# Optimization and Parallelization Experiences Using Hardware Performance Counters

Fernando G. Tinetti[1],[2], Sergio M. Martin[3], Fernando E. Frati1, Mariano Méndez[1]
[1]*III-LIDI, Fac. de Informática, UNLP*
*Calle 50 y 120, 1900, La Plata, Argentina*
[2]*Comisión de Inv. Científicas de la Prov. de Bs. As.*
*3Universidad Nacional de La Matanza*
*Florencio Varela 1903 - San Justo, Argentina*
email: {fernando, fefrati}@info.unlp.edu.ar,
smartin@ing.unlam.edu.ar, marianomendez@gmail.com

## Abstract

*Current hardware for compute intensive tasks includes a large amount of processing facilities, which are sometimes difficult to use in an optimized way. High performance computing (HPC) is always focused in solving challenging (or, at least, compute intensive) problems for which the response time is the priority. We have been working from two different but usually complementary research problems: a) updating and parallelizing legacy (HPC/numerical) software, and b) analyzing different problems and approaches to optimization and parallel processing in clusters. We have found that raw hardware event counters do not always directly provide useful information. We also found some guidelines for evaluating performance using those counters in the context of optimization and parallelization. In this article, we present those guidelines along with the performance evaluation tools that we used to determine objectively what parts of the algorithm offered better chances of improvement.*

***Keywords: High Performance Computing, Source code optimization, Performance Evaluation, Parallel Computing, Cluster Computing.***

## 1. Introduction

High performance computing (HPC) in clusters is one of the most popular approaches for solving compute, or more specifically, numerical intensive workloads/ tasks. Computers currently used as cluster's nodes usually range from low end desktop/PC computers (which are economically cheap) to high end servers/PCs. Terms such as *low* end *desktop computer* and *server computer* have many definitions depending on hardware, hardware corporations marketing, and specific market status. We will refer to them just as PCs or cluster nodes, emphasizing on their proven advantageous features, which at least include low price and high availability [10] [13]. Furthermore, we will refer to clusters as shown

in Fig. 1: a local area network of computers made up of commodity hardware components. Cluster nodes, as shown in Fig. 1 are expected to be multi-core computers sharing memory, which is the current conceptual configuration even in NUMA (Non-Uniform Memory Access) hardware [4]. The most commonly used interconnection network is 1Gb/s Ethernet (availability/commodity hardware, cost), but almost any other interconnection network could be used, such as those specifically designed for HPC in clusters, such as Infiniband.

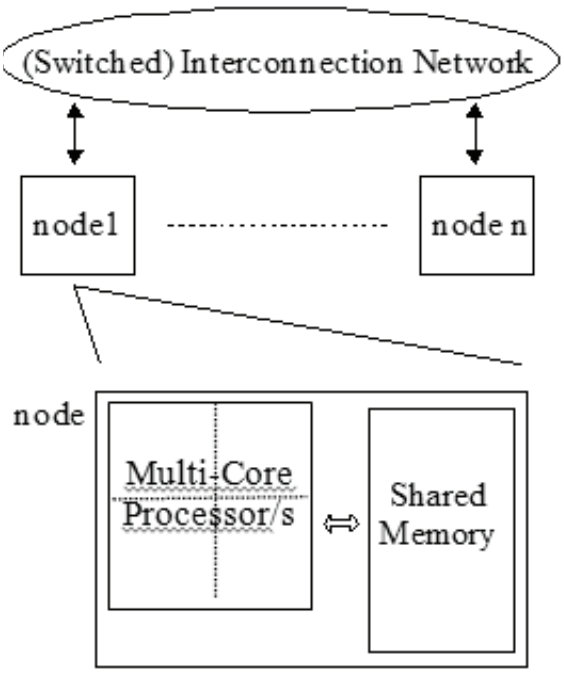


**Figure 1. Cluster used for HPC.**

There are two levels or types of parallel processing in current clusters: intra node and cluster-wide (inter-nodes). The most common programming models for parallel processing in clusters are the message passing and shared memory (threaded) ones. Message passing is usually implemented by using some (message passing library) implementation of MPI (Message Passing Interface) [9], such as OpenMPI. Shared memory parallel processing can be implemented in terms of OpenMP [10], which is currently implemented by most C and Fortran compilers. Also, it is possible to combine both programming models (message passing and shared memory) in the so called hybrid approach as in [12].

From the point of view of HPC applications, legacy software has a strong need to be updated to the new multi-core environment/s. By definition, every programmer has to deal with strong problems when facing (updating, etc.) legacy software, but HPC applications in particular have to be parallelized since the processors' clock rate is not going to increase beyond 3.8-4.2 GHz in the near future [14] [5] [16]. However, legacy software is not the only software which needs to be parallelized or even updated. New algorithms and new parallel platforms are always analyzed in order to model and optimize performance. Also, new applications and new applications sizes are taken into account as more cores are included in a multicore chip and more computers are available (or interconnected) for parallel computing.

New microprocessors also include new microarchitectures which not always are fully exploited for maximum performance. Moreover, new microprocessors often provide access to (internal) performance counters in order to analyze and optimize runtime performance [1] [6]. Interestingly, performance is associated to debug in [1], which includes a chapter named "Software Debug and Performance Resources" (Chapter 13). This *association* would also provide an idea of the related complexity.

This possibility of measuring hardware

counters as a way to determine the performance of the algorithms being executed motivated us to use profiling tools, such as Perf [19], and Perfsuite [8], to determine how our algorithms performed in such low-level. Before we did this research, we had to use a "guessing" and code analysis approach to determine which could be the causes of performance degradation, especially on multi-core architectures. Even though we could achieve some improvement, the exact causes of the initial problems remained undiscovered.

However, using tools to access and analyze hardware counters, allowed us to get a more objective idea of how to improve our algorithms. In this article, we will show how we used Perf and PerfSuite to determine which were the specific hardware counters that provided us with the clues as to how to increase their performance. As a result of using these tools, we could objectively determine which were the causes of several of their performance-degrading problems. We would expect other scientist programmers to apply these same tools on their algorithms, and analyze these same counters in search of analog opportunities to improve their performance.

The rest of this article is organized as follows. Section 2 describes the classical performance metrics and hardware performance event counters (which are also referred to as hardware performance monitoring event counters). The main concepts for performance evaluation and our initial work on a *real world* legacy program are explained in Section 3, with specific results we deduced hardware counters and experimentation. Section 4 introduces a complete example based on a well known problem and focused on parallelization, and the effect of some classical optimizations on parallel performance and the so called *memory* wall related to parallel computing on multicore multiprocessors. Finally, includes several conclusions and further work taking into account the work done and explained in this paper, mainly on legacy as well as parallel code.

## 2. Performance Metrics and Counters

Performance has been always the ultimate goal in the HPC field. Usually, sequential performance has been measured directly in terms of runtime or rates of instructions or floating point operations per second, such as MFlop/s (millions of floating point operations per second). Parallel performance has been measured also in terms of (plain) runtime or with more specific metrics such as Speedup and Efficiency (usually as functions of the number of processors), defined as in Eq. (1) and Eq. (2) respectively [16], where $p$ is the number of processors, $op\_st$ is the runtime of the optimum sequential algorithm, and $pt(p)$ is the parallel elapsed runtime using $p$ processors. It is expected (but not always possible) to obtain a $Speedup(p)$ value near p, since it means that every processor has been used for a 1/p fraction of the total processing to be done.

$$Speedup(p) = \frac{op\_st}{pt(p)} \quad (1)$$

It is worth noting that $0 < Efficiency(p) \leq 1, \forall p$ (at least in non pathological cases), and values of $Efficiency(p)$ near 1 indicates that about 100% of computing resources are used at runtime.

$$Efficiency(p) = \frac{Speedup(p)}{p} \quad (2)$$

We could argue that in the end, every performance metric is computed using elapsed runtime. On one hand, it is fair enough, since runtime is *independent* of the underlying processing hardware. But on the other hand, given a performance value, it is hard or unlikely to guess specific performance problems and/or penalties. There are some *classical* guidelines to look for performance penalties in the parallel processing area. Most of the parallel performance optimization ideas and specific algorithms try to solve several communication, synchronization, and/or computing (un)balance problems. At this point, hardware performance counters provide the most specific and accurate information of the hardware performance.

Having access to hardware performance counters tends to reduce the number of unknowns/guesses at least on specific parts of the available hardware. Thus, we can use performance (monitoring) counters for identifying (some) performance penalties and evaluate algorithms changes which are made for optimization/s [9] [2] [3] [7]. Unfortunately, hardware counters report very specific and low level information, which is not always directly/easily related to the algorithms. We will show that performance counters have to be carefully analyzed and, sometimes, specific experiments have to be carried out to collect relevant data. We will avoid using proprietary manufacturers' tools and hardware/model low level information whenever possible (e.g. Intel definition and usage of incore-uncore events). From this point of view, tools such as perf and API (Application Programming Interfaces) such as PAPI (Performance API) [18] provide more or less general and vendor independent information.

## 3. Legacy Code Example

We have selected a global climate model as an example of legacy code example: GISS-AOM(C4x3) from GISS, the NASA Goddard Institute for Space Studies [11]. We have experimented with this legacy Fortran code looking for several interesting information regarding compilers, performance counters, legacy code optimization, and prospective issues for parallelization:

- Legacy code approach/methodology: starting with profiling, and advancing to source code behavior and optimization. We consider the starting points as a completely unknown legacy code, so that experiments, profiling, and monitoring events provide a basis for a methodology on enhancing such legacy source code.
- Similarities/differences among compilers, mostly from the point of view of performance and optimization levels. We have used gfortran and ifort (Intel Fortran compiler). We are not interested in compiler-specific optimizations and language (Fortran) extensions, and this is why we have used "-O[1/2/3]" optimization levels.
- Optimization level impact on performance, based on monitoring events reported by the processor/s.

We have used PerfSuite as a high level approach for gathering hardware performance events information, i.e. to avoid doing a direct analysis of lower level tool results such as perf

[19] and instrumentation libraries such as PAPI [18]. A general syntax of usage for PerfSuite as used in our experiments is shown as follows:

```
$ psrun -o perfOutput  -a -p ./program
< I > output.prt
```

Where *perfOutput* is the output previously generated by perf, program is the binary to profile, and *output.prt* is the result file to be generated by PerfSuite. *Output.prt* will then contain an easy to interpret set of profiling data.

The experiments were carried out using the Intel Core i5-2400 (3.1 GHz) and the Intel Xeon x5550 (2.66 GHz), with Linux (kernel 2.6.38). Most of the results are similar in both compilers and platforms, so we present averages and/or main characteristics which are independent of the specific hardware and compiler. Profiling has shown that more than 80% of the total runtime is spent in 21 subprograms (Fortran functions and subroutines). This means that most of the optimization and parallelization effort should be employed in those 21 subprograms. For large legacy applications, this could be a huge reduction in the amount of source code to work on.

As a first step in the process of enhancing performance of the legacy software, we used several optimization options, shown in Table I. Most of the improvement is obtained in the first optimization level, -O1, which implied to reduce the runtime to the 62% of the non-optimized binary code. In this case, the second optimization level added some gains (which is not always obtained, i.e. in all hardware and compiler variants).

**Table I. Optimization gains (time reduction)**

| -O1 | -O2 | -O3 |
|---|---|---|
| 0.62 | 0.52 | 0.51 |

We have collected available information from event counters and one of the first problems we found can be exemplified with the data in Table II, for a reduced number of counter data. Clearly, raw numbers collected from the hardware do not provide any useful information. However, with the event counters data it is possible to obtain information not only about optimizations, but also about optimization focus.

Table III shows the improvement in cache misses, i.e. the reduction in (instructions and data) caches and TLB (Translation Lookaside Buffer) misses when using O1 relative to those obtained with –O0 (no compiler optimization). Most of the performance improvement provided by the -O1 optimization level is due to the very good work of the compiler on the instructions. Compilers are able to optimize the available resources in hardware pipelines, superscalar units, branch prediction, and almost every hardware facility for ILP (Instruction Level Parallelism). Branch instruction event counters can be also used to support this behavior:

- Conditional branch instructions have been lowered by 11.95%.
- Mispredicted conditional branch instructions have been decreased by 36.49 %.
- Not taken conditional branch instructions have been decreased by 37.97 %.

**Table II. Raw event counters numbers**

| Event | Result |
|---|---|
| Conditional branch instructions | 111948989803 |
| Branch instructions | 130698632623 |
| Conditional branch instructions mispredicted | 1849659750 |
| Conditional branch instructions not taken | 24036555287 |
| Floating point divide instructions | 23168250057 |
| Floating point operations | 56515006398 |
| Level 1 data cache | 5609489524 |
| Level 1 instruction cache misses | 172229550 |
| Level 2 data cache accesses | 29185949460 |
| Level 2 instruction cache accesses | 222243552 |
| Level 2 instruction cache misses | 74014152 |
| Level 2 cache misses | 11534648622 |

Table III shows that almost no performance enhancement is obtained by taking advantage of data cache/s.

**Table III. Misses (caches and TLB) improvement with -O1**

| Event | Reduction |
|---|---|
| Level 1, 2, and 3 data cache misses | < 5% |
| Level 1 instruction cache misses | > 98% |
| Level 2 instruction cache misses | > 85% |
| Level 2 instruction cache accesses | > 85% |
| Instruction TLB misses | 53% |

It is worth noting that legacy code parallelization has a strong relationship with data usage (accesses to cache/s and main memory, i.e. the memory hierarchy), since it is nearly impossible to recode or change the underlying algorithms on almost unknown code. At least in the initial parallelization stages/tasks, the basic algorithm is kept unchanged and the way in which data and threads and/or processes is defined so that every processing facility is used. It is clear from Table III that specific data cache improvement has to be taken into account (preferably at early stages) in the parallelization work.

## 4. Parallelization Example

In [15] is reported the work on a very simple but time consuming algorithm used for N-body/particle simulation. It has also been shown how tiling (a very common optimization technique for memory accesses) has made possible a huge performance gain for two, four, and eight cores running OpenMP threads. More specifically:

- Tiling does not provide a huge improvement in sequential computing: less than 10% [15].
- When each OpenMP thread runs the tiled code, efficiency, calculated as in Eq. (2), becomes greater than 95% for 2, 4, and 8 cores (using two quad-core processors). Conversely, when tiling is not used, efficiency is just 0.69 for two threads and drops to 0.51 for eight threads (running on eight cores sharing main memory) [15].

We had not to determine if it is possible to identify the memory contention using performance monitoring counters. To answer this question, we used perf [19] in order to experiment and gather information about hardware. We have found that perf is a simple yet powerful tool, and easier to install than PerfSuite, which also depends on other software/libraries, such as PAPI. We run specific experiments with the program (coded in C language) for identifying cache events, i.e.:

- A small number of bodies: 100000, this is useful to avoid long experiments runtime as well as issues due to memory

accesses other than memory contention.
• Several different numbers of threads: 2, 4, 6, and 8.

The syntax used for our experiments using perf is shown as follows:

```
$ perf stat –e L1-dcache-load-misses
        -e LLC-load-misses
    ./sim4OMP 10000
```

Where all the profiled counters are defined after a *–e* flag. In our case, we only need L1 cache load misses, and LLC load misses to be profiled. Finally, the binary name and its arguments are specified. In our case, sim4OMP represents the OMP version of our N-body algorithm that is being run for 10000 and one step.

The results of this experiment are shown in Table IV, in terms of Mflop/s and Efficiency (Eff.), and the percentage of Last Level Cache (LLC) and L1 data cache misses relative to the "previous" number of threads experiment. Initially, we ran experiments with 2, 4, and 8 threads, and we found a huge performance loss specifically for 8 threads: efficiency dropped to about 0.47 for 8 threads, when it was about 0.98 when 4 threads process data. Thus, we added several experiments with one more number of threads in between 4 and 8, i.e. 6, so that we are able to analyze the performance problem in detail.

**Table IV. Cache load misses (relative %) and Mflop/s**

| #cores | 1 | 2 | 4 | 6 | 8 |
|---|---|---|---|---|---|
| **L1-dclm(1)** | 100 | 100 | 92 | 326 | 220 |
| **LLC-lm(2)** | 100 | 88 | 43 | 599 | 358 |
| **Mflop/s** | 1858 | 3669 | 7297 | 8275 | 6993 |
| **Eff.** | 1 | 0.99 | 0.98 | 0.74 | 0.47 |

**(1) L1-dclm:** Level 1 data cache load misses
**(2) LLC-lm:** Last Level Cache load misses

We use "previous" in the sense of number of "previous number of threads" because it highlights the main differences related to algorithm scalability. For example, Table IV shows that the number of L1 data cache misses is almost constant for 1, 2, and 4 threads, since for 2 cores, it is about 100% of the misses for reported for 1 core, and, rather surprisingly, the number of L1 data cache misses for 4 threads is 92% of the L1 data cache misses for 2 threads. Minor differences could appear given to the statistical nature of event counting by multiplexing hardware counters.

The parallel performance is very good For 2 and 4 threads, i.e. both have more than 95% of efficiency. There is a clear problem in performance and scalability for 6 and 8 threads. One could be confused by the increase of 326% of L1 data cache misses for 6 threads regarding the number of L1 data cache misses for 4 threads. Actually, the real problem is the huge increase of 599% LLC-lm for 6 threads regarding those for 4 threads. Part of that 599% LLC-lm is "hidden" by the other level/s of cache/s, but it is clear that data from main memory is not arriving at the rate the processor needs for computing. More specifically, there are 6 threads requiring data to process from the same memory, and from those 6 threads, 4 threads share the LLC in a quad-core processor while the other two needs almost the same data to process in the other processor. The situation is even worse for 8 threads, as expected: there are 358% more LLC load misses than for those happening for 6 threads.

These results show clearly that using more cores imply more shared memory contention

when the threads are not running in the cores of a unique processor (thus having access to the shared LLC). Even though processing-time is reduced by adding additional cores, the benefits of adding them are severely reduced. This lead us to think that there may be a shared LLC cache throughput limit that, at a certain demand, collapses and reduces the overall performance. Once this limit is reached, penalization for L1 cache misses is much more time-consuming. Therefore, the overall per-core speedup is reduced. In our case, this limit is reached when using more than 4 cores (on the 6 and 8 core cases).

Previous research [17] indicate us that this so-called *memory wall* is quickly found not only when a processor runs faster but also when more cores run in the same processor, which is our case for the 6-8 core run. The number of cores per processor and the number of processors sharing main memory can be increased, but performance would be unacceptable (or even disappointing) if memory and/or the algorithms are not improved/optimized/adapted. There is clear evidence now about what we explained by the end of the previous section: data cache improvement has to be taken into account (preferably at early stages) for parallel computing. We have collected and shown in Table IV specific performance monitoring event counters evidence.

## 5. Conclusions and Further Work

Until we performed this research, we had to rely on guessing techniques, and high-level analysis to determine which were the causes of performance degradation on our parallel and legacy codes. Although some of our results were positive [15] [16], we were still lacking a formal method to know the exact causes (or their amount) that caused this degradations. Now, most of this classical guesswork used in optimization and parallelization performance analysis is now supported by performance counters.

We have shown valuable specific runtime information is now possible to be gathered for almost unknown (legacy) and self developed source code. Furthermore, we can access to and collect information from the performance event monitoring counters from different programming languages, such as Fortran and C (which, besides, are among the most popular languages in HPC).

Even when the optimization and parallelization problems remain the same in the long term, performance counters provide very useful information for both tasks. Moreover, new possibilities could be explored based on specific combinations of values. We have also shown that single events are not necessarily good enough for performance analysis, but having a minimum knowledge of hardware architecture/s will lead to combine events so that the searching space for optimizations and parallelization would be narrowed down without missing important details.

Performance event counters do not solve by themselves optimization- and parallelization-related problems, they help taking informed/supported decisions. Moreover, low level hardware details (those accounted for by event counters) could produce an unmanageable amount of information and should be used carefully. So far, there is no general methodology for approaching a program using performance event counters, and we are working on producing one. The final objective would be a

tool o set of tools for aiding HPC programmers for optimization and parallelization.

Similar experiences have also been shared by scientific programmers, for both high-performance computing and parallel algorithms in these last few years [20] [21]. These experiences along with the ones presented in this article can be used for further research trying to establish a knowledge base for algorithm optimization using profiling tools.

## References

[1] Advanced Micro Devices, AMD64 Technology, AMD64 Architecture Programmer's Manual, Volume 2: System Programming, March 2012.

[2] Advanced Micro Devices, Software Optimization Guide for AMD Family 10h and 12h Processors, February 2011. Available at http://support.amd.com/us/Processor_ TechDocs/40546.pdf

[3] Advanced Micro Devices, Software Optimization Guide for AMD Family 15h Processors, Available at http://support.amd.com/us/Processor_TechDocs/47414_15h_sw_opt_guide.pdf

[4] Advanced Micro Devices, Surviving and Thriving in a Multi-Core World, Nov. 2006.

[5] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Stephen S. Pawlowski, Justin R. Rattner, Platform 2015: Intel Processor and Platform Evolution for the Next Decade, Intel White Paper, 2005. Available at ftp://download.intel.com/tech nology/computing/archinnov/platform2015/download/Platform_2015.pdf

[6] Intel Corp., Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, Aug. 2012.

[7] Intel Corp., Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide (Nehalem Core PMU), 2010. Available at http:// software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf

[8] Rick Kufrin, "PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux", 6th International Conference on Linux Clusters: The HPC Revolution, Chapel Hill, NC, April 2005.

[9] David Levinthal, "Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors", Intel Corp., 2009. Available at http://software.intel.com/ sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[10] Gregory Pfister, In Search of Clusters, 2nd Edition, Prentice Hall, Dec. 1997, ISBN 0138997098.

[11] Gary L. Russell, James R. Miller, David Rind, "A Coupled Atmosphere-Ocean Model for Transient Climate Change Studies", Atmosphere-ocean, 33(4), 1995.

[12] Gerald Schubert, Holger Fehske, Georg Hager, Gerhard Wellein, "Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems", Parallel Processing Letters 21(3), 2011.

[13] Thomas L. Sterling, Beowulf Cluster Computing With Linux, MIT Press, 2001, ISBN 0262692740.

[14] Herb Sutter "The Free Lunch is Over: a Funda-

mental Turn Toward Concurrency in Software", Dr. Dobb's Journal, Vol. 30, No. 3, 2005, http://www.gotw.ca/publica tions/concurrency-ddj.htm.

[15] Fernando G. Tinetti, Sergio M. Martin, "Sequential and Shared and Distributed Memory Parallelization in Clusters: N-Body/Particle Simulation", 24th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2012, November 12 – 14, 2012, Las Vegas, USA.

[16] Fernando G. Tinetti, Mariano Méndez, Fortran Legacy Software: Source Code Update and Possible Parallelization Issues, ACM Fortran Forum, April 2012, Vol. 31, No. 1.

[17] Wlliam A. Wulf, Sally A. Mckee, "Hitting the Memory Wall: Implications of the Obvious", ACM SIGARCH Computer Architecture News, Vol. 23, 1995.

[18] PAPI, http://icl.cs.utk.edu/papi/

[19] Tutorial - Perf Wiki, https://perf.wiki.kernel.org/in dex.php/Tutorial

[20] Wucherl Yoo, "Automated Performance Characterization of Applications Using Hardware Monitoring Events", Doctoral Dissertation. University of Illinois at Urbana-Champaign. 2012, Illinois, USA.

[21] Xingfu Wu, Valerie Taylor, "Performance Modeling of Hybrid MPI/OpenMP Scientic Applications on Large-scale Multicore Supercomputers", Elsevier, Journal of Computer and System Sciences. May 2012, Vol. 79, No. 8.