Active learning processes to study memory hierarchy on Multicore systems

John Corredor, Juan Carlos Moure, Dolores Rexachs, Daniel Franco, Emilio Luque

Universitat Autònoma de Barcelona
Computer Architecture and Operating System Department
Barcelona, SPAIN
john.corredor@caos.uah.es, juancarlos.moure, dolores.rexachs, danielfranco, emilio.luque@uah.es

Abstract
A current challenge for computer users is to fully exploit performance of new Multicore systems. We propose a methodology for students in computational science to analyze the effect of memory hierarchy on application performance. The analysis is proposed in a experimental environment consisting of different systems with different configurations of the memory hierarchy. The new Multicore systems put tremendous pressure on memory hierarchy systems. The pressure is because, unfortunately, the effectiveness of the computing power offered by Multicore, is affected by the data communications inter-chip and off-chip to the memory hierarchy, leading to significant problems in performance for many parallel applications. In the scope of computer science, it is important that students understand these problems. This methodology was successfully applied to students, where they acquired a significant improvement in their parallel application metrics assessment as was demonstrated in our evaluation.

Keywords: Active learning, hierarchy memory, performance counters.

1. Introduction

Computational Science and Engineering (CSE) is the multi-disciplinary field of computer-based modeling and simulation for studying scientific phenomena and engineering designs. The masters program in CSE is geared towards accepting students of various backgrounds across various sectors. This includes students from different areas that make use of mathematical modeling and scientific computing technologies.

In Figure 1 the structure of the masters in CSE at the University Autonoma of Barcelona [2] is shown. The master has an optional module on High Performance Computers (HPC). This module is oriented to study the influence of the hardware on HPC application performance.

---

1This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974.
The scientific, engineering, and industry research community has long driven the need for high-performance computing. Recent Multicore systems provide an efficient and economical way to solve large-scale and/or time-constrained problems. The use of systems with Multicore architectures have become widespread, as they provide higher performance at lower cost and make more efficient use of energy [1].

A single-core, with the development of complexity to take advantage of instruction level parallelism (ILP). The Multicore chip in turn, takes advantage of the threads level parallelism (TLP), available in applications. Combined with increased clock frequency, a Multicore, multi-threaded processor chip demands higher on- and off-chip memory bandwidth and suffers longer average memory access delays despite an increasing on-chip cache size. Tremendous pressures are put on memory hierarchy systems to supply the needed instructions and data in a timely fashion.

The pressure is because, unfortunately, the effectiveness of the computing power offered by Multicore chips, is affected by the data communication inter-chip and off-chip to the memory hierarchy, leading to significant problems in performance for many parallel applications. In the scope of computer science, it is important that students understand these problems. Increasing the number of threads regardless of the memory hierarchy, can cause decreased performance in some applications.

Taking advantage of the problems presented by students when they are designing parallel applications on Multicore systems, we propose a methodology for students to analyze the effect of memory hierarchy on application performance. The analysis is proposed in an experimental environment consisting of different systems with different configurations of the memory hierarchy. This HPC module is recommended to all students without a computer architecture background.

Our methodology develops strategies which allow students to design an active learning process which integrates them into the teaching of efficient parallel programming on HPC labs. This learning process enables students constant class participation, in which efficient parallel application issues are discussed. Also, students can learn from their own mistakes. Our teaching method is designed to change the traditional lecture method applied. We mainly take into consideration the student interactions and their parallel designed contributions.

To apply the methodology, the students must have studied the concepts and themes: the basic equation of performance, calculating the CPI from the frequencies of each type of instruction, basic optimizations performed by the compiler, the operation of cache memory and virtual memory. Before attending the laboratory session, they should carefully read the experiments performed and described in the documents of practice, which contains a detailed description of applications to analyze and objectives to be achieved as described in section 5.
This paper is structured as follows: A description of the methodology is presented in section 2. An overview about memory hierarchy in section 3, here we introduces the nomenclature we will use in the remainder of the document. Section 4 describes the experimental environments and teaching laboratory procedures. Section 5 details used in order to achieve the analysis of the influence of memory hierarchy on the performance of applications, and finally conclusions are given in section 6.

2. Analysis of the influence of memory hierarchy

The proposed methodology for studying the influence of the memory hierarchy on the application performance is based on experimentation and analysis of results. Figure 2 shows the methodology.

![Methodology Flowchart](image)

Figure 2: Methodology to analyze the Memory Hierarchy influence on Application Performance

The student must know the experimental environment to attend the practice, the teacher provided the information about the different computer systems and the necessary tools and interfaces. To facilitate the use of these tools, we have implemented interfaces and given a full description of how to use them.

The teacher gives the applications to implement and instrument in the laboratory sessions. The application is instrumented and executed to measure time, misses and accesses of iL1, dL1 and L2.

Students have a questionnaire in which to interpret these results and summarize their learning or knowledge of the impact of memory hierarchy with regard to changes in algorithm.

3. Memory Hierarchy Architecture

The concept of memory hierarchy is aimed at imitating a flat, uniformly accessible, large memory, by taking advantage of locality of reference that most programs exhibit. Since fast memory is expensive, a memory hierarchy is organized in levels - each smaller, faster and more expensive than the level below it. The goal is to provide a memory system which costs almost as little as the lowest and cheapest level of the hierarchy and is as fast as the highest level of the hierarchy.

Following is a useful and simplified theoretical description of the components that we will consider in this methodology to study the influence of memory hierarchy, in the performance for an application.
3.1. Cache

The level 1 (L1) cache is the smallest, fastest and most expensive memory. It is usually co-located with the processor to minimize its access time. If the CPU needs a load the caches are searched first. Obviously, the cache cannot contain the content of the entire main memory (otherwise we would need no memory), but since all memory addresses are cacheable, each cache entry is tagged using the address of the data word in the main memory. In this way, a request to read or write to a memory address can search the caches for a matching tag. The address in this context can be either the virtual or physical address, varying based on the cache implementation [3].

Figure 3 a) shows the levels of cache and introduces the nomenclature: L1d is the level 1 data cache, L1i the level 1 instruction cache and L2 is the level 2 cache. In turn, nomenclature for the misses: L1i miss rate, L1d miss rate and L2 miss rate.

A simplified diagram (Figure 3 b and c), shows aspects that differ in the configurations of the memory hierarchy between the Single-core and multiple cores or Multicores. There are several configurations of the memory hierarchy in the current Multicore systems, i.e. there is no standard configuration. It is therefore important to analyze how different configuration affect the performance obtained when running applications. For example, architectures such as Quad-core AMD Barcelona processor have separate level 1 and level 2 data caches per core and a shared level 3 cache; and other example is the Quad-core Intel Clovertown processor, which exhibits a separate level 1 and shared level 2 cache per each two cores.

Some on-chip configurations share L2 cache architectures in multi-core processors (Figure 3 c). Shared caches have important advantages such as increased cache space utilization, fast inter-core communication (via the high-speed shared L2 cache), and reduced aggregate cache footprint through the elimination of undesired replication of cache lines [4].

A major disadvantage of shared L2 caches, however, is that uncontrolled contention can occur by allowing CPU cores to freely access the entire L2. As a result, scenarios can occur where one core constantly evicts useful L2 cache content belonging to another core without obtaining a significant improvement itself. Such contention causes increased L2 cache misses which in turn leads to decreased application performance. For example, a low priority application running on one core that rapidly streams through the L2 cache can consume the entire L2 cache and remove most of the working set of higher priority applications co-scheduled on another core.
3.2. Main Memory

In the memory hierarchy, main memory is the next level down the hierarchy, below the cache levels. There are two common measures of the main memory performance, namely memory latency and memory bandwidth. Memory latency is the time duration which starts when a read/write operation is issued and ends when the desired data arrives. Memory bandwidth is the maximum of data that may be read/write from/to the memory in one unit of time.

4. Experimental environment

In this section we describe the experimental environment and also the methodology for the development of practices or experiments. The student must understand the tools and systems available in order to receive all benefits of the development practice.

4.1. Computer Systems

In the developed practices, 2 different systems have been used. Both systems are ISA architecture compatible but with different memory hierarchies, a Single-core and a Multicore system. This could be extended to use different Multicore systems with different cache structures.

**Single-core** A computer system Intel(R) Pentium(R) 4; CPU 2.60GHz, with 768MB of main memory. The system has only one core or single core with 512KB L2 cache (Linux OS i686) See Figure 3 b).

**Multicore** A computer system with 2 x Dual-Core Intel(R) Xeon(R) 5160; CPU 2988GHz, with 12GB of main memory. The system has 2 chips, and each chip has 2 cores. For each chip share 4MB L2 cache between the cores. The system offers the ability to perform 4 logical threads on the operating system (Linux OS x86_64), See Figure 3 c).

4.2. Tools for performance measures

Current processors have hardware performance counters which provide different metrics to measure performance (clock cycles consumed, executed instructions of different types, etc). These performance counters can be saved to memory and recalled from memory whenever there is a change of context by the OS and empties the current process in execution. Thus, it can have precise information (with a margin of error) for each separate process that is running, even when multiple processes share the same processor.

The access to the HW performance counters requires the use of low-level instructions, which in many cases has a high user privilege, therefore they are accessible just by system processes. To overcome this issue, we use the Performance Application Programmer’s Interface (PAPI) in the experiments, a library of functions that are accessible by user level code [6].

The commands `papi_avail` and `papi_mem_info` (executables from the command prompt) give information about processor speed and performance events that can be measured. The events (system performance) that will be measured, are related to the performance of the memory hierarchy. In Table 1, the command `papi_mem_info` information is shown, allowing us to observe in detail, characteristics of the cache memory (L1i, L1d and L2 caches). Table 2, demonstrates the command `papi_avail`; this command shown the “commands” and “events” that allow us to take the measurements (for the systems described previously).

As the functions provided by PAPI are not simple to use, especially if you want to take measures in multi-core processors or systems with multiple processors (Symmetric Multi-Processing...
SMP), to help students in this performance evaluation task we have developed specific functions to use PAPI. These functions are used to monitor the programs in experiments. Students need to understand how these functions works and how to use them, but not their implementation in-depth.

<table>
<thead>
<tr>
<th>Vendor and model</th>
<th>Intel Core i7 970 (16)</th>
<th>Intel Xeon E5540 (16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Revision</td>
<td>11.00000000</td>
<td>9.00000000</td>
</tr>
<tr>
<td>CPU Megahertz</td>
<td>1992.000000</td>
<td>2589.659912</td>
</tr>
<tr>
<td>Total CPUs</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>PAPI events</td>
<td>Available</td>
<td>Available</td>
</tr>
<tr>
<td>Instructions completed</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Level 1 instruction cache hits</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Level 1 data cache misses</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Level 1 total cache accesses</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Level 1 instruction cache (L1) accesses</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Level 2 total cache (L2) accesses</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Level 2 cache (L2) misses</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Level 2 data cache (L2) misses</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2: Command `papi_avail` on computer system Multicore and Single-core

4.2.1. Interface for performance tools

The code of a sample program, "main.c", using the above defined functions is shown in Figure 4. The program is composed of 4 instrumentation functions: Sample_Init, Sample_Start, Sample_Stop and Sample_End to measure performance details of the function func(), which is shown later.

When the program instrumented with PAPI is run, one PAPI performance event must be
passed as parameter, to select among a number of events available in the particular system which runs (command `papi_avail`). The result of the call to the function `Sample_End` on the program is the HW counter value specified in the command line.

5. Instrumentation, Measurements and Analysis

In the following sub-sections the different applications used in the laboratory will be described. These applications were selected because they include some of the standard operations in scientific computing, their scalability and the inclusion of typical memory access patterns. Table 3 gives a brief description of these applications. In 5.1.2 and 5.2.1 the analysis is shown for Single-core and Multicore systems.

<table>
<thead>
<tr>
<th>Application</th>
<th>Objectives</th>
<th>Results</th>
<th>Proposed tasks and questions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addvec.c</code></td>
<td>sum all the elements (SZ) of a vector of double</td>
<td>analyze the effects in L1, L2.1 and L2.2 varying SZ.</td>
<td>To explain the results of workload on metric. Example question: What happens to every metric by increasing SZ?</td>
</tr>
<tr>
<td><code>mult.c</code></td>
<td>classical matrix multiplication with (SZ)^2 elements stored by rows</td>
<td>X=1000, analyze the effects in L1, L2.1, L2.2. Compare the three versions and explain the differences</td>
<td>Compared the executions Single-core and Multicore Example task: Compare the three versions and explain the differences on both systems</td>
</tr>
<tr>
<td><code>mult.c</code></td>
<td>assumes matrix b interchanging the rows and columns of the matrix (interchanging rows as columns and vice versa)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mult2.c</code></td>
<td>performs transformation called register tiling, exploits temporal locality of the algorithm using local variables to make better use of available registers and reduce total number of memory accesses.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Brief description of the applications and objectives to be achieved

5.1. Example application 1

The code in Figure 5 implements the sum of all the elements of a vector of double precision floating point numbers (SZ is the variable to identify the value of the vector size). SZ value can be modified at compile-time and is constant at runtime. The code is implemented in OpenMP[5] in order to obtain the shared memory parallel program in the Multicore system. The student must analyze the function and explain the performance measures.

```c
static double a[SZ], c;

void func() {           // Function to instrument with PAPI
    int i;

    #pragma omp for reduction(+:c)  // Pragma OpenMP, parallelizing shared memory
    for (i=0; i<SZ; i++)
        c += a[i];
}

void init() {         // init data and warm-up cache (Not instrumented)
    int i;
    for (i=0; i<SZ; i++) {
        a[i] = 2.374*i + 4.2;
    }
}
```

Figure 5: Program perform a sum all the elements of a vector a (`addvec.c`)

Analyzing the code, you can determine that the function `func()` executes the inner loop `SZ` times.
Students should compile the application whose main program "main.c" has been described earlier in this section, including PAPI instrumentation for different SZ values (SZ increasing \(x10\)) each measure, from 10,000 to 1,000,000, and run several times, using the different events of the command papi_avail (see table 2).

Now, the students must execute the application for each PAPI event (at least 30 times the same execution and PAPI event) in order to obtain average values (statistical validity), and then vary the SZ. In Table 4 a summary of averages is shown.

5.1.1. Average of measured values

The students must obtain the miss rate of instruction and data L1 caches, miss rate of the L2 cache and the runtime for both systems. The miss rate of L1 and dL1, is obtained with the instructions and data values from the event PAPI_L1.ICM and PAPI_L1.DCM divide by PAPI_L1.ICA and PAPI_L1.DCA, respectively. The L2 miss rate is obtained with the L2 misses values from the PAPI_L2.DCM event divide by PAPI_L1.DCA event. The following table (see table 4) resumes values in average.

<table>
<thead>
<tr>
<th>SZ</th>
<th>10^2</th>
<th>10^3</th>
<th>10^4</th>
<th>10^5</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 miss rate</td>
<td>0.09%</td>
<td>0.13%</td>
<td>0.009%</td>
<td>0.018%</td>
</tr>
<tr>
<td>dL1 miss rate</td>
<td>4.71%</td>
<td>10.45%</td>
<td>12.56%</td>
<td>12.57%</td>
</tr>
<tr>
<td>L2 miss rate</td>
<td>0.00%</td>
<td>0.72%</td>
<td>3.21%</td>
<td>1.73%</td>
</tr>
<tr>
<td>Runtime (ns)</td>
<td>16.32</td>
<td>3.46</td>
<td>324.67</td>
<td>27.07</td>
</tr>
</tbody>
</table>

Table 4: Average of measured values.

5.1.2. Analysis

The analysis below correspond to the evaluation of the results of the Table 4. Following the methodology, the teacher can ask questions such as: What happens each metric by increasing SZ? Now the student must explain the results.

**L1 miss rate** starts out very small and it becomes almost 0. The explanation is that are cold misses (or compulsory misses) caused by the first reference to a datum, which is very small. The larger the total number of instructions executed, the less effect the cold misses have on the program runtime. This happens in both systems.

**dL1 miss rate** starts small and ends up being important, 12.5% for the two systems.

**L2 miss rate** starts out as zero, and upon reaching \(SZ = 10^3\) in Single-core is significant (3.2%) to grow to 4.31%. In Multicore when \(SZ = 10^4\), increases substantially to 84%, because the level 2 cache is shared for each 2-threads, then the cores are in conflict for the resource, i.e. the 4 threads limit the memory bandwidth.

**Runtime** - Single-Core: we can see where the increases \(SZ\), is not sustained the increase in runtime. This is due because the first part (\(SZ\) \(10^3\)) is in the transition state and then become stable at runtime as a function of the workload. - Multicore: increasing \(SZ\), there is a almost steady increase in the runtime.

5.2. Example application 2

The following algorithms mult, multt and mult2 perform matrix multiplication with double elements, each with a SZxSZ size array. These different versions, trying to optimize aspects of
locality to improve access times to data. Each version is an enhancement to the previous one. The codes are implemented in OpenMP in order to obtain the paralleling program shared memory in the Multicore system.

**mult** classical matrix multiplication with $(SZ)^2$ elements stored by rows.

**multt** assumes matrix $b$ interchanging the rows and columns of the matrix. (interpreting rows as columns and vice versa).

**mult2** performs transformation called *register tiling*, exploits temporal locality of the algorithm using local variables to make better use of available registers and reduces total number of memory accesses.

Student must execute application 2 (matrix multiplication program: three versions) several times to fill in Table 4, shown below, with values of performance measures. They must also generate the assembler file for each version, and view what instructions are part of the main loop function.

For Multicore system, each version of events is measured separately for each thread, then the average maximum for all threads is calculated. The following table (see Table 4) gives this average values.

<table>
<thead>
<tr>
<th>SZ=1000</th>
<th>Instructions x10^6</th>
<th>L1 Accesses x10^6</th>
<th>L1 misses x10^6</th>
<th>L2 misses x10^6</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Singlecore</td>
<td>Multicore</td>
<td>Singlecore</td>
<td>Multicore</td>
<td>Singlecore</td>
</tr>
<tr>
<td>mult</td>
<td>8014</td>
<td>1,752</td>
<td>2000</td>
<td>812</td>
<td>1,255</td>
</tr>
<tr>
<td>multt</td>
<td>8013</td>
<td>1,502</td>
<td>2000</td>
<td>526</td>
<td>198</td>
</tr>
<tr>
<td>mult2</td>
<td>6005</td>
<td>907</td>
<td>1876</td>
<td>255</td>
<td>124</td>
</tr>
</tbody>
</table>

Table 5: Average of measured values.

5.2.1. Analysis

The analysis below corresponds to the results of Table 5. An example task for the analysis: comparing the three versions and explain the differences on the systems.

Improvement runtime between **multt** and **mult**:

- Single-core $14,23/5,76 = 2.47$ times. We can see the numbers of instruction and L1 accesses is almost the same. The L1 misses is around $6.3$ times less, but the cause is the reduction in cache L2 misses, that is around $15.75$ times less.

- Multicore is $2.35/0.98=2.39$ times. Instruction and access to L1 is equal like in Single-core. Now, the cause about improvement is L1 misses in $344/31=11.09$ times less.

The different between **multt** and **mult** is the order in which vector elements are accessed(change access pattern from sequential stride SZ=1000 to sequential stride 1). I.e., exploits spatial locality when accessing “vector b” (whereas it had any locality before).

Improvement runtime between **mult2** and **multt**: The improvement is given in both systems by reduction to half the misses in all data caches. The additional memory accesses (due to register spilling) have much temporal and spatial locality, because they occur on a very small process. The different between **mult2** and **multt** is that **mult2** rearranges the way to access data from $a$ and $b$, adding temporal locality.
6. Conclusion

This paper has presented a methodology to develop active learning processes for teaching efficient parallel programming to the HPC students in CSE. It is innovative because we offer students a way to analyze the effects of the memory hierarchy on application performance, in order to obtain an efficient parallel program in the new Multicore systems.

The objective of this methodology is to allow students, through the experiences they obtain during class, to develop efficient parallel applications, as has been demonstrated. Some positive results permit us to consider the effective application of our methodology in class.

Once the methodology was applied to student, we can obtain a significant improvement in students' abilities in parallel programming design. This improvement is due to the active learning process which is applied in our methodology, where students' considerations are very important to evaluate the success of this methodology. Our methodology is focused on allowing student to learn through their experiences obtained during classes, and these experiences are achieved through the issues presented in their parallel applications.

Each class is totally different and the students have different knowledge levels. For these reasons, we have to consider the heterogeneities in the students, and these heterogeneities have to be managed by instructors in order to obtain significant learning in the class. The improvement strategies have to be explained in two manners, one considering the needs of each student group, while the other strategies could be taught for all the classes. The success of these strategies mainly depends on the interaction between instructor and students.

This methodology was successfully applied to students, where they acquired a significant improvement in their parallel application metrics assessment as was demonstrated in our evaluation.

This methodology is used to teach the module of High Performance Computers in the master of CSE at the University Autonoma of Barcelona.

7. Reference