Invited paper

Teaching Concurrency and Parallelism Concepts with CMRE

Laura De Giusti, Fabiana Leibovich, Franco Chichizola, Marcelo Naiouf

Institute of Research in Computer Science III-LIDI (III-LIDI) – School of Computer Science –UNLP Argentina

{ldgiusti, fleibovich, francoch, mnaiouf}@lidi.info.unlp.edu.ar

Abstract

Possible methodologies for teaching the concepts of processor heterogeneity and its impact on speedup and efficiency in a parallel system are discussed, as well as energy efficiency of parallel algorithms based on processor power.

CMRE (Concurrent Multi Robot Environment) is expanded to be able to consider different virtual clocks in each robot (processor), as well as the cost – both in relation to time and energy consumption – of the operations carried out by the robots (Move, Put Down / Pick Up / Message / Inform).

In this paper, we analyze some examples to show how concepts are introduced to students.

Keywords: Concurrency, Parallelism, Parallel Algorithms, Heterogeneous Processors, Energy Consumption.

1. Introduction

Concurrency has been a central issue in the development of Computer Science, and the mechanisms used to express concurrent processes that cooperate and compete for resources have been in the core curriculum of Computer Science studies since the seventies, in particular after the foundational works of Hoare, Dijkstra and Hansen [1,2,3]. The concepts were traditionally taught assuming the availability of a single processor that could partially exploit the concurrency offered by the algorithm, based on the available physical architecture (even with specific hardware such as co-processors, peripheral controllers, or vector schemes that would replicate arithmetic-logical units).

Parallelism, understood as "real concurrency" in which multiple processors can operate simultaneously on multiple control threads at the same point in time, was for many years a possibility that was limited by available hardware technology [4]. Classic Computer Science curricula [5,6,7] included the concepts of concurrency in various areas (Languages, Paradigms, Operating Systems), but parallelism was almost entirely omitted, except to present the concepts of distributed systems.

Current processor architectures, which integrate multiple cores within one physical processor, have had a notorious impact, resulting in a reformulation of a processor's "base model". This has resulted in the replacement of the "Von Neuman machine" concept, which has just one control thread, with a scheme that integrates multiple cores, each with one or more control threads and several memory levels that are accessible in a differentiated manner [8,9].

At the same time, changes in technology have produced an evolution of the major topics in Computer Science, mainly due to the new applications being developed from having access to more powerful and less expensive architectures and communications networks [10]. For this reason, international curricular recommendations mention the need to include the topics of concurrence and parallelism from the early stages of student education, since all architectures and real systems with which they will work in the future will be essentially parallel [11]. However, parallel programming (and the essential concepts of concurrency) is more complex for students who are starting their studies, and new strategies are required to teach the topic.

Given the stimuli to which students are exposed from an early age, be it through video games, computers, mobile phones, tablets, or any other electronic device, the use of interactive tools to teach core concepts to students in a CS1 course [11,12,13] has become essential [14]. In this sense, the possibility to take the initial steps in the world of programming through a graphic and interactive environment allows reducing the gap that traditionally existed between abstraction and the possibility of seeing a graphic representation of how the concepts being learned are applied in an environment that is conceptually similar to those used in everyday life [10,15].

CMRE is a graphic environment that has a set of robots that move within a city, and it has allowed teaching the basic concepts of concurrency and parallelism in a beginners course in Computer Science. However, there are advanced features (such as heterogeneity, time cost, load balancing, energy consumption, etc) whose addition is of interest to use the environment in non-beginner courses as well. In this paper, we present the extension of

CMRE to include the concepts mentioned above.

The article is structured as follows: Section 2 describes CMRE in its current version. Section 3 discusses the issue of heterogeneity, while Section 4 focuses on energy consumption. Section 5 presents conclusions and future lines of work.

2. Current Version of CMRE

The main features of CMRE can be summarized as follows [8,16]:

- There are multiple processors (robots) that carry out tasks and that can co-operate and/or compete. They represent the cores of a real multiprocessor architecture. These virtual robots can have their own clock, and different times for carrying out their specific tasks.
- The environment model ("city") where the robots carry out their tasks supports exclusive areas, partially shared areas and fully shared areas. An exclusive area allows only one robot to move in it, a partially shared area specifies the set of robots that can move in it, and a fully shared area allows all robots defined in the program to move in it.
- If only one robot is used in an area that encompasses the entire city, the scheme used in Visual Da Vinci is repeated.
- When two or more robots are in a (partially or fully) shared area, they compete for access to the corners on their runs, and the resources found there. For this, they must be synchronized.
- When two or more robots (in a common area or not) wish to exchange information (data or control), they must use explicit messages.
- Synchronization is done through a mechanism that is equivalent to a binary semaphore.
- Mutual exclusion can be generated by stating the areas reached by each robot. Entering other areas in the city, as well as exiting them, is not allowed.
- The entire execution model is synchronous and allows the existence of a cycle virtual clock which, in turn, allows assigning specific times for the operations, simulating the existence of a heterogeneous architecture.
- The environment allows executing the program in a traditional manner or with step-by-step instructions, giving the user detailed control over program execution to allow them controlling typical concurrency situations such as conflicts (collisions) or deadlocks.
- In the step-by-step mode, the effect of the operations can be reflected on physical robots, communicated through Wi-Fi. The physical robots have Linux as operating system, which allows running an http server implemented on NodeJS [17]. Thus, the environment communicates with the robots (each physical robot corresponds to a virtual one in the environment). These are point-to-point, two-way communications, i.e., the environment sends instructions to the physical robot and then the robot sends its response to the environment stating that the instruction given has been fulfilled.

3. Heterogeneity in Parallel Architectures

3.1. General Concepts

Since the early computers, there has been an ever-present desire to increase their computational power. However, it is currently hard increase processor speed by increasing their clock rate. Hardware architects face two issues: heat generation and energy consumption. The solution to this problem introduced by designers has been integrating two or more computational cores within a single chip, which is known as multicore processor. Multicore processors improve application performance by distributing work among the available cores [8,18].

Current general classifications identify two types of multicores based on the features of the cores:

- Homogeneous Multicore Architectures: all cores have the same features.
- Heterogeneous Multicores Architectures: these have cores with different features in relation to performance and energy consumption, and they can use different ISAs (Instruction Set Architecture) or not.

Currently, research is focusing on this second set of multicores, since having different types of cores allows optimizing performance and, when tasks are appropriately distributed among cores, higher performance/energy ratio efficiencies are achieved.

In this type of architectures, heterogeneity is present in various aspects, most significantly in core computational power (computation speed), memory access time, and communication speed among cores. These three aspects determine the time required to execute the instructions in each core, and thus, the same sentence executed by two different cores can take different times. On the other hand, since there is a certain degree of independence of the features that cause heterogeneity, not all instructions are affected in the same proportion. That is, a floating point

operation that is run in core A, may take a fourth of the time it takes when run in core B, while a writing operation may take half the time when run in it.

To analyze the performance achieved by parallel applications on these architectures, traditional metrics are used – speedup and efficiency [19].

Speedup (S) is a measure that allows quantifying the relative benefit of solving a problem in parallel, i.e., how "fast" the parallel algorithm carries out the task compared to the sequential one. The speedup function is defined as the relation between the best sequential algorithm on a single core (T_S) and the time required to solve the same problem on a parallel architecture (T_P), as shown in Eq. 1 [20].

$$S = \frac{T_S}{T_P} \quad \text{(Eq. 1)}$$

Efficiency (E) is a measure of the fraction of time during which cores use is productive in the parallel application. This metric is defined as the relation between the speedup achieved and the optimal speedup (S_{opt}) that can be achieved in the architecture, as shown in Eq. 2 [20].

$$E = \frac{S}{S_{opt}} \quad \text{(Eq. 2)}$$

Traditionally, in homogeneous architectures the optimal speedup is given by the number of cores used (p). However, in the case of heterogeneous architectures, the computation power of the various cores in it must be taken into account, which results in a re-definition of the optimal speedup as is shown in Eq. 3.

$$S_{opt} = \sum_{i=0}^{p-1} \frac{Power\ (\text{Core}_i)}{Power\ (\text{most powerful core})} \quad \text{(Eq. 3)}$$

3.2. Heterogeneity in CMRE.

A core's power computation is given by the processor's clock; in the case of heterogeneous multicores, each core must have its own clock. This situation is modeled in CMRE using a general system clock for simulation (which is the fastest one) and multiples of that clock for the robots. This can be defined as using virtual clocks for each robot/processor.

The Put Down and Pick Up operations can be assimilated to Write and Read operations in real processors. Naturally, a heterogeneous architecture can have different times which, in the case of CMRE will be multiples of the general clock.

Similarly, communication times (Send and Receive) have to be considered, and even the time required for operations such as Report, which may have different clock cycle numbers per processor/robot.

Within CMRE, a processing magnitude relative to the pattern speed can be configured for each robot: "n times slower than." Additionally, the number of clock cycles consumed for each of its instructions can also be defined, i.e., heterogeneity can be configured based on the needs of the task at hand and the concept that is being delivered.

It should be noted that, for introducing concepts, simple differences can be defined for the virtual clocks and then on the cost of each clock cycle for robot operations. Separating these two aspects helps students understand that times are not only related to processing, but also to memory accesses and communications.

3.3. Heterogeneity and Load Balancing.

The load balance in an application directly affects its performance. This is because, if the system is unbalanced, it means that there are cores that remain idle while waiting the other cores to finish their tasks; this increases the final execution time and, therefore, decreases speedup and efficiency. It depends on the features of the application and the parallel architecture being used.

From the application side, in the case of CMRE, this will be strongly dependent on the activity carried out by each robot in their algorithms since, depending on the instructions executed as well as the specific configuration of the city, work may or may not be balanced. For instance, if robots are required to pick up pieces of paper in a private area (same size for all robots), load balancing will depend on the number of papers present in each area.

Taking into account the parallel architecture, core heterogeneity is a key factor that affects load balance. For instance, if two robots have to run through an avenue from end to end, but they move at different speeds, the one that moves taster will have to wait idle until the other one completes its run, which will affect load balance.

If we combine both aspects, the issue of balancing loads becomes even more complex, but also more challenging for students when they are required to find an algorithm that is efficient, which involves additional motivation when they implement it.

3.4. Simple Experimental Case Showing the Impact of Heterogeneity.

There are 4 robots with relative speeds of 1 step per block, 2 steps per block, 3 steps per block, and 4 steps per block. This means that the fastest robot (R1) will cover, for instance, 100 blocks of an

Avenue in 100 units of time, while the slowest one (R4) will travel the same distance in 400 units of time.

There are 4 Avenues (Av1, Av2, Av3, and Av4) that have to be run in their entirety (1 per robot), each with its own distribution of objects to be picked up, for instance, 10, 20, 40 and 80 objects, respectively. Figure 1 shows a diagram of this example.

The time required to pick up an object is 3T (where T is the time each robot needs to take a single step).

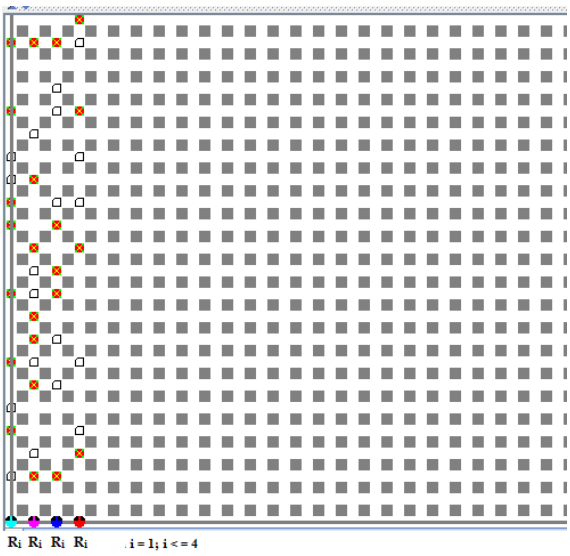

Fig. 1. Diagram of the example.

The assignment of the work to be done (in this case, which Avenue to run) will be significant for both total program time and load unbalancing:

a. If robots are assigned by direct association of robot speed and Avenues, considering expected work, we have:

   R1 runs through Av4 and takes 100 + 80x3 = 340 T
   R2 runs through Av3 and takes 200 + 40x3 = 320 T
   R3 runs through Av2 and takes 300 + 20x3 = 360 T
   R4 runs through Av1 and takes 400 + 10x3 = 430 T

   The program would take 430 T and the maximum idle time would be for R2 (110 T).
   Total idle time (which is a measure of load unbalance) would be 270 T.

b. If robots were assigned by inverse association between robot speed and Avenues, considering expected work, we have:

   R1 runs through Av1 and takes 100 + 10x3 = 130 T
   R2 runs through Av2 and takes 200 + 20x3 = 260 T
   R3 runs through Av3 and takes 300 + 40x3 = 420 T
   R4 runs through Av4 and takes 400 + 80x3 = 640 T

   In this case, the program would take 640 T and the maximum idle time would be for R1 (510 T).
   Total idle time would increase to 1110 T → More than 4 times the unbalance of the previous distribution.

4. Energy Consumption in Parallel Algorithms

Energy consumption is a key aspect of current processors. In general, the performance of a parallel algorithm is not measured only in its execution time, but also in energy consumed. Thus, there will be Flops/Watt or Flops/Joule ratios corresponding to a relation between computation and instant power or total energy [21,22].

It is important to teach Computer Science students to always use consumption metrics as an indicator of algorithm quality. Additionally, they should also understand the automatic mechanisms developed by processors according to the temperature reached (which is a direct function of the energy consumed in a period of time) [21].

4.1. Processor Clock Limitation and their Modeling in CMRE

In general, all modern processors have a clock automatic adjustment curve clock based on consumption (or the internal temperature they reach), which is represented in Figure 2.

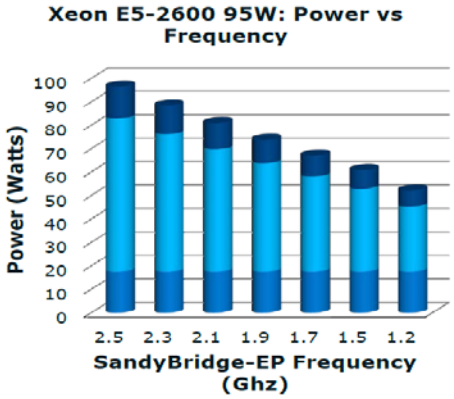

Fig. 2. Clock automatic adjustment curve.

At a given point, clock speed decreases to reduce the amount of work that the processor can do in each unit of time and thus reach an equilibrium at a lower temperature. In some cases, processor task reassignment can be managed (for instance, by controlling hardware controller information or through direct monitoring), to avoid the change in clock frequency or to adapt it to the change in frequency [21].

To model this in CMRE, the energy consumed in the operations (Move / Pick Up / Put Down /

Communicate / Report) has to be known, and the energy accumulated in each robot/processor has to be taken into account to adjust their "speed" based on a previously defined model.

This scheme is represented in Figure 2, which shows a discrete modelization of the curve represented for real processors.

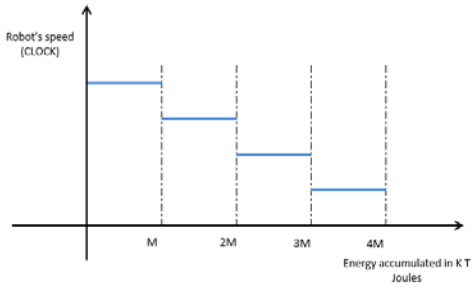

Fig. 3. Robot "speed" automatic adjustment curve.

Basically, in CMRE this means "changing" the robot's virtual clock by adjusting the robot's movement speed (and therefore, the robot's work capacity per unit of time) based on a measurement of the energy consumed in the last K units of time.

With didactic purposes, examples with K=10 have been developed simply to consider a work accumulation period that generates changes in processor internal temperature.

Based on this, the energy consumption per operation is defined, considering that MOVE requires1 Joule, and an energy cost is assigned to PUT DOWN, PICK UP, SEND, RECEIVE and REPORT. With this, calculating the total energy consumed by each robot/processor for a given algorithm is simple, and algorithms can also be compared from the point of use of energy consumption.

The adoption of a "clock change" strategy based on consumption in the simulator is more complex. So far, we have defined that, if the energy la consumed in the last 10 T of any robot is above M Joules (where M is a parameter set based on the curve shown in Figure 3), that robot's speed (clock) is decreased in one step for a minimum of R cycles (we are currently using R=3), and then calculations are re-started. If the robot is below M Joules in the last 10 T, it recovers its previous clock.

Naturally, this means that speedup, efficiency and maximum consumption vary in each 10 T interval, depending on the parallel algorithm that is being run. Additionally, there us a (virtual) measurement of the total consumption of a given algorithm.

4.2. Experimental Example

The example used for heterogeneity (section 3.4) can be relatively easily extended to the case of energy consumption y and consider clock change based on it. To the example that has already been discussed, information regarding the amount of energy consumed by the operations is added: any PICK UP operation consumes 5 Joules, and any MOVE operation consumes 1 Joule.

a. Using the first work assignment described in the example in section 3.4 (a), we have:

   R1 runs through Av4 in 100+80x3 = 340 T and consumes 100+80x5 = 500 Joules.
   R2 runs through Av3 in 200+40x3 = 320 T and consumes 200+40x5 = 400 Joules.
   R3 runs through Av2 in 300+20x3 = 360 T and consumes 300+20x5 = 400 Joules.
   R4 runs through Av1 in 400+10x3 = 430 T and consumes 400+10x5 = 450 Joules.

   The program takes 430 T and consumes a total of 1750 Joules.
   It is possible that, if the clock restriction discussed above is applied, possibly to R1 or R2, some of the robots would HAVE TO decrease its speed (based on load distribution), and, in that case, *total consumption would be constant, but the program would take longer and load unbalance could change.*

b. Using the inverse assignment used in section 3.4 (b), consumption would be:

   R1 runs through Av1 in 100+10x3 = 130 T and consumes 100+10x5 = 150 Joules.
   R2 runs through Av2 in 200+20x3 = 260 T and consumes 200+20x5 = 300 Joules.
   R3 runs through Av3 in 300+40x3 = 420 T and consumes 300+40x5 = 500 Joules.
   R4 runs through Av4 in 400+80x3 = 640 T and consumes 400+80x5 = 800 Joules.

   In this case, the program would take 640 T and it would consume the same total energy (1750 T). This is expected, because the total work to be done in MOVE and PUT DOWN operations is the same in both cases.
   However, it is less likely that any of the robot clocks will have to be decreased.

5. Conclusions and Future Lines of Work

Heterogeneity and energy consumption in parallel architectures are highly relevant issues, and they have been modeled on an environment called CMRE.

CMRE appears to be a very useful tool for introducing these concepts, based on its use since 2013 with homogeneous robots/processors.

Even though there are changes that should be implemented in the environment, these are transparent to the student and the development of

experimental work oriented to learning these topics is very natural.

We are currently working on the implementation of these extensions in 2016, considering that, when different clocks are enabled for the different robots, there will be runs with potential concurrency conflicts not only on city corners, but also at intermediate points between two corners. Thus, the complexity level of possible scenarios is increased, which poses a much more ambitious challenge that matches the technological reality of current processors.

References

[1] Hoare C. "Communicating Sequential Processes". Prentice Hall, 1985.

[2] Dijkstra E. W. "Finding the Correctness Proof of a Concurrent Program". In Program Construction, International Summer Schoo, Friedrich L. Bauer and Manfred Broy (Eds.). Springer-Verlag, 24-34, 1978.

[3] Hansen P. B. "The Architecture of Concurrent Processes". Prentice Hall, 1977.

[4] Dasgupta S. "Computer Architecture. A Moder Synthesis. Volume 2: Advanced Topics". Jhon Wilet & Sons. 1989.

[5] ACM Curriculum Committee on Computer Science. "Curriculum '68: Recommendations for the undergraduate program in computer science". Communications of the ACM, 11(3):151-197. 1968.

[6] ACM Curriculum Committee on Computer Science. "Curriculum '78: Recommendations for the undergraduate program in computer science". Communications of the ACM, 22(3):147-166. 1979.

[7] ACM Two-Year College Education Committee. "Guidelines for associate-degree and certificate programs to support computing in a networked environment". New York: The Association for Computing Machinery. 1999.

[8] Gepner P., Kowalik M.F. "Multi-Core Processors: New Way to Achieve High System Performance". In: Proceeding of International Symposium on Parallel Computing in Electrical Engineering 2006 (PAR ELEC 2006). Pp. 9-13. 2006.

[9] McCool M. "Scalable Programming Models for Massively Parallel Multicores". Proceedings of the IEEE, 96(5): 816, -831, 2008, 17, 21.

[10] Hoonlor A., Szymanski B. K., Zaki M. J., Thompson J. "An Evolution of Computer Science Research". Communications of the ACM. 2013.

[11] ACM/IEEE-CS Joint Task Force on Computing Curricula. "Computer Science Curricula 2013". Report from the Task Force. 2013.

[12] ACM/IEEE-CS Joint Task Force on Computing Curricula. "Computer Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering". Report in the Computing Curricula Series. 2004.

[13] ACM/IEEE-CS Joint Interim Review Task Force. "Computer Science Curriculum 2008: An Interim Revision of CS 2001". Report from the Interim Review Task Force. 2008.

[14] De Giusti, L., Leibovich, F., Sanchez, M., Chichizola, F., Naiouf, M., De Giusti, A. "Desafíos y herramientas para la enseñanza temprana de Concurrencia y Paralelismo". Congreso Argentino de Ciencias de la Computación (CACIC), 2014.

[15] AMD. "Evolución de la tecnología de múltiple núcleo". http://multicore.amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution. 2009.

[16] De Giusti, A., De Giusti L., Leibovich, F., Sanchez, M., Rodriguez Eguren, S. "Entorno interactivo multirrobot para el aprendizaje de conceptos de Concurrencia y Paralelismo". Congreso Tecnología en Educación, Educación en Tecnología. 2014.

[17] NodeJs API. https://nodejs.org/api/http.html

[18] Mc Cool M, "Programming models for scalable multicore programming", 2007, http://www.hpcwire.com/features/17902939.html.

[19] Dongarra J. , Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. "Sourcebook of Parallel computing". Morgan Kaufmann Publishers 2002. ISBN 1558608710.

[20] Grama A., Gupta A., Karpyis G., Kumar V. "Introduction to Parallel Computing". Pearson – Addison Wesley 2003. ISBN: 0201648652. Second Edition (Chapter 3).

[21] Balladini J., Rucci E., De Giusti A., Naiouf M., Suppi R., Rexachs D., Luque E. "Power Characterisation of Shared-Memory HPC Systems". Computer Science & Technology Series – XVIII Argentine Congress of Computer Science Selected Papers. Pp. 53-65. 2013.

[22] Brown D. J., "Toward Energy-Efficient Computing", Magazine Communications of the ACM Volume 53 Issue 3, March 2010