ABSTRACT
Current hardware for compute intensive tasks includes a large amount of processing facilities which is sometimes hard to use in an optimized way. High performance computing (HPC) is always focused in solving grand challenge (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 bare hardware performance 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. Eventually, those guidelines could be transformed in a set of tools to aid the scientific programmer in the task of solving the computational problems in the minimum time, i.e. optimizing the usage of available hardware/hardware facilities.

KEY WORDS

1. Introduction
High performance computing (HPC) in clusters is one of the most popular approaches for solving compute or specifically numerical intensive workloads/tasks. Current computers used as nodes in the clusters usually range from more or less low end desktop/PC computers (which are economically cheap) to high end servers/PCs. Terms such as low end desktop computer and server computer take many definitions depending on hardware, hardware corporations marketing, specific market status, etc. We will refer to them just as PCs or cluster nodes, making emphasis on their proven advantageous features, 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 usual 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 (e.g. Infiniband).
software has a strong need to be updated to the new multicore 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 (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.

The rest of this article is organized as follows: Section 2 introduces the initial non optimized algorithm. The current hardware used for HPC which should be taken into account for optimizations as well as parallelization is briefly explained in Section 3. The most successful sequential optimizations are shown in Section 4. Section 5 explains the parallelization for (shared memory) multiprocessor environments via OpenMP (i.e. with several threads of execution) and its relationship with memory access optimization. The parallelization for distributed memory environments, usually clusters, via MPI (Message Passing Interface) is introduced in Section 6. Section 7 explains several details related to performance from the point of view of optimization and parallelization. Finally, Section 8 includes conclusions and further work.

2. Performance 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, 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.

\[
Speedup(p) = \frac{op_{st}}{pt(p)} \tag{1}
\]

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

\[
Efficiency(p) = \frac{Speedup(p)}{p} \tag{2}
\]

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, also, 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 [19] 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 point 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 [8] as a high level approach for gathering hardware performance events information, i.e. to avoid lower level tools such as perf [19] and instrumentation libraries such as PAPI [18].

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>
<thead>
<tr>
<th>Event</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional branch instructions</td>
<td>111948989803</td>
</tr>
<tr>
<td>Branch instructions</td>
<td>130698632623</td>
</tr>
<tr>
<td>Conditional branch instructions mispredicted</td>
<td>1849659750</td>
</tr>
<tr>
<td>Conditional branch instructions not taken</td>
<td>24036555287</td>
</tr>
<tr>
<td>Floating point divide instructions</td>
<td>23168250057</td>
</tr>
<tr>
<td>Floating point operations</td>
<td>56515006398</td>
</tr>
<tr>
<td>Level 1 data cache</td>
<td>5609489524</td>
</tr>
<tr>
<td>Level 1 instruction cache misses</td>
<td>172229550</td>
</tr>
<tr>
<td>Level 2 data cache accesses</td>
<td>29185949460</td>
</tr>
<tr>
<td>Level 2 instruction cache accesses</td>
<td>222243552</td>
</tr>
<tr>
<td>Level 2 instruction cache misses</td>
<td>74014152</td>
</tr>
<tr>
<td>Level 2 cache misses</td>
<td>11534648622</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Event</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1, 2, and 3 data cache misses</td>
<td>&lt; 5%</td>
</tr>
<tr>
<td>Level 1 instruction cache misses</td>
<td>&gt; 98%</td>
</tr>
<tr>
<td>Level 2 instruction cache misses</td>
<td>&gt; 85%</td>
</tr>
<tr>
<td>Level 2 instruction cache accesses</td>
<td>&gt; 85%</td>
</tr>
<tr>
<td>Instruction TLB misses</td>
<td>53%</td>
</tr>
</tbody>
</table>

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].

Is it possible to identify the memory contention using performance monitoring counters? We have 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.

and the results 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 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. Clearly, more threads 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).

The so-called memory wall [17] is quickly found not only when a processor runs faster but also when more cores run in the same processor. 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

Most of the 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

<table>
<thead>
<tr>
<th>#cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-delm(1)</td>
<td>100</td>
<td>100</td>
<td>92</td>
<td>326</td>
<td>220</td>
</tr>
<tr>
<td>LLC-lm(2)</td>
<td>100</td>
<td>88</td>
<td>43</td>
<td>599</td>
<td>358</td>
</tr>
<tr>
<td>Mflop/s</td>
<td>1858</td>
<td>3669</td>
<td>7297</td>
<td>8275</td>
<td>6993</td>
</tr>
<tr>
<td>Eff.</td>
<td>1</td>
<td>0.99</td>
<td>0.98</td>
<td>0.74</td>
<td>0.47</td>
</tr>
</tbody>
</table>

(1) L1-delm: 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,
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.

References


