Performance and Energy Analysis of OpenMP Runtime Systems with Dense Linear Algebra Algorithms

João V. F. Lima*, Issam Raïs§, Laurent Lefèvre§, Thierry Gautier§

*UFSM, Brazil
§Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude-Bernard Lyon 1, LIP, France

Abstract—In this paper, we analyse performance and energy consumption of four OpenMP runtime systems over a NUMA platform. We present an experimental study to characterize OpenMP runtime systems on the three main kernels in dense linear algebra algorithms (Cholesky, LU and QR) in terms of performance and energy consumption. Our experimental results suggest that OpenMP runtime systems can be considered as a new energy leverage. For instance, a LU factorization with concurrent write extension from libKOMP achieved up to 1.75 of performance gain and 1.56 of energy decrease.

1. Introduction

Energy-efficiency is one of the four major challenges that should be overcome in the path to exascale computing [1]. Despite improvements in energy-efficiency, the total energy consumed by supercomputers is still increasing due to the even quicker increase in computational power. High energy consumption is not only a problem of electricity costs, but it also impacts greenhouse emissions and dissipating the produced heat can be difficult. As the ability to track power consumption becomes more commonplace, with some job schedulers supporting tracking energy use [2], soon users of HPC systems may have to consider both how many CPU hours they need and how much energy.

Energy budget limitation imposes a high pressure to the HPC community making energy consideration a prominent research field. Most of the gain will come from technology by providing more energy efficient hardware, memory and interconnect. Nevertheless, recent processors integrate more and more leverages to reduce energy consumption (e.g. classical DVFS, deep sleep states) and low level runtime algorithms provide orthogonal leverages (e.g. dynamic concurrency throttling). However few of these leverages are integrated and employed in today local level software stack such as middleware, operating system or runtime library. Due to the complexity of this statement, we restricted our investigation to local node energy consumption by HPC OpenMP applications.

OpenMP is an API standard to express parallel portable programs. Most of controls are implementation defined and rely on the specific OpenMP programming environment used. The OpenMP standard does not impose any constraint on implementations. Even if there are more precise specifications, e.g. mapping of threads to cores, it is very tricky to precisely control performance or energy consumption using what OpenMP specification proposes [3]. Previous works have dealt with a specific OpenMP runtime [4], [5], [6], [7], [8], [9] that may be difficult to generalize to other OpenMP runtime systems without strong development effort. To the knowledge of the authors, there is no related work comparing OpenMP runtime systems in order to analyse performance and energy consumption.

In this paper, we analysed performance and energy consumption of four OpenMP runtime systems over a NUMA system. We restrict our experiments on three dense linear algebra algorithms: Cholesky, LU and QR matrix factorizations. Source codes are based on KASTORS [10] benchmark suite and the state of the art PLASMA library using its new OpenMP implementations [11] that rely on OpenMP tasks with data dependencies.

The contributions of this paper are:

• We present early experiments of performance and energy consumption over the dependent tasks model proposed by OpenMP.
• We report early comparisons of OpenMP runtime systems in order to present the respective gains with respect to one of the criteria.
• We observed that a LU factorization with concurrent-write access mode achieved up to 1.75 in performance gain and 1.56 in energy over original LU algorithm.

The remainder of the paper is organized as follows. Section 2 presents the related work. Section 3 gives some details of the OpenMP task programming model and an overview about five runtime implementations. Our experimental results are presented in Section 5. Finally, Section 6 and Section 7, respectively, present the discussion and conclude the paper.

2. Related Work

Other works use the simplicity proposed by OpenMP to vary the number of threads, for energy efficiency. Authors in [12] and [13] defend the Dynamic Concurrency Throttling (DCT) and underline the fact that using OpenMP to control the number of threads could be energy efficient, depending on the algorithm or the chosen hardware.
Previous works show that various energy behaviors of computing nodes are possible through various leverages (DVFS, DCT, etc). But none of the previous work focus on OpenMP runtime systems as a leverage. None of the previous work dealt with the energy-performance trade-off and thus underlined possible variability concerning energy and performance for existing runtime systems. Thus, to the knowledge of the authors, no related work were trying to compare several OpenMP runtime libraries together for various representative workloads, as presented in our paper.

We use state of the art PLASMA library [11], on three main kernels in dense linear algebra (Cholesky, LU and QR factorizations), that implements dependent tasks model. This model is new and never addressed in related works. In [4], [5] the authors based their experiments using the BOTS [14] benchmarks that require only the independent tasks.

3. OpenMP Task programming model and implementations

In 2013 the OpenMP Architecture Review Board introduced in the OpenMP revision 4.0 a new way of expressing task parallelism using OpenMP, through the task dependencies. This section introduces the task dependency programming model targeted by the selected benchmark suites. We also present how the model is implemented in various runtime libraries.

3.1. Dependent task model

OpenMP dependent task model allows to define dependencies between tasks using declaration of accesses to memory with in, out, or inout. Two tasks are independent (or concurrent) if and only if they do not violated the data dependencies of a reference sequential execution order.

Figure 1 illustrates a LU factorization based on PLASMA [11]. The programmer declares tasks and the accesses in, inout they made to a memory region (here only lvalue or memory reference, i.e. pointer).

The OpenMP library computes tasks and dependencies at runtime, and schedules concurrent tasks on the available processors. The strategy for task dependencies and task scheduling depends on the runtime implementation. Nevertheless, their implementations impact the performance and the energy consumption. Moreover, the absence of precise OpenMP specification about the task scheduling algorithm is the key point to allow research to improve performance and energy efficiency with implementation concerns.

3.2. Runtime system implementations

Table 1 summarizes the properties of four OpenMP runtime systems.

libGOMP is the OpenMP runtime that comes with the GCC compiler. Dependencies between tasks are computed through a hash table that map data (pointer) to the last task writing the data. Ready tasks are pushed into several scheduling dequesues. The main dequeue stores all the tasks generated by the threads of a parallel region. Tasks seem to be inserted after the position of their parent tasks in order to keep an order close to the sequential execution order. Because threads share the main dequeue, serialization of operations is guaranteed by a pthread mutex which is a bottleneck for scalability. To avoid overhead in task creation, libGOMP implements a task throttling algorithm that serialize task creation when the number of pending tasks is greater than a threshold proportional to the number of threads.

libOMP was initially the proprietary OpenMP runtime of Intel for its C, C++ and Fortran compilers. Now it is also the target runtime for the LLVM/Clang compiler and sources are open to community. libOMP manages dependencies in the same way that libGOMP by using a hash table. Memory allocation during task creation relies on a fast table. Task allocation relies on a fast table that map data (pointer) to the last task writing the data. Ready tasks are pushed into several scheduling dequesues. The main dequeue stores all the tasks generated by the threads of a parallel region. Tasks seem to be inserted after the position of their parent tasks in order to keep an order close to the sequential execution order. Because threads share the main dequeue, serialization of operations is guaranteed by a pthread mutex which is a bottleneck for scalability. To avoid overhead in task creation, libGOMP implements a task throttling algorithm that serialize task creation when the number of pending tasks is greater than a threshold proportional to the number of threads.

libOMP was initially the proprietary OpenMP runtime of Intel for its C, C++ and Fortran compilers. Now it is also the target runtime for the LLVM/Clang compiler and sources are open to community. libOMP manages dependencies in the same way that libGOMP by using a hash table. Memory allocation during task creation relies on a fast table. Task allocation relies on a fast table that map data (pointer) to the last task writing the data. Ready tasks are pushed into several scheduling dequesues. The main dequeue stores all the tasks generated by the threads of a parallel region. Tasks seem to be inserted after the position of their parent tasks in order to keep an order close to the sequential execution order. Because threads share the main dequeue, serialization of operations is guaranteed by a pthread mutex which is a bottleneck for scalability. To avoid overhead in task creation, libGOMP implements a task throttling algorithm that serialize task creation when the number of pending tasks is greater than a threshold proportional to the number of threads.

libOMP was initially the proprietary OpenMP runtime of Intel for its C, C++ and Fortran compilers. Now it is also the target runtime for the LLVM/Clang compiler and sources are open to community. libOMP manages dependencies in the same way that libGOMP by using a hash table. Memory allocation during task creation relies on a fast table. Task allocation relies on a fast table that map data (pointer) to the last task writing the data. Ready tasks are pushed into several scheduling dequesues. The main dequeue stores all the tasks generated by the threads of a parallel region. Tasks seem to be inserted after the position of their parent tasks in order to keep an order close to the sequential execution order. Because threads share the main dequeue, serialization of operations is guaranteed by a pthread mutex which is a bottleneck for scalability. To avoid overhead in task creation, libGOMP implements a task throttling algorithm that serialize task creation when the number of pending tasks is greater than a threshold proportional to the number of threads.
dependencies on steal request, which is a perfect application of
the work first principle to report overhead in task creation
to critical path. The XKaapi based OpenMP runtime also
has support to some OpenMP extensions such as task affin-
ity [19] that allows to schedule tasks on NUMA architecture,
and to increase performance by reducing memory transfer
and thus memory energy consumption.

libKOMP [20] is a redesign of [17] on a top of the
Intel runtime libOMP. It includes following features coming
mainly from XKaapi: the dequeue management and work
stealing with request combining; task affinity specific work
stealing heuristic; a dynamically resized hash map that avoid
high conflicts when finding dependencies for large tasks’
graph; and tracing tool based on the OpenMP OMPT API;
and finally a task concurrent write extension with a Clang
modification 2 to provide the OpenMP directive clause. This
latter extension allows better parallelism and was used in
one of our LU benchmark and it very closed of the task
reduction feature currently under discussion in the OpenMP
architecture review board.

3.3. Discussion

In our study of the mentioned OpenMP runtime systems,
none of them include energy leverage such as thread throt-
tling or DVFS. Nevertheless, their different task scheduling
algorithms may impact energy efficiency. The main dequeue
accesses in libGOMP serialize threads using a POSIX mu-
tex. On Linux the mutex will block waiting threads after
short period of active polling which ensure that few core
cycles will be waste in the synchronisation.

On the other hand, libOMP, XKaapi and libKOMP work
stealing actively poll dequeues until the program ends or a
task is found. In order to reduce activity during polling,
libOMP and libKOMP may block threads after an unsuc-
cessful search of work by 200ms (default value). Once work
is found, all threads are waked up.

4. Tools and Methods

This section details the hardware configurations we ex-
perimented on and the OpenMP runtime systems we com-
pared. We also give hints about the methodology used to
process the collected data using statistical tools R.

4.1. Evaluation platform

Our experimental platform was the Brunch machine
composed of four NUMA nodes with one Intel Xeon E7-
8890 processor each (total 4 processors) and 24 cores per
processor (96 cores total) running at 2.2GHz, and 1.5 TB of
main memory. The operating system on Brunch is a Debian
with Linux kernel 4.9.13 with 3 over 5 C-State activated
(idle states: POLL C1-BDW C1E-BDW) with turbo-boost
on and performance governor.

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>BRUNCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>8192</td>
<td>224</td>
</tr>
<tr>
<td>16384</td>
<td>288</td>
</tr>
<tr>
<td>32768</td>
<td>352</td>
</tr>
</tbody>
</table>

4.2. Software description

4.2.1. Benchmarks. We used kernels from two bench-
mark suites: the KASTORS [10] benchmark suite and an
OpenMP-parallelized PLASMA version [11]. Both bench-
mark suites tackle the same computational problems but use
different algorithms in some cases. KASTORS was built
from PLASMA 2.6.0 (released in dec. 2013) at a time
when PLASMA parallelism was supported by a specific task
management library called QUARK.

We focused our study on three dense linear algebra
kernels:

- A Cholesky factorization (dpotrf);
- A LU factorization (dgetrf);
- A QR factorization (dgeqrf).

Cholesky factorization algorithms in both the benchmark
suites are the same. All these linear algebra kernels we used
rely on the BLAS routines, we used the implementation of
OpenBLAS version 0.2.19. Table 2 shows the block size
configuration on each execution test for the three machine
platforms.

4.2.2. Runtime Systems. We compared the following run-
time systems during our experiments:

- LibGOMP – the OpenMP implementation from
  GNU that comes with GCC 6.3.0.
- LibOMP – a port of the Intel OpenMP open-source
  runtime to LLVM release 4.0.
- LibKOMP [20] – a research runtime system, based
  on the Intel OpenMP runtime, developed at INRIA.
  It offers several non-standard extensions to OpenMP.
  We evaluate the concurrent write (CW) feature in
  our experiments coupled with Cilk T.H.E work steal-
  ing protocol. We make experiments with version
  efb6c36.
- XKAAPI [16] – research runtime system developed
  at INRIA. It has lightweight task creation over-
  head, and it offers several non-standard extensions
to OpenMP [17] We evaluate its version efa5fd1.

4.3. Energy measurement methodology

Since several metrics have to be considered depending
on the objective, we consider performance (GFlop/s) and
energy consumption (energy-to-solution). GFlop/s is mea-
sured by the each benchmark itself: it corresponds to the
Algorithmic count of the number of floating point operations over the elapsed time, using fact that matrix-matrix product does not rely on a fast algorithm such as Strassen like algorithm. Times are get using the Linux clock_gettime function with CLOCK_REALTIME clock.

We employed two sources of data acquisition for energy measurement. One was the Intel RAPL (Running Average Power Limit) feature that exposes the energy consumption of several components on the chip (such as the processor package and the DRAM) through MSRs (Model Specific Registers). Due to access limitation of MSRs on the tested system, we designed a small tool querying periodically the RAPL counters based on LIKWID [21]: Energy consumption for the whole package (PWR_PKG_ENERGY), for the cores (PWR_PKG0_ENERGY), for the DRAM (PWR_DRAM_ENERGY), as well as the core temperature (TEMP_CORE). The tool gets the counter values periodically and associate them with a timestamp.

The Brunch machine has been instrumented through a high-accuracy (error < 0.1%) power meter LMG450 from Zimmer®. The power meter is attached to the wall outlet and it measures the entire energy consumption of the machine, including power supply, disk, motherboard, etc. The output of the power meter (energy and power) periodically send data recorded with a timestamp. The Brunch machine measured 176W to the wall outlet on the same period of inactivity. The idle power is a mean of 20 minutes of inactivity on the system with a process monitoring the RAPL counters.

4.4. Experimental methodology

All benchmarks are composed of two steps: the first allocates and initializes a matrix; the second step is the computation. We report execution time only from the computation step. Each experiment is repeated at least 30 times, each computation on a newly random matrix (as implemented by the benchmark). All the processes are spawned within the context of numactl to distribute memory pages among the NUMA nodes. In parallel of the computation, we monitor the system by collecting various energy counters from RAPL and the watt meter plugged on the wall outlet.

For each computation we collect the performance (GFlop/s) timestamped by the beginning and the end of the computation. This two timestamps are used in data post-processing to compute energy consumed by the computation between the two timestamps. Values are interpolated by linear function if missing in the collected energy values sampled periodically. Post-processing employs R script to compute energy per computation and to output basic statistic for each configuration. In our experimental results, energy values are the mean computed among the at least 30 computations of each configuration.

5. Experimental results

The presented runtime systems have been experimented on the two benchmark suites presented in section 4.2.1. We build two configurations of libKOMP using two sets of options [20]. On the following libkomp refers to libKOMP configured with T.H.E Cilk work stealing queue and requests combining protocol; and libkomp_cw is the same configuration than libkomp with addition to support concurrent write extension used in the KASTORS LU code dgetrf [10].

5.1. Runtime impact

Figure 2 shows performance and energy results with three matrix sizes and over all machine resources available. We used as reference the GCC runtime to compute the difference over the other three runtime systems, represented on the bar plots by a percentage value.

These results suggest that libkomp and xkaapi attained the best performance results in most cases. Xkaapi outperformed others with Cholesky and QR on smaller input sizes (8192 and 16384), while libkomp had better results with input size 32768 on both algorithms. The LU algorithm with CW showed significant improvement compared to other runtime systems (up to 107.4% over gcc), followed by state of the art LU with libkomp.

In energy our experiments suggest that gcc had generally better energy efficiency on the three benchmarks, except for LU with CW. Besides, if we compare only the original runtime systems coming with gcc and Intel compilers, it seems that gcc configuration performed better in performance and energy. This can be explained by the passive list scheduling in gcc, which is less reactive than work-stealing based strategies. Regarding LU energy results, the CW LU version with libkomp_cw reduced energy up to 24% (RAPL) and 31% (ZES) over gcc. Other runtime systems had lower energy efficiency than gcc.

5.2. Focus on LU factorization

Thanks to the concurrent write, the LU algorithm with libkomp_cw runtime had more parallelism than other runtime systems due to the CW algorithm extension based on KASTORS [10]. Figure 3 illustrates a Gantt execution from the LU factorization using libkomp_cw.

On the LU factorization, even if CW generates more parallelism, the algorithm has poor efficiency and threads are frequently idle. The Gantt diagram on all the 96 cores of brunch illustrates long periods of inactivity. Gcc is the only runtime where threads lock common dequeue to get task. Linux will put these lightweight process idle which is captured by energy sensors (ZES on brunch and by using RAPL counters). If we do not consider libkomp_cw’s algorithmic variant, then gcc is the best runtime in term of energy consumption for LU factorization. This is not true for runtime systems based on task scheduling by work stealing such as libKOMP, libOMP or XKaapi which are very active process that consume energy.

5. https://www.zes.com/en/Products/Precision-Power-Analyzzer/LMG450
6. Discussion

Majority of best configurations from Figure 2 were runtime systems using work-stealing based scheduling. On fine grain problems, xkaapi and libkomp were generally better. These results can be explained by the smaller task creation overhead on xkaapi than libkomp and gcc. The difference between libomp and libkomp is the new features we add into the original Intel libomp runtime: the lightweight work stealing algorithm from Cilk and the request combining protocol from xkaapi. These features not only impact performance but also the way tasks are scheduled: it suppresses the bounded dequeue limitation that may degenerate task creation into task serialization. It means that at runtime a thread may be forced to execute immediately tasks for which no or less affinity exist. Without bounded size dequeue, a thread that completes a task will always activate one of the successors following a data flow relationship producer-consumer, thus sharing a data resident into cache; or the thread becomes idle and try to steal tasks. We will investigate by more finer experiments the exact impact of these additions in libkomp.

On LU factorization where algorithmic variant libkomp_cw was the best, it was followed by xkaapi and libkomp on performance. LU factorization is a relevant code with inactivity sections from the dependencies imposed by the algorithm, mainly due to a search of pivot and swap of elements. This optimized algorithm allowed to increase performance while energy is decreased due to libkomp_cw runtime and concurrent write OpenMP extension [10]. Nevertheless, the platform characteristic, and especially its memory network, had also an impact on both performance and energy consumption.

Without these algorithm variants, LU factorization code consumes less energy using gcc runtime. In gcc the synchronization between threads on the shared task dequeue resource wastes less cycles. A work stealing based runtime may have interest to incorporate part of [22] in which is used to lower the speed of threads that are not in the critical path with a warranty on performance. One of the big challenges is the design of adaptive OpenMP runtime capable to saving energy on short delays of inactivity.
7. Conclusion

In this paper, experiments with four production based OpenMP runtime systems on the three main kernel in dense linear algebra were conducted on a NUMA platform. We showed that OpenMP runtime is a new leverage for controlling energy. Our experimental results suggest that small algorithmic and runtime improvements may allow performance gains up to 1.75 and thus reducing the energy by a factor of 1.56. Besides, GCC runtime was energy efficient in some cases due to synchronizations over a shared task dequeue.

Future works include an extension of our experimental comparison over a wide range of architectures, including Intel Xeon KNL many-core. In addition, we will evaluate the impact on performance and energy of different configurations of leverages to control processor consumption and activity as available at operating system level.

Acknowledgments

This work is integrated and supported by the ELCI project, a French FSN ("Fond pour la Société Numérique") project that associates academic and industrial partners to design and provide a software environment for very high performance computing.

References


