Enhancing Energy Efficiency using Efficient Parallel Programming Techniques *

Marco A. Z. Alves, Márcia C. Cera, João V. F. Lima,
Nicolas Maillard, and Philippe O. A. Navaux
Universidade Federal do Rio Grande do Sul (UFRGS), Informatics Institute
Av. Bento Gonçalves, 9500 - Porto Alegre, Rio Grande do Sul, 91501-970, Brazil
{mazalves, marcia.cera, joao.lima, nicolas, navaux}@inf.ufrgs.br

Abstract

Currently, large-scale parallel architectures include processors with increasing number of cores. They are high processing power systems and consequently large energy consumers. Thus, energy consumption becomes a relevant issue on High Performance Computing with many hardware level initiatives to allow energy saving. In software level, the adoption of efficient parallel algorithms can also contribute to save energy, since they are able to better explore the parallel architecture features. This paper aims to show the impact of algorithms choices on energy consumption. In this sense, we will analyze the use of energy of different high performance implementations in a real scenario. We conclude that green computing issues can be supplied by adopting efficient parallel programming techniques.

1 Introduction

Nowadays, high performance computing uses aggressive techniques in order to obtain parallelism on multiple levels: on the instruction level using pipelines and superscalarity with Out-Of-Order execution (OOO) [25]; on the thread and process level using multi-core machines and even clusters of many multi-core nodes. However, instruction level parallelism (ILP) techniques such as deep pipelines, larger OOO execution windows, frequency increase, and others [12, 26], have less room due to walls of wire-delay problems, power consumption [5], and ILP extraction problems [1].

Considering this high performance scenario, processors with increasing number of cores became the most likely way to the industry to continue delivering more powerful processors on each new generation. The focus changed from the instruction parallelism to the thread and process parallelism [27]. Multi-core processors are built with cores simpler than traditional single cores [21], leading the possibility of putting more cores on the same physical silicon area. This complexity reduction on the cores is also beneficial for the power constraints inside of the chip.

Power consumption is becoming very relevant considering that many companies and research centers use large-scale clusters with lots of cores. In this context, each technique that leads to power consumption reduction has great impact on the final system consumption. Thus, new processors begin to present frequency scaling options, auto subsystems power off and others techniques to ensure that machines will spend energy on demand.

However, such energy saving techniques do make sense only if High Performance Computing (HPC) programmers use efficient algorithms even for the simpler tasks where performance is not the main constraint. Hence, parallel programming is the key to leverage the energy efficiency. Many recent studies describe the Explicit Task Parallelism paradigm as a technique to extract parallelism of the algorithms at runtime [2, 17, 23]. It allows parallel applications to adapt their behavior and unfold the parallelism on demand; consequently, they improve resource utilization as well as energy consumption.

Our study aims to show the importance of HPC on the green computing context, showing that the programmer has a major role on the energy consumption in current and future parallel architectures. Some experiments are presented in order to illustrate the correlation between energy consumption and performance on parallel machines.

The remainder of this paper is structured as follows. Section 2 describes the current hardware level techniques aiming at energy efficiency. Afterward, there is an overview of the efficient parallel programming techniques in Section 3. Section 4 describes the architectural environment, the parallel programming techniques and applications tested. The experimental results are exposed and analyzed in Section 5. Section 6 presents our final remarks and conclusions, as well as our future works perspectives.

*This work was partially supported by CAPES and CNPq.

2 Energy Efficiency Context

Embedded system architects are very familiar with low power budget systems designs; however, power constraint arrived on general purpose systems. This change, among other factors, relies that on the past the sole focus was on the processing power and associated equipment spending, while infrastructure (including power, cooling and data center space) was always assumed available. But, today, the infrastructure is becoming a limiting factor that can determine how and if IT equipment can be deployed [28]. In this new scenario with given peak power budget, different level approaches have been studied in order to match performance and power for future general purpose systems.

Considering the multi-core chips, the works from Isci [15] and Sartori [24] propose dynamic techniques to global CMP (Chip Multiprocessor) monitoring, control and management of the power. They estimate the application behavior across different operating modes, preventing instantaneously that the power exceed the peak budget. Thus, the authors can dynamically control the power consumption of all cores inside of the chip in order to achieve high performance with low energy consumption.

About future many-core chips, the work from Woo [29] takes power and energy into account to predict the design of many-core processors. The paper suggests a many-core alternative, featuring many small and energy-efficient cores integrated with a full-blown processor, in order to achieve high energy efficiency. Once many-core processors aim to join tens of cores, as the Intel tera-scale project processor with 80 cores and 3D stacked memory [3], thermal problems shall arrive together with power challenges [20]. In context, more than reducing the total power consumption, the systems will need to reduce the total heat dissipation. Huang's [13] paper shows the relevance of the theme, presenting a study about temperature-aware design for many-core processors and investigating the relationship between core size and on-chip hot spot temperature. It concludes that with the same power density, smaller cores are cooler than larger cores due to a spatial low-pass filtering effect of temperature.

On the cluster level, the study presented by Khargharia [16] addresses the power consumption on high performance servers platform. In this study, the processor and/or the memory subsystem are dynamically reconfigurated to suit the application resource requirements.

On cloud computing scale, we can also cite works which aims to improve energy savings. For instance, Duy [8] presents a Green Scheduling Algorithm which is a predictor based on neural network for energy saving in Cloud computing.

By the presented works it is notable that many advances on power consumption techniques have been proposed on processors, cluster and cloud levels. These researches clearly affect the final system energy consumption; however, the programmer has its contribution on the overall energy consumption, which is discussed in our paper. Next section presents some programming approaches that will be evaluated with focus on the energy consumption.

3 Efficient Parallel Programming

Besides the initiatives described in the previous section, which are focused on hardware, there is an increasing interest in energy-efficient applications, *i.e.* those are able to provide an efficient use of the energy. The development of energy-efficient applications is directly related to HPC, since high performance techniques improve the application execution and, consequently, save energy. In this section, we will discuss about current parallel programming techniques and their use for energy-efficient programming.

Recent studies take advantage of the Explicit Task Parallelism programming paradigm to allow the extraction of parallelism on demand [2, 17, 23]. According to this paradigm, the programs are structured as abstract tasks, which are generated at runtime, unfolding the parallelism on demand. The programmers basically identify independent units of work (abstract tasks), dependencies among them, and the runtime scheduler balances the workload [19]. Moreover, this paradigm is efficient to deal with irregular problems, in which the workload depends on the input data and can vary during the execution. Due to its adaptive nature, we believe that the explicit tasks parallelism can support energy-efficient applications.

Aiming to provide high performance in multi-core architectures, several shared-memory programming interfaces or languages support the explicit tasks parallelism, *e.g.*, Cilk [17], OpenMP 3.0 [2] and Intel *Threading Building Blocks* (TBB) [23]. These APIs attend the explicit task parallelism offering means to define abstract tasks, to synchronize tasks solving dependencies among them, and to schedule the dynamic tasks on-the-fly.

While Cilk represents tasks as procedures (the keyword `spawn` generate dynamic tasks), OpenMP uses a block of instructions (defined by the *task* construct), and TBB uses instances of a task class (deriving from *tbb::task* abstract class). The dependencies among tasks are expressed as barriers using keywords in Cilk (*sync*) and OpenMP (*taskwait*), whereas TBB allows synchronization with either the *blocking* style (similar to Cilk) or the *continuation* style (asynchronous standby). Cilk and TBB scheduling is based on Work Stealing (an idle thread chooses randomly another to steal some workload), while OpenMP includes several simple strategies and is still under development.

In distributed-memory systems, explicit task parallelism features are achieved in KAAPI (Kernel for Adaptative,

Asynchronous Parallel and Interactive programming) [9]. KAAPI tasks are function calls (generated by *fork*), which returns no value except through the access of a global address space called *global memory*. The dependencies are solved accessing shared objects on the global memory. KAAPI scheduling also is based on Work Stealing, but it was adapted to distributed-memory context: several local stealing are performed before request remote ones.

The standard parallel API for HPC in distributed-memory is the MPI (Message-Passing Interface) [10]. It is an interface that allows inter-processes communications, in which are defined issues about point-to-point and collective communications, as well as organization of the programs such as groups and topologies of the processes and communications contexts. Since the definition of MPI-2 [11], it specifies the dynamic process creation feature, in which new processes can be spawned at runtime providing some flexibility to the MPI applications. This feature can be explored to adaptive MPI applications with a behavior close to that achieved by explicit task parallel applications. In the next section, we will explain how an adaptive behavior can be reached in MPI applications.

4 Proposed Evaluations

This paper aims to analyze the impact of algorithmic choices on energy consumption. In this sense, we monitor the power consumption of latest generation multi-core processors that implements energy saving techniques in the hardware level, as described in Section 4.1. For the software level, the parallel programming techniques as well as the test applications are described in Section 4.3.

4.1 Architecture and Monitoring Issues

In order to evaluate the energy consumption relating it with the performance, we describe here the multi-core environment used for the measures. One Dell PowerEdge R710 node was adopted, containing 12 GB DDR3 memory, and 2 Intel Nehalem processors model Xeon E5530 2.4 GHz manufactured in 45 nm technology. They are quad-core processors with each core support up to 2 threads running in parallel using the simultaneous multi-threading (SMT) technique, each processor has a maximum amount of power TDP (thermal design power) of 80 Watts. The nodes use Linux Ubuntu 10.04 operating system (kernel 2.6.32-21-server #32-Ubuntu SMP) with GCC 4.4.3 and Open MPI 1.4a1r22335.

This Nehalem processor model controls power consumption using the Intel Turbo Boost technology [14], which enables higher performance through the availability of increased core frequency. This technology is the materialization of studies of dynamic power monitoring and control such as those presented on Section 2. This technique is activated under some configurations and workloads, and when the processor is operating below rated power, temperature, and current specification limits. A Nehalem processor can have just few cores activated, but all active cores in the processor will operate at the same frequency. Even at frequencies above the base operating frequency, all active cores will run at the same frequency and voltage. Although the knowledge of the system evaluated is important, our proposal is not to study some determined technology but evaluate the new processors techniques in order to show the impact of the parallel programming performance.

We measure the power consumption of nodes through the iDRAC6 (Integrated Dell Remote Access Card) [7] systems management hardware and software solution. It provides remote management capabilities, crashed system recovery, and power control functions. After enabled, the iDRAC provides access to system information and status of component. To obtain the system information, the free-ipmi software was installed. IPMI (Intelligent Platform Management Interface) specification is a standard which defines a set of common interfaces to a computer system enabling the system administrators to monitor the system health and manage the system. Using the IPMI to access the iDRAC we were able to monitor the power consumption of the entire system.

4.2 Parallel Programming Techniques

We verify the impact of efficient parallel programming techniques by four different ways to implement adaptive solutions to a target problem. To illustrate the approaches, we will show their use to solve the Fibonacci calculation problem. It returns the ith element in the Fibonacci sequence $(0, 1, 1, 2, 3, 5, 8, 13, ...)$ in which each element is the sum of the previous two.

4.2.1 Explicit Tasks with TBB

We chose TBB to implement an explicit task parallelism application because it has a task scheduler based on Work Stealing [4]. It maps tasks to native threads for the most efficient usage of the underlying hardware, aiming to minimize memory demands and cross-thread communication. Hence, the runtime takes responsibility of scheduling for locality and load balancing.

As TBB tasks are abstract user-defined classes, our tests implement an unit of work as a class derived from the abstract class *tbb::task*. An initial task has the entire problem N and it divides the work in d other tasks, thus creating d new task objects of $\lceil N/d \rceil$ size by the method *t*bb::task::spawn. This approach is similar to the recursive Divide and Conquer strategy, but its usage on a different programming strategy is straightforward.

Figure 1 shows an implementation of the Fibonacci with TBB. The user-defined *FibTask* derives from *task*. If the n is less than a threshold, the computation is made sequentially (locally on the object). Otherwise, two new dynamic tasks are generated to compute $n-1$ and $n-2$ respectively. The forker waits for children results (*spawn_and_wait_for_all*), sums them and returns.

```
class FibTask: public task {
public:
 const long n;
 long* const sum;
 FibTask( long n_, long* sum_ ) :
   n(n_), sum(sum_)
 {}
 task* execute() {
  if( n < CutOff ) {
    *sum = SerialFib(n);
  } else {
   long x, y;
   FibTask& a =
     *new( allocate_child() ) FibTask(n-1,&x);
   FibTask& b =
     *new( allocate_child() ) FibTask(n-2,&y);
   set_ref_count(3);
   spawn( b );
   spawn_and_wait_for_all(a);
   *sum = x+y;
  }
  return NULL;
 }
};
```

Figure 1. Fibonacci C++ code using TBB: the user-defined *FibTask* class that devives from *task*, dynamic tasks are generated until reaches the *CutOff*.

4.2.2 OpenMP Task Region

We also chose OpenMP to implement explicit task parallelism because it is a standard API for shared-memory platforms. An OpenMP task is a block of instructions, perhaps containing calls to another *task* block. Consequently, the parallelism can be unfolded during runtime. The created tasks synchronize using the *taskwait* construct, suspending the current task until all its children tasks have completed.

Figure 2 illustrates the explicit task parallelism in OpenMP. To each value of the n, two new tasks are dynamically generated (*task* construct) to compute $n-1$ and $n-2$ respectively. The *taskwait* ensures that the parent execution will only continue when the children results are available.

4.2.3 MPI Dynamic Processes

MPI tasks can be explicitly created as processes using the `MPI_Comm_spawn` primitive. The standard does not specify mapping schemes, and each MPI implementation can provide different approaches to distribute dynamic processes between the available processing elements [6, 22, 18]. The dependencies are explicit and specified by message-passing among processes without any restriction. Since MPI tasks can be dynamically created and synchronized, MPI-2 enables the development of programs with a structure closer to the explicit task paradigm, but for distributed-memory environments.

```
int fib(int n) {
   if (n < 2) return n;
   else {
      int x, y;
      #pragma omp task shared(x)
      {     x = fib(n-1); }
      #pragma omp task shared(y)
      {     y = fib(n-2); }
      #pragma omp taskwait
      return(x + y);
   }
}
```

Figure 2. Fibonacci C code using OpenMP: *task* directive creates tasks dynamically.

The MPI explicit task approach generates tasks dynamically until a threshold (stop condition). It increases the grain size of each new MPI process generated but the programmer has to establish the threshold level. Although the simplicity of a stop condition to control grain size, it has no guarantee of efficiency on different systems. Indeed, a large number of processes may affect performance as processors become oversubscribed. In this paper, we propose two approaches to optimize task creation: lazy and adaptive.

In Lazy, whenever tasks must be dynamically generated, the forker process (or the parent) will keep with some work to compute locally. To illustrate this, consider that two new tasks must be created, using the Lazy approach the parent will spawn a new process to compute one task and the other will compute locally. The local computation is implemented as a recursive call; furthermore, local and remote computations are synchronized through message-passing. Figure 3 illustrates the Lazy approach solving the Fibonacci problem. Notice that we take advantage of an unblocking received while the local computations are performed, and we suppress the `MPI_Comm_spawn` parameters for the sake of space.

In the Adaptive approach, the goal is to guide the dynamic task creation (MPI processes) to take into account the number of processing elements (processors or cores) available. This is achieved by adding a second stop condition in dynamic tasks creation, which checks the number of already running tasks against the number of processing elements. The dynamic creation stops when all processing

```
int mpi_fib( int n ) {
 if ( n < 2 )    return n;
 else {
  MPI_Comm_spawn("mpi_fib", n-1, &child );
  MPI_Irecv(&x, child, &req);
  y = mpi_fib( n-2 );
  MPI_Wait( req );
  return (x + y);
 }
}
```

Figure 3. Fibonacci C code using MPI - Lazy approach: in each recursion level, one task is spawned dynamically (`MPI_Comm_spawn`) while the other is computed locally.

elements have received a task, and from this moment on, all further tasks are computed locally, *i.e.* computed as recursive calls. Figure 4 shows the Adaptive approach in the Fibonacci calculation. Dynamic MPI tasks are generated until their number is less than the number of processing elements, after that, the computation is performed locally through recursive calls.

```
int mpi_fib( int n ) {
 if ( n < 2 )    return n;
 else {
  if ( curr_UE < nb_PE ){
   MPI_Comm_spawn("mpi_fib", n-1, &child[0] );
   MPI_Comm_spawn("mpi_fib", n-2, &child[1] );
   MPI_Recv( &x, child[0] );
   MPI_Recv( &y, child[1] );
  } else {
   x = mpi_fib( n-1 );
   y = mpi_fib( n-2 );
  }
  return x + y;
 }
}
```

Figure 4. Fibonacci C code using MPI - Adaptive approach: tasks are dynamically spawned until the current number of processes (*curr_UE*) is less than the number of processing elements (nb_PE).

4.3 Test Applications

The approaches described above were used to implement two test applications: a Merge sort sorting algorithm and a matrix multiplication, as described in the following.

4.3.1 Mergesort

Merge sort is a well-known recursive sorting algorithm. Our parallel implementation works as the sequential version, where recursive calls are replaced by tasks generation. It divides an initial input of size N in two smaller parts of sizes $\lceil N/2 \rceil$ for each task. Then, it recursively calls itself until a threshold from which the sequential algorithm is used. The conquer phase merges the results of two children and returns to the upper level. Although the OpenMP and TBB merge sort implementations use shared-memory arrays for input and output numbers, the MPI versions use message passing for send the input to each new task. The experiments used n random numbers as input.

4.3.2 Matrix Multiplication

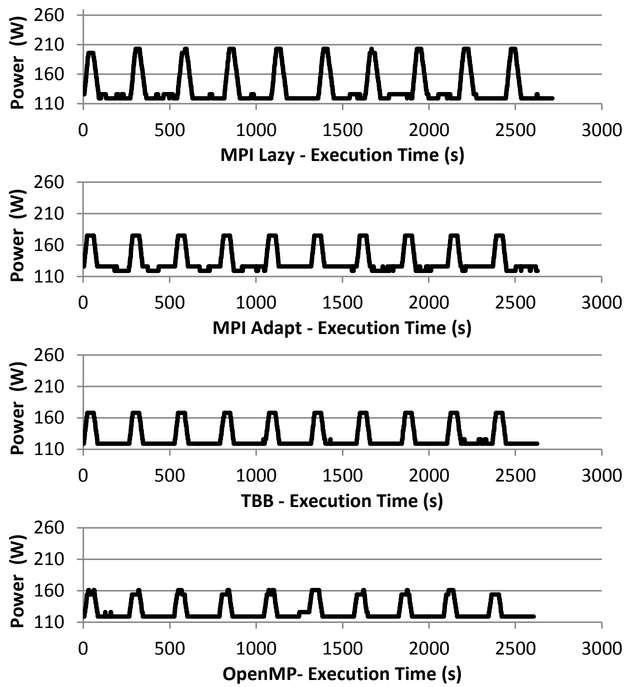
In matrix multiplication, the target problem is to compute the multiplication of two matrices A and B with $n \times n$ elements each, storing the results in a matrix $C_{n \times n}$. We implemented a recursive solution following the traditional algorithm, multiplying row by column. The input matrices A and B are partitioned in two halves each, in such a way that four new dynamic tasks are generated per level in the dividing phase. Each dynamic task will compute one quadrant of the $C_{n \times n}$. The recursive task creation stops when a threshold is reached, and then the sequential multiplication is computed. In the conquer phase, the results of the children are merged and sent to the upper level. The input matrices A and B are composed by random elements.

5 Experimental Results

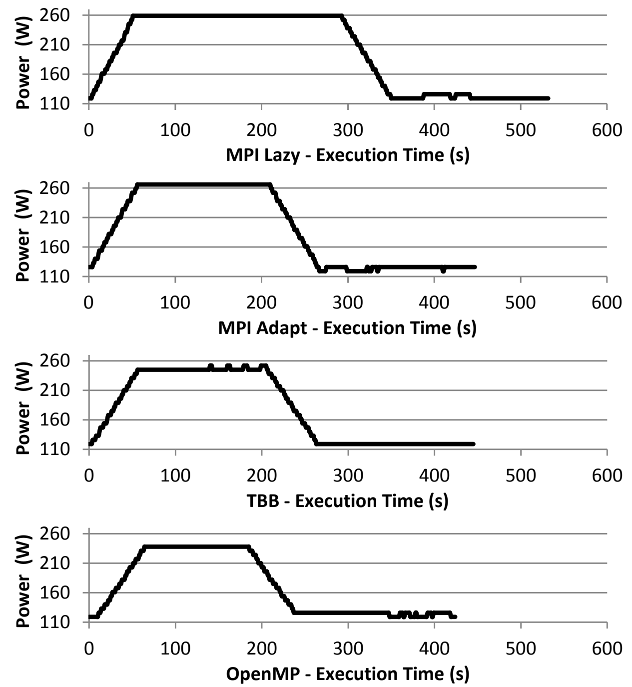
Based on the methodology presented in the previous section, the results for the matrix multiplication and merge sort execution are presented in this section. The input of matrix multiplication was two matrices with $8,192 \times 8,192$ random elements each, and the threshold was 512 elements. The merge sort sorted $500,000,000$ random elements with a threshold of $1,000,000$.

Figure 5 shows the power consumption along the execution time for the implementations of matrix multiplication and merge sort, as well as the target approaches. We organize the power consumption results by sets of 10 executions in two different scenarios: $1by1$ that separates each execution in an interval of 4 minutes, aiming to verify the peak of consumption achieved by each application; and $sequentially$ in which all 10 executions are executed one after another, and only after the last execution a 4 minutes wait is performed. All our executions were in parallel using 16 cores.

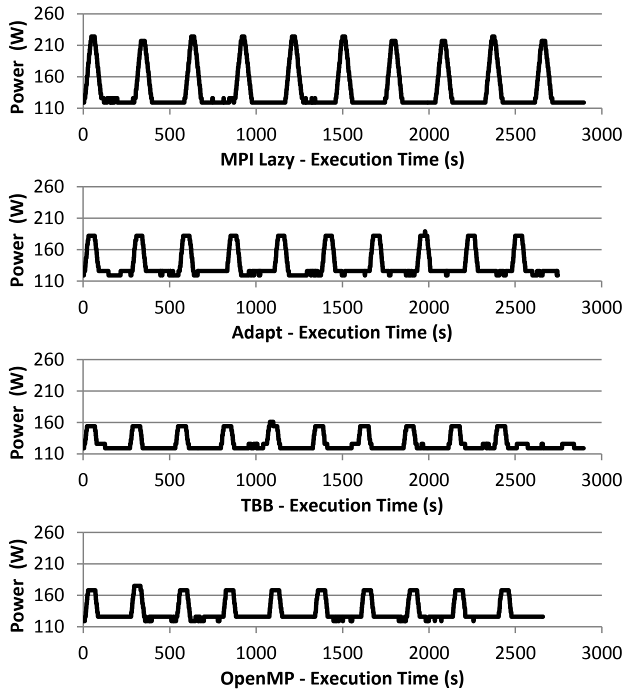
The plots show a high power consumption variation, which is clear with the execution $1by1$ showing the processor rising its power consumption at each execution. This



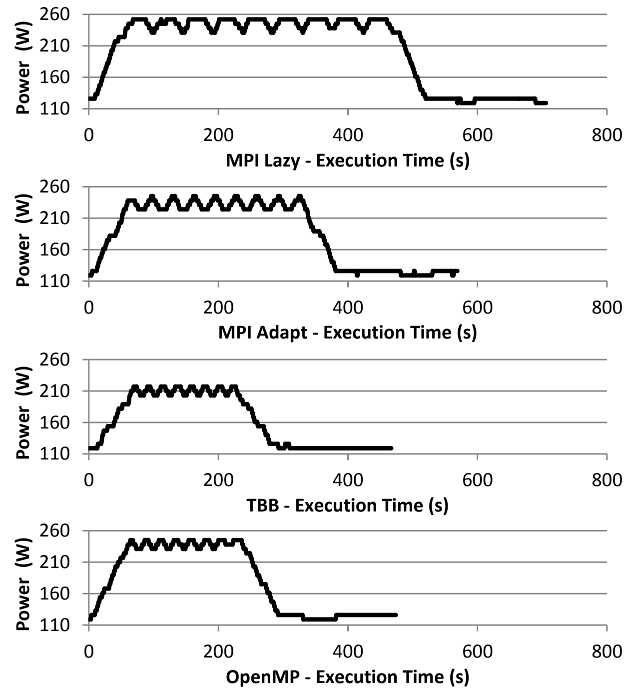
(a) Matrix multiplication executing $1by1$, with 4 minutes on idle interval between the executions.



(b) Matrix multiplication executing $sequentially$ 10 times with 4 minutes on idle by the end of the executions.



(c) Merge sort executing $1by1$, with 4 minutes on idle interval between the executions.



(d) Merge sort executing $sequentially$ 10 times with 4 minutes on idle by the end of the executions.

Figure 5. Matrix multiplication and merge sort executions using 1 processing node with 16 cores.

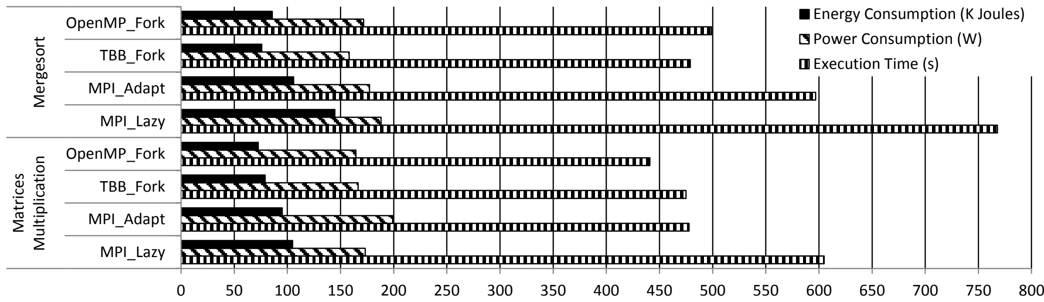


Figure 6. Total of energy consumption and execution time for the different algorithms and implementations.

variation demonstrates the relevance of architectural techniques to reduce the power and energy consumption, in this case the Intel Power Boost technology. On *sequentially* execution, the merge sort showed a power consumption variation of every new execution. This is due to application characteristic which leads to the load unbalance occurred at the beginning and most at the end of the execution, where just a few processes or threads continue executing.

It is also possible to note variations between different implementations, where the matrix multiplication had its lower peak of power consumption running the TBB (238 W), OpenMP (252 W), MPI Adaptive (266 W), MPI Lazy (259 W) implementations respectively. The merge sort had its lower consumption executing the OpenMP (217 W), TBB (245 W), MPI Adaptive (245 W), and MPI Lazy (252 W) implementation respectively.

The high power consumption for the applications implemented with MPI can be explained in part due to higher cost from the MPI processes compared with the OpenMP or TBB threads, and part by the network usage. In the case of network usage, even using the shared memory implementation of the OpenMPI, the dynamic process creation does not support shared memory; thus, the new processes created dynamically use TCP message communication, increasing the overall power consumption.

Other explanation to the high power consumption peak for the MPI Lazy is due to its high number of process created in order to parallelize the problem, showing the importance of picking the right algorithm for each problem.

The power variation between each execution can indicate that performance evaluation using these new architectures may be affected by these automatic variations on the architecture, adding more noise to the measures. Although this is an interesting topic, we will leave this performance evaluation topic aside for a future work.

The summary results for matrix multiplication and merge sort running 10 times *sequentially* are shown on Figure 6. It has the total energy consumption for the different implementations together with the average power consumption and the execution time. Within the plots it is possible to compare the execution time among the experiments, in which OpenMP and TBB achieved the best performance thanks to their scheduling optimizations. These two implementations also had the lower total energy consumption.

Although this relation between lower execution time and lower energy consumption may seem obvious, the results presents that sometimes the system can behave in a different way. As in the case of matrix multiplication with MPI Adaptive, it is possible to see the overhead caused in part by the higher cost from the MPI process and the network communication, leading the MPI Adaptive to spent 9.1% more energy just to execute 0.6% more time than the TBB. This difference in the rate which energy has been consumed is presented by the Power Consumption metric in the plot.

It is important to explain that we choose to evaluate the message passing interface on a single multi-core node because it is a simplification of a cluster scenario, whose TBB and OpenMP will not fit on this distributed-memory architecture. However, we have shown that MPI implementations brings an extra power and energy consumption to the multi-core node, and it indicates that a hybrid programming model shall be used in order to achieve both high performance and low power/energy consumption.

6 Conclusion

Concerning the large-scale clusters of machines, all the energy saved is beneficial for the entire cooling system which will need to dissipate less heat, for the total power consumption, and to the environment. Thus, it is clear that computer architects have key role on the power consumption. However, programmers can also help to reduce the total energy consumption with more intelligent algorithms on parallel applications.

Due to the topic relevance, our measures showed the impact of parallel applications on the current processors energy consumption. The results presented up to 47% of energy saved for the merge sort algorithm using the TBB implementation. For the matrix multiplication, gains of 30% were achieved using the OpenMP implementation.

Moreover, we presented that depending on the implementation, the system will vary the power consumption, leading to different energy consumptions. Thus, the expectations that longer execution time leads to a more energy consumed can be wrong for some cases, revealing that programmers need to start looking both performance and power consumption to best application choose.

Moreover, future performance evaluations on power aware multi-core and many-cores shall considers the warm-up time in order to obtain reliable performance results.

Future works focus on the both architectural and programming improvements in order to achieve lower power consumption by the architecture and lower energy consumption by the parallel applications. Evaluations about the new power save technologies on the performance evaluations are also considered as future work. Moreover, a more extensive evaluation on homogeneous and heterogeneous cluster of machines is important, in order to analyze the tendency of hybrid programming for these scenarios.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *ACM SIGARCH Computer Architecture News*, 2000.

[2] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *Transactions on Parallel and Distributed Systems*, 2009.

[3] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, and P. Kundu. Integration challenges and tradeoffs for tera-scale architectures. *Intel Technology Journal*, 11, 2007.

[4] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 1998.

[5] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 1999.

[6] M. C. Cera, G. P. Pezzi, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Improving the Dynamic Creation of Processes in MPI-2. *European PVM/MPI Users Group Meeting*, 2006.

[7] Dell Inc. Integrated delltm remote access controller 6 (idrac6) version 1.0. *Dell User Guide*, 2009.

[8] T. V. T. Duy, Y. Sato, and Y. Inoguchi. Performance evaluation of a green scheduling algorithm for energy savings in cloud computing. *Int. Symposium on Parallel Distributed Processing, Workshops*, 2010.

[9] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. *Int. Workshop on Parallel Symbolic Computation*, 2007.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994.

[11] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2 Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1999.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, USA, 4th edition, 2007.

[13] W. Huang, M. Stant, K. Sankaranarayanan, R. Ribando, and K. Skadron. Many-core design from a thermal perspective. *Design Automation Conference*, 2008.

[14] Intel Corporation. Intel turbo boost technology in intel core microarchitecture (nehalem) based processors. *Intel White Paper*, November 2008.

[15] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. *Int. Symp. on Microarchitecture, MICRO-39*, 2006.

[16] B. Khargharia, S. Hariri, and M. Yousif. Autonomic power and performance management for computing systems. *Cluster Computing*, 2008.

[17] C. E. Leiserson. The Cilk++ concurrency platform. *Annual Design Automation Conference*, 2009.

[18] J. V. F. Lima and N. Maillard. Online mapping of MPI-2 dynamic tasks to processes and threads. *Int. Journal of High Performance Systems Architecture*, 2009.

[19] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Computing*. Software Patterns Series. Addison Wesley, 2004.

[20] V. Natarajan, A. Deshpande, S. Solanki, and A. Chandrasekhar. Thermal and power challenges in high performance systems. 2008.

[21] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *Int. Symp. on Architectural Support for Programming Languages and Operating Systems*, 1996.

[22] G. P. Pezzi, M. C. Cera, E. Mathias, N. Maillard, and P. O. A. Navaux. On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing. *Int. Symp. on Computer Architecture and High Performance Computing*, 2007.

[23] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly & Associates, Inc., Sebastopol, USA, 2007.

[24] J. Sartori and R. Kumar. Three scalable approaches to improving many-core throughput for a given peak power budget. *Urbana*, 51:61801.

[25] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. 1995.

[26] W. Stallings. *Computer Organization and Architecture: Designing for Performance*. Prentice Hall, 4th edition, 1996.

[27] T. Ungerer, B. Robic, and J. Silc. Multithreaded processors. *British Computer Society*, 2002.

[28] D. Wang and S. Teradata. Meeting green computing challenges. *Int. Symp. on High Density packaging and Microsystem Integration. (HDP)*, 2007.

[29] D. Woo and H. Lee. Extending Amdahl's Law for energy-efficient computing in the many-core era. *Computer*, 2008.