Energy-Aware Scheduling of Parallel Programs*

Alexandre Carissimi, Claudio F. R. Geyer, Nicolas Maillard, Philippe O. A. Navaux
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{asc, geyer, nicolas, navaux}@inf.ufrgs.br

Gerson G. H. Cavalheiro, Maurício L. Pilla, Adenauer C. Yamin
Universidade Federal de Pelotas
Pelotas, Brazil
{gerson.cavalheiro, pilla, adenauer}@ufpel.edu.br

Andréa S. Charão, Benhur Stein
Universidade Federal de Santa Maria
Santa Maria, Brazil
{andrea, benhur}@inf.ufsm.br

César A. F. De Rose, Luis G. L. Fernandes, Tiago C. Ferreto, Avelino Zorzo
PUCRS
Porto Alegre, Brazil
{derose, gustavo, ferreto, zorzo}@inf.pucrs.br

Abstract

In this paper, we discuss the impact on the energy consumption of different approaches to schedule parallel application on shared-memory and distributed architectures. Basically, a trade-off must be determined between the use of more CPUs, which enables to shorten the runtime (and thus save energy), and the extra energetical cost of using these extra resources; besides, when the parallel program is running on a single, multicore CPU, the energy consumption can be further reduced by tuning the frequency and the voltage of the chip. We show how these adaptations of the parallel and sequential characteristics of both the applications and the hardware can be performed on a variety of parallel platforms, ranging from a single multicore chip to a dynamic cluster.

*This work is part of the project "Green Grid", supported by a PRONEX FAPERGS/CNPq grant.

1. Introduction

Concerns regarding environmental issues are steadily growing in the IT community. Greenhouse gas emissions and contamination due to untreated waste are the most notable symptoms. A simple PC has a footprint of about a ton of carbon dioxide during its lifetime [10].

Engineers have started to worry about power dissipation in the mainline processors to solve issues such as battery life and heat exhaustion in laptop computers. However, techniques such as frequency modulation and V_{core} adjustment have found their way to desktops and servers, making it possible to determine tradeoffs between energy consumption and runtime.

These techniques can be applied to control the energy costs of concurrent applications. When it comes to running a parallel program on a distributed memory platform (cluster of p CPUs), the energetical consumption is basically multiplied by p. Therefore, the parallel execution will be energetically worthwhile only if the speed-up is linear in function of p, in order to compensate the energy overhead. For most applications, this will be true only up to a certain number of CPUs, which will provide the upper limit of the parallelism

that can be used while maintaining a good ratio energy/speedup.

This paper discusses these energetical costs associated to parallel applications and architectures. It presents some results on the efficient use of parallel resources from an energy consumption point of view, and the measurements that have been obtained by experience on real applications and benchmarks.

The rest of this article is organized as follows: First, Sec. 2 provides the necessary background about the consumption of energy in a processor, and in a cluster of CPUs. Then, Sec. 3 shows the kind of control of the energetic overhead that can be obtained for parallel applications running on a multicore chip, which enables a fine control of the cost, and a finer-grained schedule based on threads. Afterwards, we switch to the case of clusters (Sec. 4) to treat the issues related to scheduling parallel applications: with more *a-priori* information on the resources and their availability, a batch-scheduler can obtain good usage of the cluster. When the parallel application is dynamic (*i.e.* creates tasks during its execution), the schedule can be adapted at each moment to use exactly the right number of resources; this is the subject of Sec. 5. Finally, the last section provides insights of future developments.

2. Power and Energy in Clusters of CPUs

2.1. Power and Energy in a Single Processor

Energy is the ability to do some work, being measured in $Joules$. In digital circuits, energy is provided by electrical sources. *Power* can be defined as the time rate of energy, being measured in $Watts$. A $Watt$ is a $Joule$ per second.

For previous generations of CMOS designs, the main component of power was gate switching. Considering that C_{sc} is the switched capacitancy, V_{dd} is the voltage at which the circuit is operating, f is the operation frequency, $I_{Leakage}$ is the leakage current, N is the number of cycles required for the program, then the energy E required for a program can be defined as [8]:

$$E_{op} = C \times {V_{dd}}^2 \times f \times (\frac{N}{f}) + I_{Leakage} \times V_{dd} \times (\frac{N}{f})$$

Hence, energy consumption of a processor may be reduced by: *(i)* reducing the total number of operations performed (N); *(ii)* reducing capacitancy; and *(iii)* by reducing V_{dd} and frequency f. Although depending on the number of executed instructions, the number of switching operations is not derived directly from it due to the complexity of modern chip designs. Many works have been developed focusing to reduce capacitance of CMOS designs, but these are outside the scope of this paper. Energy spent due to current leakage $I_{Leakage}$ is becoming increasingly important in modern CMOS designs [11]. As the gate length reduces, more current pass through it even when the transistor is not switched on. Finally, reducing V_{dd} and f is a possibility open to users of modern microprocessors.

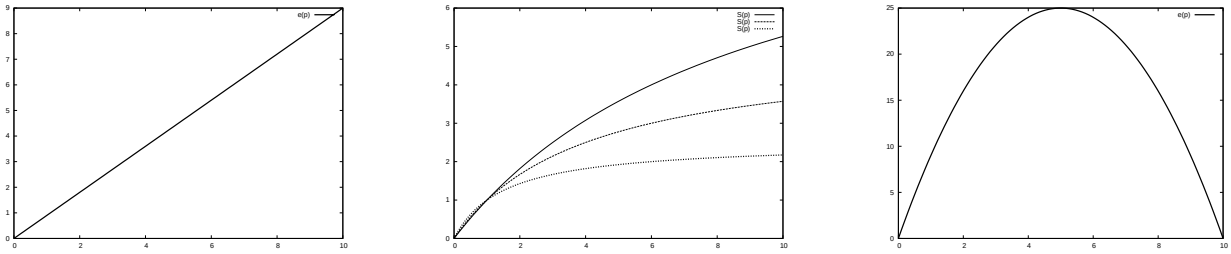
Reducing only V_{dd} and not the operation frequency f is not an option. The propagation delay of a gate is inversely proportional to the voltage [8]. Thus, at lower voltages more time is required for the propagation of a signal through the gate. Reducing f makes execution slower, as more time is required for each operation. Decreased operation frequencies do not imply in less energy, and it can even increase in modern processors where $I_{Leakage}$ plays an important role. However, when f is reduced, Vdd can be reduced as well, hence achieving energy savings.

2.2. Energy Consumption of a Cluster of p CPUs

In order to achieve higher scale of parallelism, the parallel application must be run on a cluster of $p > 1$ CPUs, interconnected by a network. A parallel application that runs during T seconds on a distributed platform constituted of p resources has an energetical cost $e(p)$ that is proportional to $p \times T$. Thus, in order to save energy, two approaches can be used: reducing the runtime T, or reducing p. However, T depends on p. Normally, a scalable parallel application has a speedup $S(p) = K/T(p)$, and T decreases when p increases. All the problem of an energy-aware parallel execution is therefore to balance the gains in runtime due to a high parallelism with the energetic losses due to too much parallelism.

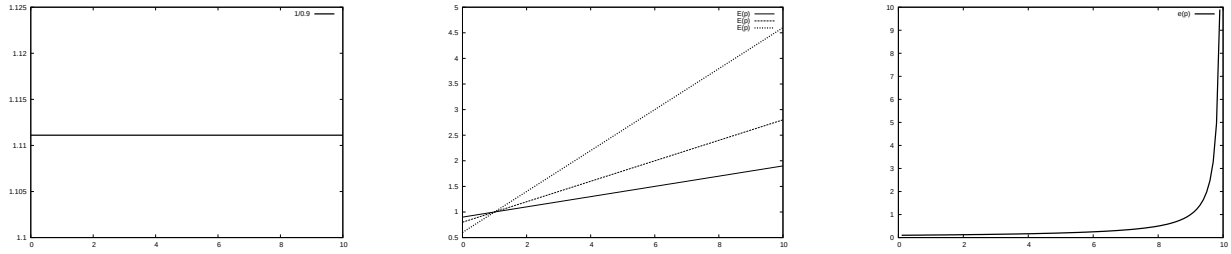
As an illustration, consider the three typical cases for the speed-up $S(p)$ detailed in Fig. 1. The corresponding energy cost functions $e(p) = pK/S(p)$ (where K is a constant factor which is the energy cost of the sequential execution) are plotted in Fig. 2. Theses cases only aim at providing a trend, so that the exact values do not matter (specific cases are discussed in depth in the next sections).

As can be seen, for a linear speedup, the energy is of course constant with regard to p (Fig. 2a). Now considering the speed-up of a parallel program limited by Amdahl's law [6], with for instance 90% of its runtime totally parallel, one gets $S(p) = 1/(0.1 + 0.9/p)$, and the energy $p/S(p)$ has a linearly increasing behavior (See Fig. 2b for different energy curves, depending on the parallel percentage of the program), with slope equal to the sequential percentage of the program —



(a) Linear sub-optimal speedup (b) Saturated sub-optimal speedup (Amdahl law) (c) Limited sub-optimal speedup

Figure 1: 3 typical speed-up functions: linear, Amdahl, and decreasing.



(a) Energy (b) Energy (c) Energy

Figure 2: The 3 energy functions, for each kind of speedup (linear, Amdahl, and decreasing).

i.e. the more parallel the program, the less the energy increases. In the last case (Fig. 2c), where the speedup decreases from a given number of resources, the energy cost shows an exponential increase, which suggests that the parallelism in a cluster will be interesting up to, but not exceeding, a given limit.

Thus, the energy cost of a parallel application can be reduced either at the level of a multicore computer for a concurrent application (typically multithread), or by limiting the use of distinct nodes of a cluster when the speed-up does not compensate the energy cost. The following sections detail these two approaches.

3. Scheduling Threads in Multicore CPUs

Thread scheduling may be done in the application level, in the system level, or both. In the application-level scheduling, the objective of scheduling is to enhance a program-specific index and not the system performance. Heuristics for application-level scheduling take in account knowledge of the internal structure of the program that would not be available to system-level scheduling.

Processor affinity allows associating a given thread to a core, hinting the scheduler where this thread should be run. Pinning a group of threads to a processor may enhance performance by better exploiting reference locality, avoiding migration of data and code through caches. In most cases, setting processor affinity is a task that requires commitment of the programmer and full knowledge of both the program and the system where it will run, and it may not improve performance in all cases [7].

In Section 2, the relationship between power and voltage was discussed. Modern processors are able to change their clocks, but first V_{dd} must be set. Changes in operation frequencies are not immediate, requiring cycles where the processor stops processing before the new clock can be set [14]. A simplification of the formulas described in Section 2 can be expressed as:

$$C_{index} = \frac{f \times \alpha}{t} \tag{1}$$

Where C_{index} is the consumption index, f is the operation frequency, α is the percentage of processor

use, and t is time.

Nowadays, general solutions for the problem of energy consumption are mostly based on system-level regulation of frequency and voltage, using *governors* to dynamically adjust them to the requirements of the entire system in a given time. With nowadays technologies, it is possible to set clock rates for each individual core. Hence, parallel programs must have strategies to deal with heterogeneous processors [1]. Both processor affinity and frequency setting are exposed to users.

To better understand the impact of clock rate and affinity in both performance and energy consumption, a case study based on the Josephus problem [5] was developed. This problem consists in the random generation of a circular list of tasks with non-uniform loads. Each task was executed by a thread. The experiments were run in a Core 2 Duo computer, and repeated 20 times, producing similar results. Loads were generated so that 65% of tasks would present a load 40% or less of the total load.

Figure 3 shows the Consumption Index traces for three different settings. The first one uses frequency governing, i.e., cores with threads will run in maximum frequency, while idle cores will reduce their clock rates. Besides, threads are bound to processors (affinity) according to their loads. The second trace shows a run with neither frequency governing nor processor affinity. The last trace was done with frequency set on minimum, regardless of load, and without affinity. The vertical axis presents the consumption index C_{index}, while the horizontal axis shows time in seconds. Without affinity nor frequency governing, energy consumption is 1.41× larger than with affinity and frequency governing, and 1.64× larger than with the minimal clock rate. The use of affinity and frequency governing allowed for a reduction of 29.1% in energy consumption for an increase of 8.3% in time, while running with the slow clock rate reduced energy consumption by 39.3% but increased times in 75.3%.

A second case study applying the Monte Carlo method for numeric integration was developed. The graph representing tasks and threads was made unbalanced on purpose, i.e., threads in the left most part of the graph execute over larger intervals of the problem. This application was executed with and without affinity and application-level scheduling in a Core 2 Quad computer. For the first case, the average time was 10.02 s, while for the second one time was 6.173 s, a speedup of 1.62 for the same architecture. Besides, 63.2% less energy was required. Scheduling with more priority threads that have more load is one factor that helped performance. As the Monte Carlo method is applied over a large matrix, processor affinity increases

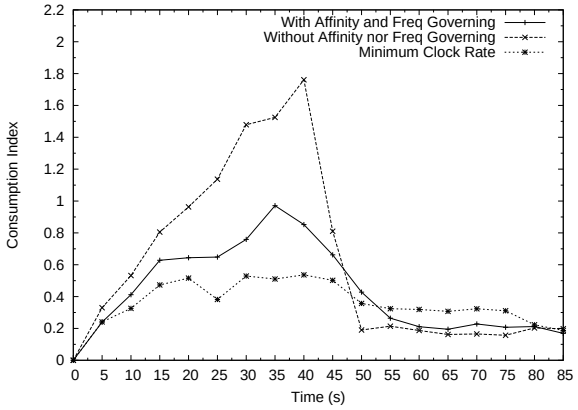


Figure 3: Time versus consumption index for the Josephus problem.

data locality on caches. This controlled experiment shows that it is possible to improve performance while reducing frequencies using heuristics based from the application knowledge.

Thus, knowing the structure of a program and using this knowledge in the scheduling may impact in both performance and energy consumption of a multicore CPU. The next section treats the case of a distributed memory cluster.

4. Scheduling Parallel Applications in Clusters

In this section, we discuss the energetic cost of a typical parallel applications run on a cluster : the Lattice Boltzmann Method. As presented in Sec. 2.2, the speed-up curve can be analyzed in order to determine if it compensates for the energetical cost.

An important technique for the mesoscopic simulation of fluid dynamics technique is the Lattice Boltzmann Method (LBM) [13, 12]. In [13], a parallel version is presented for 2D and 3D simulations, based on block partitioning. On medium-sized clusters, the speed-ups have shown to be good considering the kind of application. Fig. 4 presents the speed-ups obtained on a lattice of $N = 128 \times 128 \times 128$ points, where the fluid flowing in a 3D pipe with obstacles has been simulated. All measurements have been made on the cluster Labtec, at the Federal University of Rio Grande do Sul. This cluster is composed by 20 dual nodes Pentium III, 1.266 Ghz with 512 MB of RAM memory, interconnected by a *Fast Ethernet* network.

The speed-up is provided in Fig. 4 for various implementations to illustrate how the right granularity improves the speed-up of a parallel program. In the

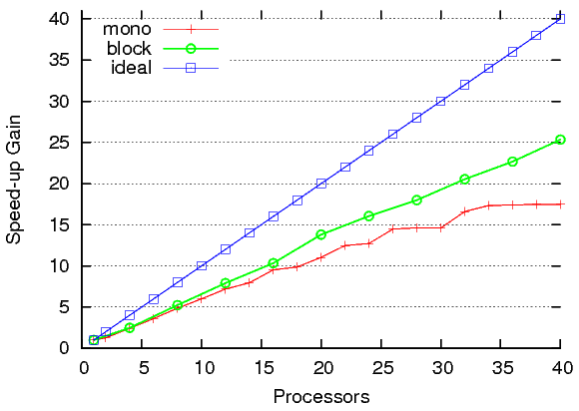


Figure 4: Speed-up for mono and blocked partitioning of the 3D Lattice-Boltzmann method (from [13]).

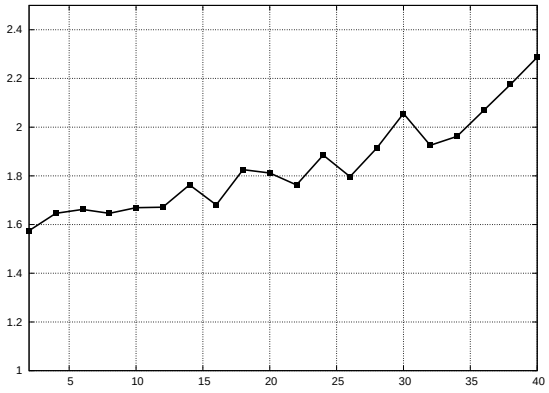


Figure 5: Energy cost of the blocked partitioned 3D Lattice-Boltzmann method (in relation to the energy cost of the sequential execution).

case of this application, the granularity is controlled by a parameter which specifies the geometry of each part of the lattice that a given processor simulates.

Evaluating the energy cost as a function of the number of processors, based on the best speed-up curve, one gets Fig. 5. What is interesting in this case is that the energy cost is, of course, increasing when more parallelism is used. This reflects the non-optimal speed-up. However, the energy cost is not increasing fast. For $p = 20$ CPUs, the cost is only 1.8 higher than the sequential execution. Even with 20 to 25 CPUs, the energy cost is lower than twice the cost of the sequential execution — and the runtime is divided by almost 25.

However, not all the applications can be parallelized in such an efficient way. Some of them simply do not provide parallelism during part of their execution to fully and constantly use the computing resources of a cluster. For such application, a dynamic scheduling can be used, as shown in the next section.

5. Scheduling Dynamic MPI Applications

Malleable applications can adapt their usage of parallel resources during their execution. This is important for many applications, which do not constantly have the same degree of parallelism. Typically, Branch & Bound algorithms have to explore a tree of possible solutions, and they have to develop new branches, and cut some of them, during the execution. If each branch is run as a separate task, the number of useful CPUs will vary at runtime.

A traditional approach in a cluster of p CPUs would be to decide and run one MPI process per CPU. Each one of them would start exploring the tree from its root, until at least p branches have been created. Then, each process only explores its own subset of branches, thus distributing the computation. This implementation relies on static tasks, since they are all scheduled to a process at the start of the execution. However, using the dynamic creation of MPI processes, one can use a dynamic scheduling of the tasks: each process still starts exploring one sub-tree, however it maintains a stack of branches to be explored, while it works on one of them. Whenever one of the processes does not have any branch anymore in its stack, it sends a steal request to one of the others, which sends it back part of the branches in its stack. If a steal attempt reaches a process that also have an empty stack, the stealer attempts to steal from an other victim. This dynamic load balancing algorithm, called Work Stealing [2], leads to close to optimal speed-up. Notice that, due to the inherent irregular behavior of the Knapsack problems, an optimal schedule cannot be statically determined: good performance can only be achieved through a dynamic treatment.

Fig. 6 shows the compared speed-ups of the static and dynamic versions of classical NP-Complete problem: a Knapsack with 2.500 items [5]. All the executions have been performed on the same cluster as in the previous section. As it can be seen, the speed-up of the dynamic version is linear and almost optimal, whereas the static implementation suffers from bad load balance and has a saturated speed-up.

Fig. 7 provides insight on the energy cost of both versions. As explained in the introduction, the energy cost of the static version presents a linear increase

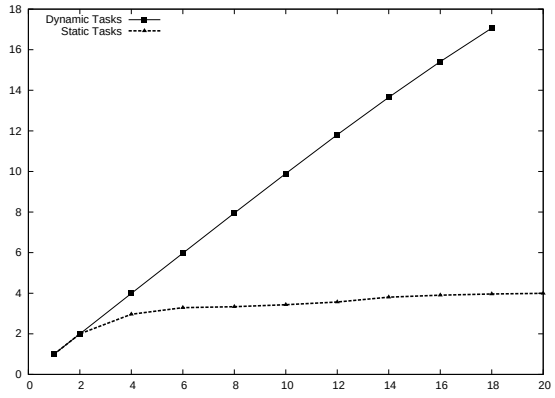


Figure 6: Compared speed-ups of the static and dynamic MPI version of Knapsack. Taken from [9].

when more resources are used. Adding the bad speed-up and this increase, one can conclude that the more parallelism the static version uses, the more power it consumes, without accelerating the program. Clearly, from Fig. 6, there is almost no more acceleration with more than 15 CPUs.

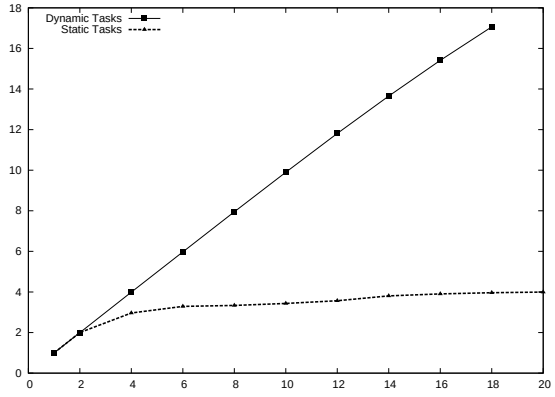


Figure 7: Compared energy costs of the static and dynamic MPI version of Knapsack (in relation to the energy cost of the sequential execution).

On the other hand, Fig. 7 shows that the energy cost of the dynamic version remains almost constant up to 20 CPUs (albeit a slight increase for more than 15 CPUs). In this case, the almost linear speedup compensates for the linear increasing energy cost. With such a dynamic scheduling, the MPI Knapsack version is energy-safe.

In [4], the authors have explored the use of MPI-2 dynamic processes to provide malleability in MPI programs. MPI processes are spawned when new resources become available, and when tasks are ready to be run. There is a scheduler helping in this issue, which is able to identify the changes in the amount of resources available and indicate which resources are able to receive new processes [3]. It provides two policies: Round-Robin and workload-based. The scheduler has been integrated to a batch-scheduler in order to be notified when new resources are available. With this more general work, which can be used for MPI programs that are not restricted to Brand & Bound computations, similar results in terms of speed-up and energy are to be expected.

6. Conclusion

This paper has presented a series of studies of the different, representative, parallel applications that can be run on a range of platforms that go from the single, multicore chip, up to a cluster with dynamic, on-demand allocation of CPUs to the application. In each case, a discussion of the impact of the scheduling techniques on the speed-up and energy cost was presented. The bottom line is that the energy cost of running a parallel application on parallel hardware can be maintained reasonably constant if the speedup is close to linear, in order to compensate the linear increase of CPUs (or cores) that are used; when coupled to efficient, low-level mechanisms to adapt the frequency of the CPUs, a fine balance between runtime gain and controlled energy overhead can be achieved.

One of the actual key points in order to guarantee the energy efficiency is the ability to adapt both the parallelism of the program, and the activity of the resources, to the behavior of the application at runtime. From the programmer point of view, this means that a dynamic schedule must be used, and this implies that the programming framework be adapted to this end. Typically, a static, parallel MPI program running one process by CPU will not have the necessary malleability.

The open problem that remains is to automize the different levels of control that we have highlighted in this paper: frequency rate and affinity at the chip level, on-demand allocation of part of the nodes of a cluster, limitation of this number based on the energy *vs.* speed-up criteria. This clearly leads to a multi-criteria optimization problem, without having all the necessary prior information at the start of the execution. In future work, we intend to apply heuristics to advance on

this point.

An other promising platform for parallel computing is a cloud of volunteer resources. Through the use of idle resources of Desktop Grid nodes, this may generate a significant power consumption saving, compared to the use of dedicated clusters in addition to users workstations. In this second case, both groups of machines (cluster servers and workstations) consume power during busy and idle periods. In contrast, the use of CPU cycles of idle nodes, adds to the power consumption only the extra CPU work in these machines. Improving the efficiency of task scheduling, and making a good use of the availability windows in such a volunteer environment, would thus be very important to achieve reductions in energy consumption.

References

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, 2005.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[3] M. Cera, G. Pezzi, E. Mathias, N. Maillard, and P. Navaux. Improving the dynamic creation of processes in mpi-2. In *13th European PVMMPI Users Group Meeting*, volume 4192/2006 of *LNCS*, pages 247–255, Bonn, Germany, 2006.

[4] M. C. Cera, Y. Georgiou, O. Richard, N. Maillard, and P. O. A. Navaux. Supporting malleability in parallel architectures with dynamic cpusets and dynamic mpi. In *Distibuted Computing and Networking: 11th International Conference, ICDCN 2010*, pages 242–257. Springer Verlag, 2010.

[5] H. T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[6] C. A. F. De Rose and P. O. A. Navaux. *Arquiteturas Paralelas*. Editora Sagra-Luzzato, Série Livros Didticos, número 15, 2004.

[7] A. Foong and D. Newell. Improved Linux* scaling: User-directed processor affinity. Available at `http://softwarecommunity.intel.articles/eng/1781.htm`, October 2008.

[8] G. Konduri, J. Goodman, and A. Chandrakasan. Energy efficient software through dynamic voltage scheduling. In *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on*, volume 1, pages 358 –361 vol.1, jul 1999.

[9] S. D. K. Mór and N. Maillard. Melhorando o desempenho de algoritmos do tipo branch & bound em mpi via escalonador com roubo aleatrio de tarefas. In SBC, editor, *Anais do X Simpsio em Sistemas Computacionais, WSCAD-SSC 2009*, pages 11–18, So Paulo, BRA, Outubro 2009. SBC.

[10] S. Murugesan. Harnessing green it: Principles and practices. *IT Professional*, 10(1):24 –33, jan.-feb. 2008.

[11] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91:305–327, February 2003.

[12] C. Schepke and N. Maillard. Performance improvement of the parallel lattice boltzmann method. In *19th International Symposium on Computer Architecture and High Performance Computing*, pages 71–78. IEEE Computer Society, 2007.

[13] C. Schepke, P. O. A. Navaux, and N. Maillard. Parallel lattice boltzmann method with blocked partitioning. *International Journal of Parallel Programming*, 37:593–611, 2009.

[14] S. Uhrig and T. Ungerer. Energy management for embedded multithreaded processors with integrated edf scheduling. In *High Perf. Computing for Computational Science. Proc. of the 18th Intl. Conf. on Arch. of Computing Systems, System Aspects in Organic and Pervasive Computing (ARCS)*, pages 1–17, March 2005.