBand fabric, which serves as a swap device of kernel virtual memory (VM) system for efficient page transfer to/from remote memory servers. They demonstrate that quick sort performs 1.45 times slower than local memory system, and up to 21 times faster than local disk. Oguchi et al. investigate the feasibility of using available remote nodes' memory as a swap area when application execution nodes need to swap out their real memory contents during the execution of parallel data mining on PC clusters [15]. In this work application execution nodes acquire extra memory dynamically from several available remote nodes through an ATM network. Experimental results on a PC cluster show that the proposed method is considerably better than using hard disks as a swapping device. In [16] the use of remote memory for virtual memory swapping in a cluster computer is described. The design uses a lightweight kernel-to-kernel communications channel for fast, efficient data transfer. Performance tests are made to compare the proposed system to normal hard disk swapping. The tests show significantly improved performance when data access is random. Oguchi et al. [17] investigated the feasibility of using available idle nodes' memory as a swap area when some nodes need to swap out its real memory contents, during the execution of parallel data mining on PC clusters. In this paper, they report results in which application executing nodes acquire extra-memory dynamically from several available idle nodes through ATM network. Their results improve considerably a solution using hard disks swapping techniques. Midorikawa et al. propose the distributed large memory system (DLM), which provides very large virtual memory by using remote memory distributed over the nodes in a cluster [18]. The performance of DLM programs using remote memory is compared to ordinary programs using local memory. The results of STREAM, NPB and Himeno benchmarks show that the DLM achieves better performance than other remote paging schemes using a block swap device to access remote memory. DLM is a user-level software without the need for special hardware. To obtain high performance, the DLM can tune its parameters independently from kernel swap parameters.

2.3.2 Scheduling Local and Remote Nodes

All applications running in a system are not affected in the same way by memory accesses. There are some applications called *memory-hungry* or *memory-bounded* in which the time to complete a given computational problem is primarily driven by the amount of fast memory available to hold data. In other words, the limiting factor of solving a given problem is the average memory access time and so it is a critical issue for the application performance. Distributing the memory modules in the same ratio among the applications running in different nodes could provide poor performance for this kind of applications.

Moreover, a given application uses to have memory requirements that are dynamically changing through the execution time; depending on the part of the application where data are mostly accessed.

Therefore, there is a need to schedule memory modules among applications in order to maximize the system performance of the whole system. A first and simpler approach to tackle this concern would be a static approach. That is, the scheduler would provide all the memory modules assigned to a given application before starting its execution. This work is aimed at exploring the potential of this approach. To this end, we analyze several choices or heuristics which an hypothetical RAM Memory Scheduling Algorithm could have considered. As shown in results in Chapter 4, depending on the assignment criterion, performances vary indeed.

Chapter 3

Modeling The System

3.1 Introduction

To study the impact on performance and due to practical constraints (some parts of the system are not properly working yet), our approach has been decided to be evaluated through simulation. The system has been modeled with SIMICS 3.0.31 and extended by GEMS 2.1.

Other members of the research group are using or have used Simics 2.X with Solaris as Target Operating System. This system was not used because of the reasons stated further in this introduction section.

Several aspects such as interaction with the other working groups had influence on all the decisions which have been made along the system modeling, as described below.

At this point, we are working on regular processes although we are starting to work with Virtual Machines, which can be seen as a kind of process. After a deep analysis of some virtualization solutions, in which *Universidad de Valencia* also takes part, we finally concluded that KVM (Kernel-based Virtual Machine) is the most suitable choice for our purpose.

The kernel component of KVM is included in mainline Linux, as of 2.6.20. So a Linux based Operating System with 2.6 kernel version has to be modeled in order to get closer to the real system we are dealing with. Simics Version 2.X caused several problems when trying to simulate a 2.6 Linux Kernel; but we managed to do it on Simics 3, which is more up-to-date. Among all the possibilities that Simics offers, we selected the one described below.

3.2 Simics Target: Sunfire

The memory system has to be evaluated, so it is necessary to use Simics module GEMS (it will be described further on the text), which implies using Sparc Architecture. In addition, Linux is the desired target Operating System, as KVM is going to be used at the next researching step.

Because of the mentioned requirements, Simics/SunFire has been chosen from the available Simics targets, as it fits all the needs quite well. On one hand, it models the Sun Enterprise 3500 - 6500 class of servers. A SunFire server can be configured with up to 30 UltraSPARC II processors and 60GB of memory. On the other hand, both Solaris and Linux are supported as target Operating Systems.

3.2.1 Sunfire Simulated Machine: Cashew

- Cashew is a Sun Enterprise 6500 server with a single UltraSPARC II processor running at 168 MHz with 256 MB of memory. It has one Ethernet adapter, one SCSI disk and one SCSI CD-ROM.
    - Cashew is configured for an existing Aurora Linux 2.0 disk dump, that can be downloaded from the Virtutech web site.
    - Additional information:

      * Aurora 2.0 Linux (Fedora Cora 3), installed directly on Simics.
      * Linux kernel 2.6.13

* SimicsFS support.
* Configured to get IP address using DHCP.

3.3 Additional Modules: Gems

GEMS [19] is a set of modules for Virtutech Simics that enables detailed simulation of multiprocessor systems, including Chip-Multiprocessors (CMPs).

It leverages the power of Virtutech Simics to simulate a *Sparc* multiprocessor system.

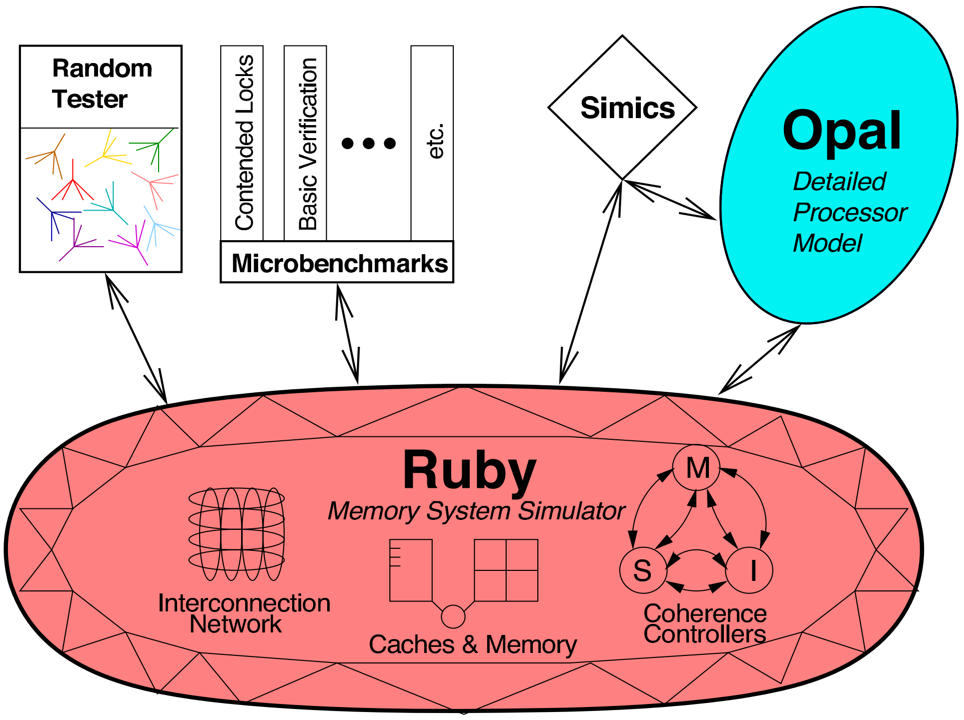


Figure 3.1: A view of the GEMS architecture. Ruby, the memory simulator.

3.3.1 Submodules: Ruby

Ruby is a timing simulator of a multiprocessor memory system that models: caches, cache controllers, system interconnect, memory controllers, and banks of main memory. It combines hard-coded timing simulation for components that are largely independent of the cache coherence protocol (e.g., the interconnection network) with the ability to specify the protocol-dependent components (e.g., cache controllers) in a domain-specific language called SLICC (Specification Language for Implementing Cache Coherence).

Some modifications to the original Ruby code have been necessary in order to adapt it to the requirements. The main change consists in implementing the Remote Memory Controller. These changes not only imply code extensions on Ruby itself but also on the coherence protocol. RMC functions have been added to *AMD Hammer protocol*, which has been chosen in order to model a system closely resembling the Opteron processor.

Ruby code is also extended with support to configure memory scheduling. Local to Node, Local to Board and Remote memory regions were defined (See description of these regions in Section 3.4). Some parameters are added to model the assignment of the desired amount of memory to each region.

3.4 System Model

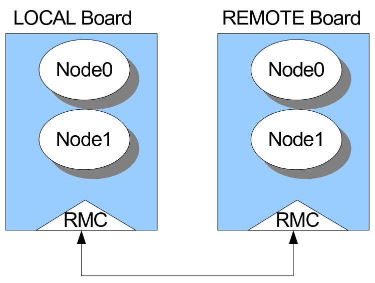


Figure 3.2: System Model. Each node has both processor and memory.

The whole system has been scaled down to achieve a reasonable simulation time. The diagram of the modeled system is shown in Fig. 3.2. Its main characteristics are listed below:

- The system is composed of two motherboards, which is the minimum configuration to let us define both local and remote memory. We agreed to set three different regions of memory:
  - *Local to Node:* Number of memory modules which are located in the processor in which the application is launched (see node0).
  - *Local to Board:* Number of memory modules which are located on the same board but in the other processor .
  - *Remote:* Number of memory modules which are located in the other motherboard.
- *Number of Processors*: 2 per motherboard
- *Number of Memory Modules*: 32. 8 memory modules per processor, which will be distributed among the applications in the system as appropriate. Each different combination will be subjected to analysis.

- *L1 Cache Rows*: 2
- *L1 Cache Number of Sets:* $2^9 = 512$
- *L2 Cache Rows*: 16
- *L1 Cache Number of Sets:* $2^{10} = 1024$
- *Block Size (bytes)*: 64
- *Memory Size (bytes):* 268435456
- *Network Topology:* Non-Uniform Cache Access architecture network (NUCA) [20]
- *Local Cache Latency:* 1
- *Other Node's Cache Latency*: 1
- *Local Memory Latency*: 100
- *Remote Memory Latency:* 2700
- RMC has been assumed with no internal storage capacity (i.e, cache memory) during all the experiments.
- *Coherence Protocol:* AMD Hammer protocol.
    - Extended with RMC functionality.

All the latencies are expressed in cycles.

3.4.1 Binding a Process to a Specific CPU

In order to control local and remote execution, we overrided the kernel's process scheduling and bind a certain process to a specific CPU on the DSM system. The Schedutils command *taskset* is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new COMMAND with a given CPU affinity. So we used it to our purpose; that is, each time an application is launched, one of the nodes is selected as the local node.

By editing GEMS Network Files and specifically creating certain Ruby parameters and functions, we managed to locate the desired number of memory modules in each either node or motherboard. In this way, we have full control of memory distribution on the system.

3.4.2 Workload Characterization

This work aims to deduce how the specified memory parameters impact on performance, depending on the workload. To this end, we selected four different benchmarks to carry out the experiments.

Selecting Benchmarks

We are mainly interested on memory access, so the characteristics that have been taken into account for all the considered workloads (Stream, SPLASH suite) are:

Total Instructions
Total Reads
Total Writes
Cache Miss Rate

Workload Description

Stream [21]

The STREAM benchmark is a synthetic benchmark program, written in standard Fortran 77 (with a corresponding version in C). It measures the performances of four long vector operations. These operations are: *Copy:* measures transfer rates in the absence of arithmetic.

Scale: adds a simple arithmetic operation.

Sum: adds a third operand to allow multiple load/store
ports on vector machines to be tested.

Triad: allows chained/overlapped/fused multiply/add operations. These operations are representative of the "building blocks" of long vector operations. The array sizes are defined so that each array is larger than the cache of the machine to be tested, and the code is structured so that data re-use is not performed. The intent of STREAM is not to suggest that "real" applications have no data re-use, but rather to decouple the measurement of the memory subsystem from the hypothetical "peak" performance of the machine.

Before designing the experiments aimed at exploring different memory scheduling policies, we run experiments for each individual benchmark in order to select the appropriate problem size. For illustrative purpose, we only show two of them, that is, Stream and FFT benchmarks.

As explained in Section 3.4.1, thanks to *taskset* command we are able to control local and remote execution. So we defined a system with only one motherboard containing two nodes (node 1 and node 2) two memory modules, and run experiments in which we varied the local node (where application is run) and remote node (the other node in the system), as well as the node which contained the two memory modules . That is:

1. Application is running in node 1.

    (a) Memory in node 1.
    (b) Memory in node 2.

2. Application is running in node 2.

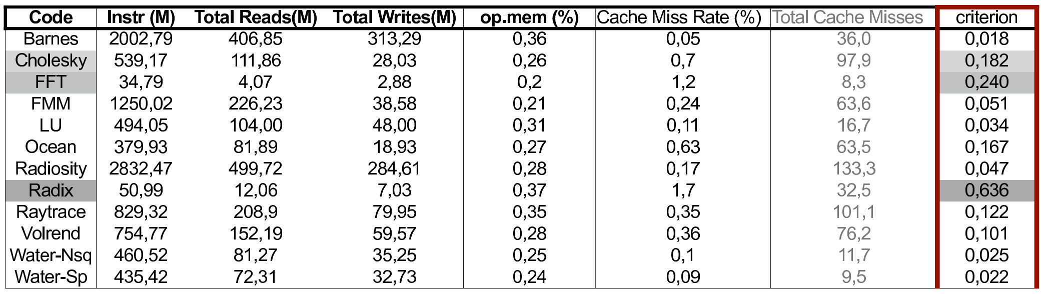
    (a) Memory in node 1.
    (b) Memory in node 2.

In this way, the performance improvement between sub-cases (a) and (b) was deduced. We proceeded similarly for different vector sizes, searching for a size for which this improvement was as similar as possible for both cases 1 and 2 above. Firstly, we did a coarse grain analysis of sizes, and then, a fine grain one (See Fig. 3.3) to find the proper size.

From Fig. 3.3 we can see that 275K is the lower vector size which leads to an improvement percentage stabilization. We chose the lower one as simulation time considerably grows with vector size.

SPLASH-2 The SPLASH (*Stanford Parallel Applications for Shared Memory*) [22][23] benchmark suite has been selected because it is commonly used to evaluate the performance of multiprocessor systems. This suite offers several kernels and applications which were evaluated regarding the *Choosing Criterion* presented in Fig. 3.4.

Source: [24]
Cache line size: 64 bytes
***Criterion*=** (Total Reads + Total Writes) / Instr) * Cache Miss Rate



Code	Instr (M)	Total Reads(M)	Total Writes(M)	op.mem (%)	Cache Miss Rate (%)	Total Cache Misses	criterion
Barnes	2002,79	406,85	313,29	0,36	0,05	36,0	0,018
Cholesky	539,17	111,86	28,03	0,26	0,7	97,9	0,182
FFT	34,79	4,07	2,88	0,2	1,2	8,3	0,240
FMM	1250,02	226,23	38,58	0,21	0,24	63,6	0,051
LU	494,05	104,00	48,00	0,31	0,11	16,7	0,034
Ocean	379,93	81,89	18,93	0,27	0,63	63,5	0,167
Radiosity	2832,47	499,72	284,61	0,28	0,17	133,3	0,047
Radix	50,99	12,06	7,03	0,37	1,7	32,5	0,636
Raytrace	829,32	208,9	79,95	0,35	0,35	101,1	0,122
Volrend	754,77	152,19	59,57	0,28	0,36	76,2	0,101
Water-Nsq	460,52	81,27	35,25	0,25	0,1	11,7	0,025
Water-Sp	435,42	72,31	32,73	0,24	0,09	9,5	0,022

Figure 3.4: Splash2 benchmarks characteristics.

From the data shown in Fig. 3.4 we can conclude that the best SPLASH workloads for our study are Radix (criterion value= 0.636), FFT (criterion value= 0.240) and Cholesky (criterion value= 0.182), as they have the higher values for the established criterion.

FFT The FFT program is a complex, one-dimensional version of the Six-Step FFT described in [25]. Four command-line parameters must be specified: the number of points to transform, the number of processors, the log base 2 of the cache line size, and the number of cache lines. The number of data points must be an even power of 2. The number of processors must be a power of 2.

The method for choosing the most suitable matrix size (number of points to transform) was the same as the one explained for Stream benchmark. The results obtained for FFT are shown at Fig. 3.5.

log2 matrix size	#taskset mask	#node	Ruby_cycles	performance improvement(%)
10	1	1	892063	2,74
	1	2	916541	
	2	1	932083	3,06
	2	2	904381	
12	1	1	3731952	1,73
	1	2	3796473	
	2	1	3796584	1,79
	2	2	3729707	
14	1	1	15987400	1,38
	1	2	16207622	
	2	1	16202674	1,37
	2	2	15983931	
16	1	1	80043074	10,53
	1	2	88469080	
	2	1	88693217	10,56
	2	2	80222120	
18	1	1	351999040	7,67
	1	2	378996703	
	2	1	379378404	7,69
	2	2	352277145	

Figure 3.5: FFT Size Fine Grain Analysis

So, we chose $2^{16} = 65536$ total complex data points transformed.

Regarding the rest of the command line options, we used the following:

- 1 processor (default value)
- 1024 cache lines.
- $2^6 = 64$ cache line length in bytes.

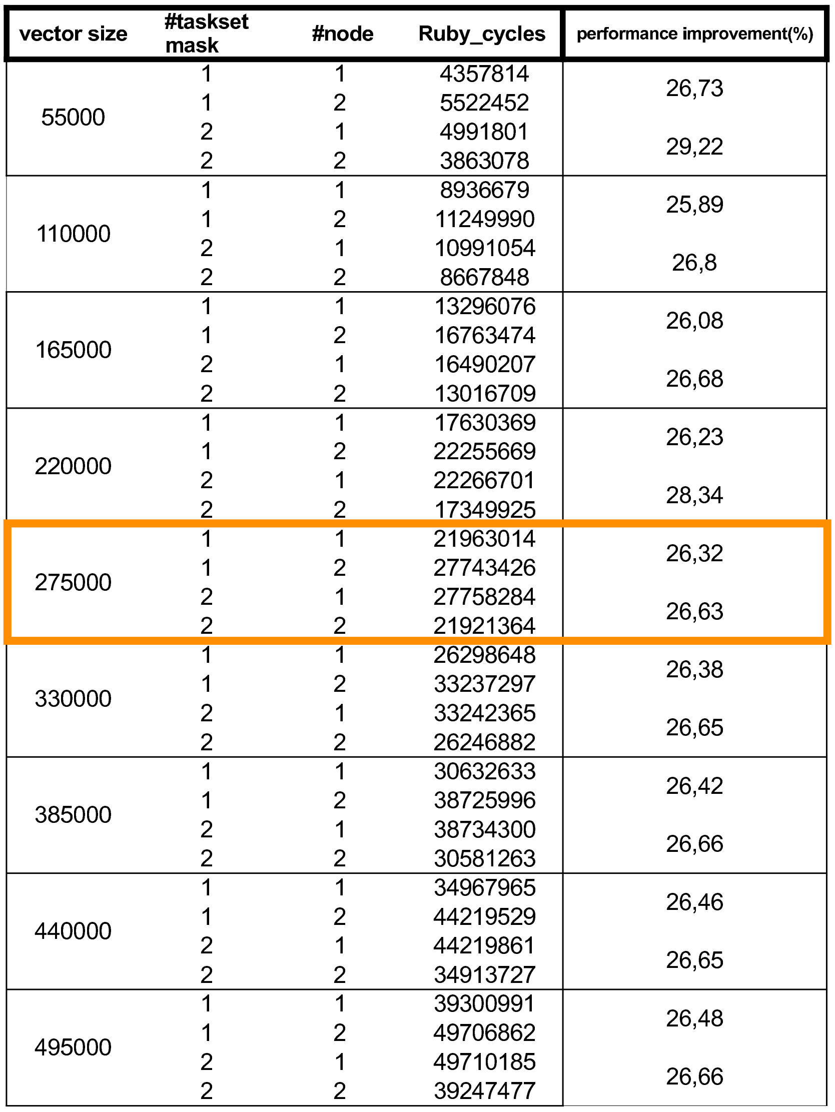
Radix The RADIX program implements an integer radix sort based on the method described in [26]. Four command line parameters can be specified. The number of keys to sort, the radix for sorting, and the number of processors are those parameters that would normally be changed. The radix used for sorting must be a power of 2. The values of these parameters are listed below:

- 1 processor (default value)
- Radix for sorting (Must be power of 2)= 1024 (default value)
- Number of keys to sort (Must be power of 2)= 524288= 1M/2
  - The value of this parameter used for evaluation at [24] is 1M integers. However, when executing the application with this size, we got excessive simulation times, so we agreed to use 50% of 1M.
- Maximum key value= 524288(default value). Integer keys k will be generated such that $0 <= k <= 524288$.

Cholesky The blocked sparse Cholesky factorization kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose.

The size of the cache (in bytes) should be specified on the command line, as well as the number of processors being used. The postpass partition size should be kept at the default value of 32.

- Postpass partition size= 32(default value)
- Cache size in bytes= 65536
- 1 processor (default value)
- Input file= tk15.O



vector size	#taskset mask	#node	Ruby_cycles	performance improvement(%)
55000	1	1	4357814	26,73
	1	2	5522452	
	2	1	4991801	29,22
	2	2	3863078	
110000	1	1	8936679	25,89
	1	2	11249990	
	2	1	10991054	26,8
	2	2	8667848	
165000	1	1	13296076	26,08
	1	2	16763474	
	2	1	16490207	26,68
	2	2	13016709	
220000	1	1	17630369	26,23
	1	2	22255669	
	2	1	22266701	28,34
	2	2	17349925	
275000	1	1	21963014	26,32
	1	2	27743426	
	2	1	27758284	26,63
	2	2	21921364	
330000	1	1	26298648	26,38
	1	2	33237297	
	2	1	33242365	26,65
	2	2	26246882	
385000	1	1	30632633	26,42
	1	2	38725996	
	2	1	38734300	26,66
	2	2	30581263	
440000	1	1	34967965	26,46
	1	2	44219529	
	2	1	44219861	26,65
	2	2	34913727	
495000	1	1	39300991	26,48
	1	2	49706862	
	2	1	49710185	26,66
	2	2	39247477	

Figure 3.3: Stream Size Fine Grain Analysis.

Chapter 4

Experimental Results

Several distributions of memory modules on three different memory regions –local (i. e., local to the processor, refered as L), local to board (refered as Lb), remote (refered as R)– are defined. Once the system is configured with a given distribution, analysis is made for each pair of benchmarks, that is, every experiment consists of two applications running on the system, each one on a different processor of the board. The system is assumed to include 32 modules, which must be scheduled to the benchmarks. As modules reside in three different places, many combinations appear when scheduling them among the workload. To simplify the large amount of alternatives, we assumed that both applications have the same number of memory modules. In this way, a given scheduling for an application only combines with a single scheduling of the other one. These alternatives are shown in Fig. 4.1 for both applications concurrently running.

4.1 Environment

As we can see in Fig. 4.2, the studied system is composed of:

- 2 motherboards
- 2 nodes per motherboard

Application1	Application2
(L, Lb, R)	(L, Lb, R)
(0,0,16)	(8,8,0)
(0,4,12)	(4,8,4)
(0,8,8)	(0,8,8)
(2,2,12)	(6,6,4)
(2,6,8)	(2,6,8)
(4,0,12)	(8,4,4)
(4,4,8)	(4,4,8)
(4,8,4)	(0,4,12)
(6,2,8)	(6,2,8)
(6,6,4)	(2,2,12)
(8,8,0)	(0,0,16)
(8,0,8)	(8,0,8)
(8,4,4)	(4,0,12)

Figure 4.1: Scheduling alternatives considered.

- 8 memory modules per node (refered as “Directory” in the figure)
- Total Memory Modules of the system: 32

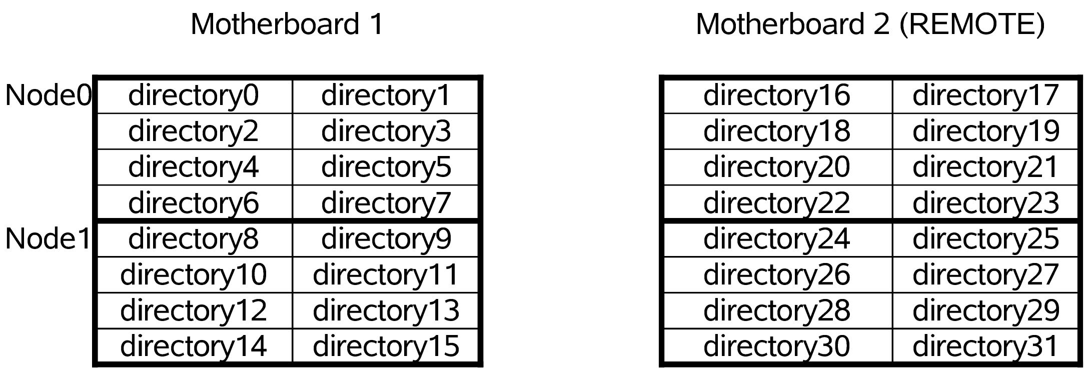
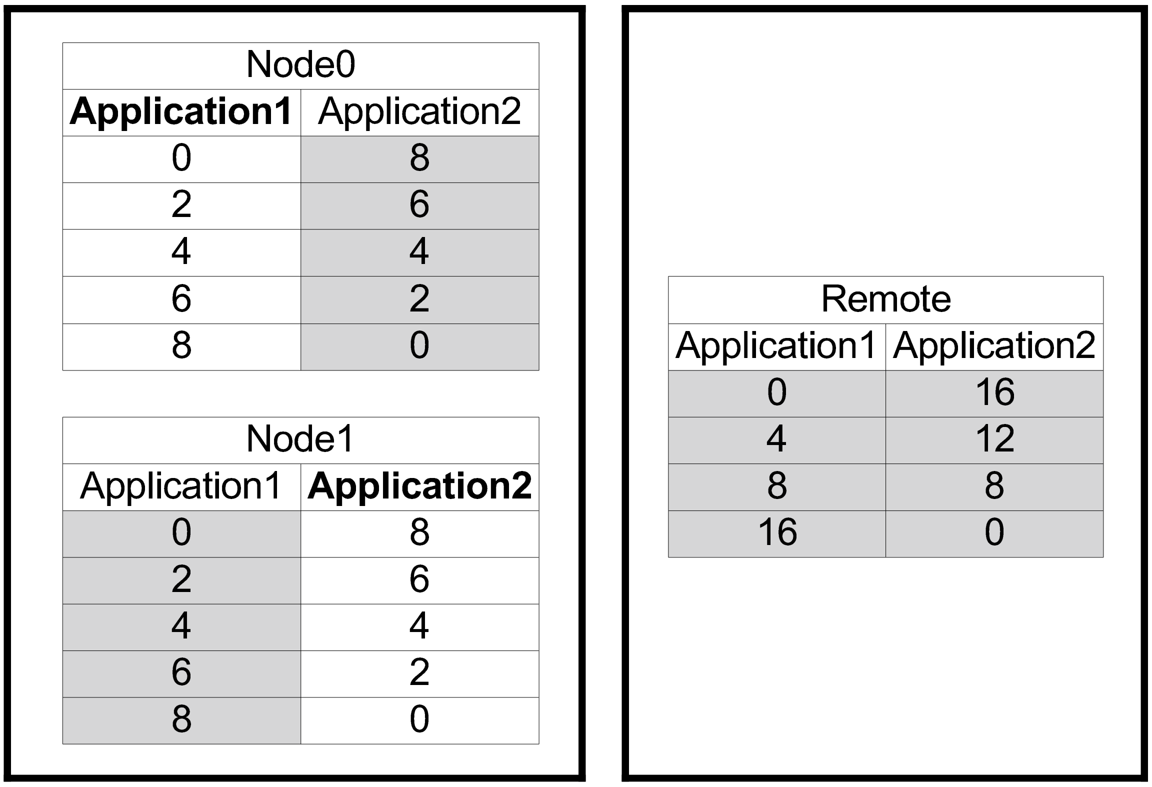


Figure 4.2: Modeled System

In this study, as mentioned above, there are two applications concurrently running on the system. Each application is launched on one of the nodes of Motherboard 1 (see Fig. 4.3): Node0 is the local node to Application1 and Node1 is the local node to Application2. Each application does not necessarily use the eight memory modules on its local node: it can use 0, 2, 4, 6 or 8 modules, leaving the remaining memory (grey-shaded on Fig. 4.3) available to be borrowed to the other application. The remote motherboard has the same node structure as the local motherboard, but it is assumed to be a black box with 16 memory modules. By defining a memory scheduling providing less or more remote memory modules, the performance of the corresponding application would be benefited or adversary affected, as local memory is faster than remote one. The aim of this study is to deduce the way in which the overall performance is affected depending on the available local and remote memory assigned to each of the two applications running concurrently on the system.



Node0	
Application1	Application2
0	8
2	6
4	4
6	2
8	0

Node1	
Application1	**Application2**
0	8
2	6
4	4
6	2
8	0

Remote	
Application1	Application2
0	16
4	12
8	8
16	0

Figure 4.3: Number of modules per node/motherboard and application

4.2 Evaluation Criteria

From each experiment, we gathered a statistical file. It contains some useful data such as execution time, total instructions executed, total cache misses, cache miss rate... per node and per application. These parameters provide useful information to evaluate the memory scheduling from different points of view. Different criteria have been explored for performance comparison purposes.

The optimum memory scheduling is computed on the basis of each different criterion. The priority of each benchmark can be estimated attending to the impact of giving more or less local memory to that benchmark depending on the application running on the other processor. In this way, the four benchmarks can be ordered with respect to its priority. The different priority orders obtained from each criterion are compared and help us to understand the influence of the memory scheduling in the overall performance (i. e., considering both co-runners).

Firstly, the most intuitive criterion to use is Execution Time. For each memory scheduling alternative, the sum of both execution times was taken as the value to identify the optimum case. In this case, the higher priority corresponded to FFT benchmark, as the optimum of the combinations of FFT with any other benchmark always implies as much as local memory to FFT, with the corresponding degrading (i. e., more remote memory space) to the other one of the pair. The following benchmark regarding to priority was Stream, then Radix, and finally Cholesky.

To corroborate these results we used the L2 cache miss rate criterion. This criterion was used since accessing to other motherboards has a strong impact on performance (i. e., 2700- cycle latency). In this case, the deduced priority order slightly differed the one described above for the Execution

Time. Stream seemed to have the higher priority and then the three SPLASH benchmarks ordered in the same way: FFT, Radix and the last one Cholesky. Notice that this order differs from the provided when applying the Execution Time criterion.

Finally, the IPC criterion was considered in order to explain the anomalous Stream behaviour. IPC measures the processing speed. The aspects which have influence on IPC are, among others, the machine characteristics (branch predictor, in-order or out-of-order execution, issue-width, ...) and workload characteristics (dependance chains of a load instruction, Floating Point or Integer instructions, branch instructions, ...). We found why Stream has a characteristic behaviour: the main problem is that its IPC is, in general, much lower (about one order of magnitude) than its co-running SPLASH benchmarks. For instance, as shown in Fig. 4.4, for the (8,0,8) configuration, Stream has an IPC of 0.029 while its co-runner FFT has a value of 0.196. Stream takes, on average, hardly 50 cycles to execute an instruction, so giving more local memory space to it does not benefit the overall performance as much as giving more local memory space to any of its co-runners. In this way, the best memory scheduling which includes Stream, will always give all the local (i. e., local to node and local to board) memory to its co-runner, forcing Stream to fully run on remote memory space.

As being the stronger criterion, only the IPC results will be shown in the next section.

4.3 Optimization

Results of all the experiments are shown at Figs. 4.4 to 4.9. Each of these figures shows a set of memory scheduling combinations for a pair of applications concurrently running in the system and should be interpret as follows:

For each of the two applications in the system, the number of available memory modules of the three types assumed (Local, Local to Motherboard, Remote) are defined by the 3-tuple (L, Lb, R). Each 3-tuple of an application only matches a single 3-tuple of the other one, as we assume that each application is provided with 16 modules.

Each line in the table represents a different evaluated memory scheduling. Performances in Processor Cycles as well as the Total Instructions Executed have been obtained for each different memory scheduling. From these two parameters, the Instructions Per Cycle value is calculated, for each application executed as well as the sum of them.

Finally, the best case is highlighted in bold in each IPC column.

To see more clearly the effect of scheduling on performance, some representative memory modules distributions from each pair of applications have been graphically represented in Fig. 4.10. The total of local memory modules of the first application is on the x axis. Under each value of the x axis, the corresponding scheduling for each application is specified. The IPC is on the y axis. In this way, it can be observed how the IPC trend of a single application as well as the total of the pair of applications changes as getting more or less local memory space. Take the particular case on Fig. 4.10 (a). The amount of local memory for Stream benchmark increases along the x axis, so its IPC value grows. On the contrary, local memory space is decreasing for the FFT benchmark (and consequently increasing its remote memory space), so its performance degrades. The higher point of the Total IPC line gives the optimum scheduling for the given pair of applications. In this particular case, the optimum configuration corresponds to all the local memory modules for FFT and all the remote ones for Stream; that is, Stream (0,0,16)- FFT(8,8,0).

row	Stream	FFT	Processor cycles(M)		Instructions		IPC		
0	(L, Lb, R)	(L, Lb, R)	Stream	FFT	Stream	fft	Stream	fft	Total
1	(0,0,16)	(8,8,0)	386,97	80,04	10268699	70441678	0,027	**0,880**	**0,907**
2	(0,4,12)	(4,8,4)	385,37	230,03	10266577	70686756	0,027	0,307	0,334
3	(0,8,8)	(0,8,8)	339,01	367,32	10201344	70896978	0,030	0,193	0,223
4	(2,2,12)	(6,6,4)	385,35	229,07	10266786	70685621	0,027	0,309	0,335
5	(2,6,8)	(2,6,8)	338,88	366,1	10201292	70894993	0,030	0,194	0,224
6	(4,0,12)	(8,4,4)	385,36	228,53	10266707	70684820	0,027	0,309	0,336
7	(4,4,8)	(4,4,8)	338,71	365,16	10201328	70893340	0,030	0,194	0,224
8	(4,8,4)	(0,4,12)	248,27	497,93	10056349	71119297	0,041	0,143	0,183
9	(6,2,8)	(6,2,8)	338,53	364,12	10200322	70892010	0,030	0,195	0,225
10	(6,6,4)	(2,2,12)	247,92	497,08	10055443	71117929	0,041	0,143	0,184
11	(8,8,0)	(0,0,16)	21,96	615,49	9707511	71291539	**0,458**	0,116	0,573
12	(8,0,8)	(8,0,8)	338,29	363,23	10200453	70890301	0,029	0,196	0,225
13	(8,4,4)	(4,0,12)	247,66	496,02	10055403	71115887	0,041	0,143	0,184

Figure 4.4: IPC and memory modules distribution for the Stream and FFT benchmarks.

row	Stream	Radix	Processor cycles(M)		Instructions		IPC		
0	(L, Lb, R)	(L, Lb, R)	Stream	Radix	Stream	Radix	Stream	radix	Total
1	(0,0,16)	(8,8,0)	386,97	160,53	10268699	157395418	0,027	**0,980**	**1,007**
2	(0,4,12)	(4,8,4)	385,37	252,93	10266577	157538420	0,027	0,623	0,650
3	(0,8,8)	(0,8,8)	339,01	343,39	10201344	157676599	0,030	0,459	0,489
4	(2,2,12)	(6,6,4)	385,35	252,52	10266786	157542204	0,027	0,624	0,651
5	(2,6,8)	(2,6,8)	338,88	342,93	10201292	157676595	0,030	0,460	0,490
6	(4,0,12)	(8,4,4)	385,36	252,3	10266707	157542891	0,027	0,624	0,651
7	(4,4,8)	(4,4,8)	338,71	342,47	10201328	157673143	0,030	0,460	0,491
8	(4,8,4)	(0,4,12)	248,27	432,88	10056349	157802620	0,041	0,365	0,405
9	(6,2,8)	(6,2,8)	338,53	342,09	10200322	157675530	0,030	0,461	0,491
10	(6,6,4)	(2,2,12)	247,92	432,44	10055443	157803733	0,041	0,365	0,405
11	(8,8,0)	(0,0,16)	21,96	522,44	9707511	157932514	**0,458**	0,302	0,760
12	(8,0,8)	(8,0,8)	338,29	341,65	10200453	157674620	0,029	0,462	0,491
13	(8,4,4)	(4,0,12)	247,66	432,02	10055403	157803405	0,041	0,365	0,406

Figure 4.5: IPC and memory modules distribution for the Stream and Radix benchmarks.

row	Stream	Chol.	Processor cycles(M)		Instructions		IPC		
0	(L, Lb, R)	(L, Lb, R)	Stream	Chol.	Stream	Chol.	Stream	chol.	Total
1	(0,0,16)	(8,8,0)	386,97	382,69	10268699	375547185	0,027	**0,981**	**1,008**
2	(0,4,12)	(4,8,4)	385,37	480,75	10266577	375688546	0,027	0,781	0,808
3	(0,8,8)	(0,8,8)	339,01	563,54	10201344	375809534	0,030	0,667	0,697
4	(2,2,12)	(6,6,4)	385,35	480,03	10266786	375686108	0,027	0,783	0,809
5	(2,6,8)	(2,6,8)	338,88	563,05	10201292	375807216	0,030	0,667	0,698
6	(4,0,12)	(8,4,4)	385,36	479,81	10266707	375686653	0,027	0,783	0,810
7	(4,4,8)	(4,4,8)	338,71	562,38	10201328	375805055	0,030	0,668	0,698
8	(4,8,4)	(0,4,12)	248,27	640,82	10056349	375940778	0,041	0,587	0,627
9	(6,2,8)	(6,2,8)	338,53	561,58	10200322	375803477	0,030	0,669	0,699
10	(6,6,4)	(2,2,12)	247,92	640,25	10055443	375939579	0,041	0,587	0,628
11	(8,8,0)	(0,0,16)	21,96	713,01	9707511	376055248	**0,458**	0,527	0,985
12	(8,0,8)	(8,0,8)	338,29	561,08	10200453	375803867	0,029	0,670	0,699
13	(8,4,4)	(4,0,12)	247,66	639,7	10055403	375938805	0,041	0,587	0,629

Figure 4.6: IPC and memory modules distribution for the Stream and Cholesky benchmarks.

row	Radix	FFT	Processor cycles(M)		Instructions		IPC		
0	(L, Lb, R)	(L, Lb, R)	Radix	FFT	Radix	FFT	Radix	FFT	Total
1	(0,0,16)	(8,8,0)	522,44	80,04	157932514	70441678	0,302	**0,880**	**1,182**
2	(0,4,12)	(4,8,4)	432,88	230,03	157802620	70686756	0,365	0,307	0,672
3	(0,8,8)	(0,8,8)	343,39	367,32	157676599	70896978	0,459	0,193	0,652
4	(2,2,12)	(6,6,4)	432,44	229,07	157803733	70685621	0,365	0,309	0,673
5	(2,6,8)	(2,6,8)	342,93	366,1	157676595	70894993	0,460	0,194	0,653
6	(4,0,12)	(8,4,4)	432,02	228,53	157803405	70684820	0,365	0,309	0,675
7	(4,4,8)	(4,4,8)	342,47	365,16	157673143	70893340	0,460	0,194	0,655
8	(4,8,4)	(0,4,12)	252,93	497,93	157538420	71119297	0,623	0,143	0,766
9	(6,2,8)	(6,2,8)	342,09	364,12	157675530	70892010	0,461	0,195	0,656
10	(6,6,4)	(2,2,12)	252,52	497,08	157542204	71117929	0,624	0,143	0,767
11	(8,8,0)	(0,0,16)	160,53	615,49	157395418	71291539	**0,981**	0,116	1,097
12	(8,0,8)	(8,0,8)	341,65	363,23	157674620	70890301	0,461	0,196	0,657
13	(8,4,4)	(4,0,12)	252,3	496,02	157542891	71115887	0,625	0,143	0,768

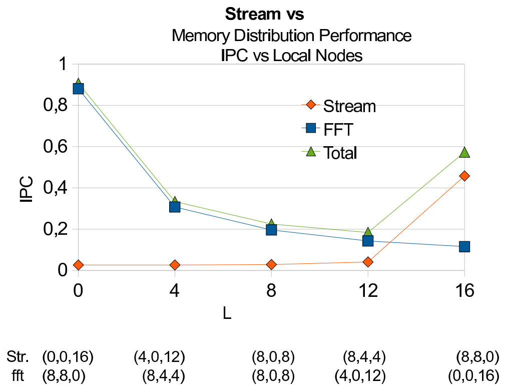
Figure 4.7: IPC and memory modules distribution for the Radix and FFT benchmarks.

row	Chol.	FFT	Processor cycles(M)		Instructions		IPC		
0	(L, Lb, R)	(L, Lb, R)	Chol.	FFT	Chol.	FFT	Chol.	FFT	Total
1	(0,0,16)	(8,8,0)	713,01	80,04	376055248	70441678	0,527	**0,880**	**1,407**
2	(0,4,12)	(4,8,4)	640,82	230,03	375940778	70686756	0,587	0,307	0,894
3	(0,8,8)	(0,8,8)	563,54	367,32	375809534	70896978	0,667	0,193	0,860
4	(2,2,12)	(6,6,4)	640,25	229,07	375939579	70685621	0,587	0,309	0,896
5	(2,6,8)	(2,6,8)	563,05	366,1	375807216	70894993	0,667	0,194	0,861
6	(4,0,12)	(8,4,4)	639,7	228,53	375938805	70684820	0,588	0,309	0,897
7	(4,4,8)	(4,4,8)	562,38	365,16	375805055	70893340	0,668	0,194	0,862
8	(4,8,4)	(0,4,12)	480,75	497,93	375688546	71119297	0,781	0,143	0,924
9	(6,2,8)	(6,2,8)	561,58	364,12	375803477	70892010	0,669	0,195	0,864
10	(6,6,4)	(2,2,12)	480,03	497,08	375686108	71117929	0,783	0,143	0,926
11	(8,8,0)	(0,0,16)	382,69	615,49	375547185	71291539	**0,982**	0,116	1,097
12	(8,0,8)	(8,0,8)	561,08	363,23	375803867	70890301	0,669	0,196	0,866
13	(8,4,4)	(4,0,12)	479,81	496,02	375686653	71115887	0,783	0,143	0,926

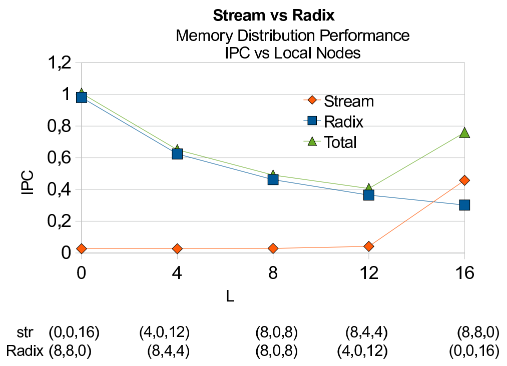
Figure 4.8: IPC and memory modules distribution for the Cholesky and FFT benchmarks.

row	Radix	Chol.	Processor cycles(M)		Instructions		IPC		
0	(L, Lb, R)	(L, Lb, R)	Radix	Chol.	Radix	Chol.	Radix	Chol.	Total
1	(0,0,16)	(8,8,0)	522,44	382,69	157932514	375547185	0,302	**0,981**	1,284
2	(0,4,12)	(4,8,4)	432,88	480,75	157802620	375688546	0,365	0,781	1,146
3	(0,8,8)	(0,8,8)	343,39	563,54	157676599	375809534	0,459	0,667	1,126
4	(2,2,12)	(6,6,4)	432,44	640,25	157803733	375686108	0,365	0,587	0,952
5	(2,6,8)	(2,6,8)	342,93	563,05	157676595	375807216	0,460	0,667	1,127
6	(4,0,12)	(8,4,4)	432,02	479,81	157803405	375686653	0,365	0,783	1,148
7	(4,4,8)	(4,4,8)	342,47	562,38	157673143	375805055	0,460	0,668	1,129
8	(4,8,4)	(0,4,12)	252,93	640,82	157538420	375940778	0,623	0,587	1,210
9	(6,2,8)	(6,2,8)	342,09	561,58	157675530	375803477	0,461	0,669	1,130
10	(6,6,4)	(2,2,12)	252,52	640,25	157542204	375939579	0,624	0,587	1,211
11	(8,8,0)	(0,0,16)	160,53	713,01	157395418	376055248	**0,981**	0,527	**1,509**
12	(8,0,8)	(8,0,8)	341,65	561,08	157674620	375803867	0,461	0,670	1,131
13	(8,4,4)	(4,0,12)	252,3	639,7	157542891	375938805	0,625	0,587	1,212

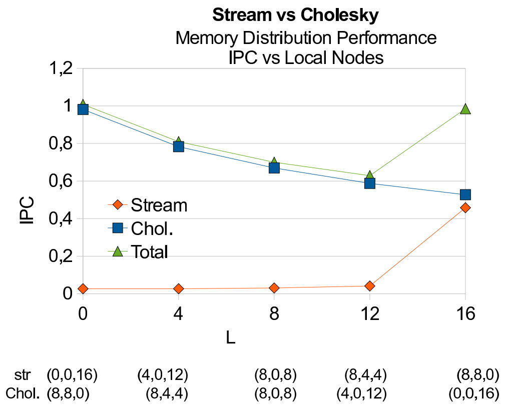
Figure 4.9: IPC and memory modules distribution for the Radix and Cholesky benchmarks.



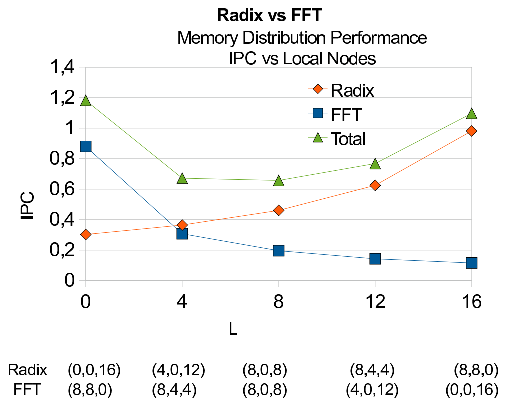
(a) Stream vs FFT



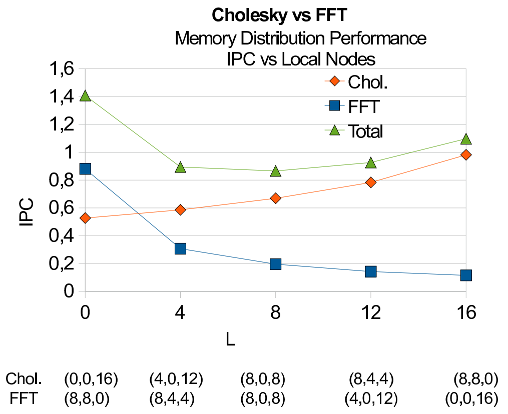
(b) Stream vs Radix



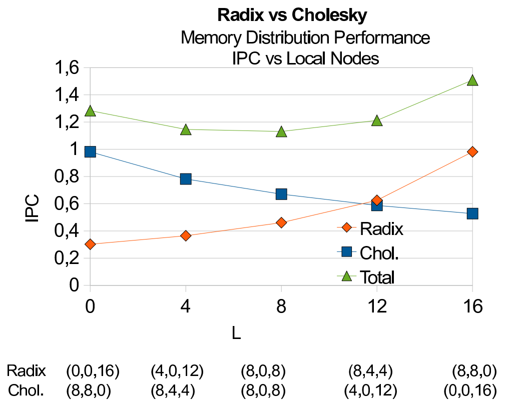
(c) Stream vs Cholesky



(d) Radix vs FFT



(e) Cholesky vs FFT



(f) Radix vs Cholesky

Figure 4.10: IPC and memory scheduling diagrams.

Chapter 5

Conclusions and Future Work

The analysis of the influence of scheduling memory on the performance of applications concurrently running in computing platforms has been presented. This work takes part of a larger project. To carry out this work, some previous steps must be performed.

In this context, an extension of hypertransport protocol in a Supermicro scientific parallel cluster, to support memory scheduling among different servers, has been proposed in the group and assumed in this work to perform. Also, a remote memory controller that allows the management of the memory of all the motherboards of the system linked with hypertransport, not only the onboard memory but the abroad memory as well, has been designed in the overall project.

A static assignment of memory for scientific parallel applications has been set up.

The evaluation of these distributions has shown that the performances of the applications depends on the memory assignment heuristic.

It is remarkable the fact that the information (i. e., Instructions Per Cycle) to apply the heuristics can be easily implemented in simple hardware. For instance, some hardware counters to keep track of executed instructions and cycles. Then, this information could be read by the Operating System.

As for future work heuristics will be applied dynamically depending on the memory requirements of the applications. Different heuristics will be explored because they have different effect on the performance. The disk swap will be considered since large computing systems are composed of huge disks, in this case results are expected to be improved. A parameter estimating the Quality Of Service will be included to avoid excesive performance degradaton in a given application. Furthermore, a non-uniform criterion of distributing memory modules will be applied (i. e., not limiting each application to a half of the available memory, and permiting different proportions). Finally, it would be also interesting to test more workloads with different memory requirements concurrently running.

Bibliography

[1] IBM journal of Research and Development staff. Overview of the IBM Blue Gene/P project. . *IBM J. Res. Dev.*, 52(1/2):(199–220, 2008.

[2] http://www.numachip.com/.

[3] B. Nitzberg and V. Lo-University of Oregon. Distributed shared memory: concepts and systems, 1998. *IEEE Computer Society.*

[4] David Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *IEEE Computer*, 38(5):11–13, 2005.

[5] Jim Gray, David T. Liu, María A. Nieto-Santisteban, Alexander S. Szalay, David J. DeWitt, and Gerd Heber. Scientific Data Management in the Coming Decade. *CoRR*, abs/cs/0502008, 2005.

[6] Gaussian 03. http://www.gaussian.com.

[7] IBM z Series. http://www.ibm.com/systems/z.

[8] HP Integrity Superdome. http://h20341.www2.hp.com/integrity/cache/342254-0-0-0-121.html.

[9] HyperTransport Technology Consortium. HyperTransport I/O Link Specification Revision 3.10, 2008. .

[10] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, 2003.

[11] J. Duato, F. Silla, and S. Yalamanchili. Extending HyperTransport Protocol for Improved Scalability. *First International Workshop on HyperTransport Research and Applications*, 2009.

[12] H. Litz, H. Fröening, M. Nuessle, and U. Brüening. A HyperTransport Network Interface Controller for Ultra-low Latency Message Transfers. *HyperTransport Consortium White Paper*, 2007.

[13] John Oleszkiewicz, Li Xiao, and Yunhao Liu. Parallel Network RAM: Effectively Utilizing Global Cluster Memory for Large Data-Intensive Parallel Programs. In *ICPP*, pages 353–360. IEEE Computer Society, 2004.

[14] Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *CLUSTER*, pages 1–10. IEEE, 2005.

[15] Masato Oguchi and Masaru Kitsuregawa. Using Available Remote Memory Dynamically for Parallel Data Mining Application on ATM-Connected PC Cluster. In *IPDPS*, pages 411–420. IEEE Computer Society, 2000.

[16] Paul Werstein, Xiangfei Jia, and Zhiyi Huang. A Remote Memory Swapping System for Cluster Computers. In David S. Munro, Hong Shen, Quan Z. Sheng, Henry Detmold, Katrina E. Falkner, Cruz Izu, Paul D. Coddington, Bradley Alexander, and Si-Qing Zheng, editors, *PDCAT*, pages 75–81. IEEE Computer Society, 2007.

[17] Masato Oguchi and Masaru Kitsuregawa. Dynamic remote memory acquisition for parallel data mining on ATM-connected PC cluster. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 246–252, New York, NY, USA, 1999. ACM.

[18] H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato. DLM: A distributed Large Memory System using remote memory swapping over cluster nodes. In *Cluster Computing, 2008 IEEE International Conference on*, pages 268–273, 29 2008-Oct. 1 2008.

[19] GEMS Webpage. http://www.cs.wisc.edu/gems/.

[20] Changkyu Kim, Doug Burger, and Stephen W. Keckler. NUCA: A Non-Uniform Cache Access Architecture for Wire-Delay Dominated On-Chip Caches. *IEEE Micro 2004*.

[21] Stream Benchmark. http://www.cs.virginia.edu.

[22] Splash2 benchmark. http://www-flash.stanford.edu/apps/splash/.

[23] Splash2 for Linux. http://kbarr.net/splash2.

[24] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. pages 24–36, 1995.

[25] David H. Bailey. FFTs in External or Hierarchical Memory. In *PPSC*, pages 211–224, 1989.

[26] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. *Commun. ACM*, 39(12es):273–297, 1996.