Parallel Processing of Economic Programs, a New Strategy in Groups of Firms

Loredana MOCEAN, Monica-Iuliana CIACA, Alexandru VANCEA
Babeş-Bolyai University
loredana.mocean@econ.ubbcluj.ro, monica.ciaca@econ.ubbcluj.ro, alexandru.vancea@cs.ubbcluj.ro

In recent years parallel and distributed systems have become increasingly attractive for applications with high computational demands such as simulation of complex systems from groups of companies. The main advantage of such systems is the ratio, rather than attractive, between the price and performance that can be achieved. In the present paper, authors describe some possibilities of parallel processing at the level of economic programs in groups of firms. The architecture, model and future development are shown below. This paper is an extended version of the paper presented at International Conference on Informatics in Economy (IE 2013), Bucharest, Romania
Keywords: *Decidable Problems, Parallel Processing, Parallel Calculus, Groups of Firms*

1 Introduction

Having a quite complicated organizational structure, behavioral flexibility and lack of bureaucracy – present in all the sectors of the industry, commerce and services, groups of firms easily adapt to the constantly changing economic and social conditions.
Modeling of group of companies economic systems containing hundreds of market actors (companies) who apply management strategies common to group or individually (at company level) can be a difficult task. During the last years, had been remarked two classes of individual strategies: one based on envy 'comp benefit "(companies compare their benefits with those of competitors and copy competitors with higher profit side, well known in the literature) and maximization strategies "max-benefit" (company calculates benefits obtained for increasing, decreasing, or maintaining selling prices and take a decision as to maximize its profits without regard for other companies).

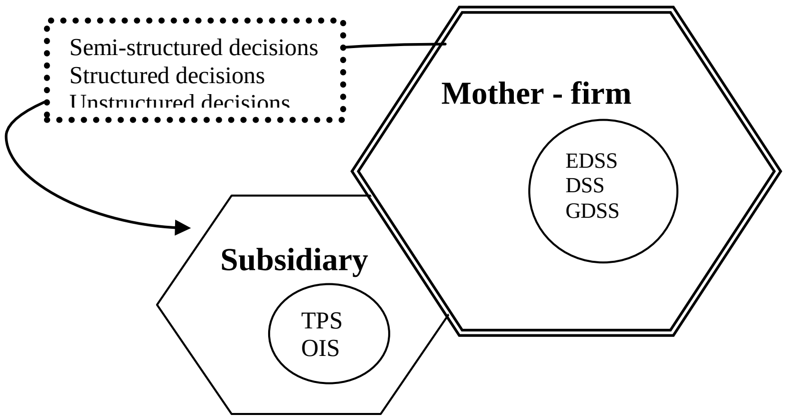

Fig. 1. Mother-firm management and subsidiaries management

The necessity of passing, on a large scale, from the mother-firm management to the subsidiary management required passing from sequential programming to parallel processing, in groups, having at least the following basic motivations:

- No matter the future performance of one processor, it is impossible to have an unlimited increase of execution capacities and efficiency of the uniprocessor systems;

DOI: 10.12948/issn14531305/18.2.2014.11

- The inherent and strong parallel nature of many algorithms; such an algorithm suggests the design of a program by initiating multiple processes, that would cooperate to fulfill a common purpose;

2 Related Work

Research on how the IT community can perform this compulsory passage has started during the 7th decade ([1], [4], [5], [7]).

As compared to other problems whose solutions were found more quickly, parallel programming has not yet managed to assert itself as a general method of design, implementation and execution of algorithms, at the level of groups of firms.

Solutions for problems arising from a completely new approach have proved to be difficult to find, at the level of theoretical substantiation, but *especially at the level of mentalities in the developers community*.

The fact that the broad IT community is learning and thoroughly studying an algorithmic approach that is inherently sequential used at the level of imperative programming languages (actually also deriving from the inherent sequential nature of management) makes this compulsory transition become extremely complicated.

3 Research Methodology

A parallel system can be used to describe groups of firms indicating the firms central of the group, that coordinates the function of strategy and eventually influence the company has focused on companies in the group.

A parallel system can be used to describe how they are affected by price fluctuations on where a company is located, competition neighbors, the purchasing power of the customers in that area, etc.

Taking groups of firms as practical example, we identify two practical ways of passing from group management to subsidiary management (referring to the passage from sequential programming to parallel programming):

1). Designing algorithms conceived by parallel approach that would then be implemented at the level of some programming languages designed for such execution;

2). Elaboration of specialized software for the automatic transformation of sequential programs in efficient parallel versions.

For the current stage, only the level of technological development of hardware equipment could contribute to the rapid finalization of such a transition (despite the lack of architectural standards for parallel computers). Unfortunately, the smart and especially correct management of the resources involved at the level of an algorithm proves to be quite difficult when trying a parallel approach of algorithm elaboration. This is why it remains more as a research domain, rather than a well established and universally accepted practical methodology.

For these reasons, a large part of the current research, oriented towards the parallel processing in groups of firms is directed towards the development and analysis of theoretical models that would allow the substantiation of certain general constructive principles, as well as the efficient implementation of *restructuring compilers* (translators which restructure a sequential program for the purpose of parallel execution).

The development of algorithms for groups of firms using the parallel approach will certainly remain, in time, the only programming methodology able to justify and accomplish the computer science development, in terms of software.

4 Manifestation of Parallel Processing Possibilities, at the Level of a Program in Groups of Firms

The main feature of imperative languages is the reflection of the *von Neumann architecture* at the level of language constructions. Such a paradigm is focused on the ***assignment instruction*** and provides programs whose effect can be described as a ***sequence*** of transformations of the memory cells values on which the program acts.

DOI: 10.12948/issn14531305/18.2.2014.11

The complete programming methodology used in the last decades by the programmers' community emphasizes the following description of the algorithm design activity: any algorithm is nothing more than ***a sequence of assignment instructions*** that a programmer imposes to certain memory locations through control structures available through the programming language that they work with.

It is worth noticing the necessity of ***sequencing***, imposed by the intellectual activity of elaborating algorithms according to the current methodology.

On the other hand, the varied nature of the real problems that computer science is expected to solve often makes algorithmic sequencing become obviously forced in relation to a natural description of a solution for that problem, a solution that most of the times contains activities that can be and should be executed simultaneously.

One of the main reasons for using the computer is the repetition of a certain sequence of instructions for a large amount of data. Parallel processing becomes an essential factor, the only possible one, for obtaining the desired performance, to the extent that the data features allow it.

Imperative languages are mainly oriented towards scientific calculations involving large amounts of data. The problems of programs are not much related to their length (requirements of internal and external memory are mostly already solved, from the technological point of view) but to *generally large amount of time required for their execution*.

Thus, the identification of parallelizing possibilities at the level of a sequential program becomes of maximum importance, as a first step towards obtaining its parallel version.

It is obvious (experimental studies prove it) that *the most of the time required for a sequential execution (the classical estimation is around 90%!)[6] is consumed during the iterations*. Thus the iteration loops become the main candidates for parallelization.

Hierarchically structured machines [3] currently represent the dominant configuration of hardware elements in computer systems. The feature of these systems is the ***N****on-****U****niform* ***M****emory* ***A****ccess time*, which is why they are called NUMA machines.

sNon-uniform access is reflected in relation to the major difference between the time required for a processor to access data in the local memory associated to it and the time required to access data located at a distance (most probably through communication methods based on message exchange).

The execution itself, of a parallel program on a particular system in the group, requires partitioning, distribution (also called ***mapping***) and planning of data and calculations at the level of the nodes in a network of processors.

The creation of an application that would accomplish this optimally, in an automatic way, and independently from the particularities of the computer system is considered to be an NP-complete problem [9][2].

In the absence of an automatic support which can accomplish this task, the optimal partitioning of calculations and data remains, in many cases, the responsibility of the developer, who manually performs it at the level of the program source code, using a specialized description language (as in the case of the Occam language, for example [8]).

In order to emphasize the utility of equivalent transformations at the level of loops, let us consider a distributed system (network of processors with local memory) where the following program sequence is executed:

```
for i := 1 to m do
            for j := i to n do
             A[i, j] := A[i-1, j] + 
B[j];
            end for
end for
```

where A is an $m \times n$ matrix, B is an n-vector and $n \geq m$.

For the example above, let us assume that we make a mapping of the calculations, so that

DOI: 10.12948/issn14531305/18.2.2014.11

each instance of the outer loop is executed at the level of one processor (thus we must have at least *m* available processors); in this case, each processor will sequentially execute the corresponding iterations of *j* from the previous loop. Thus, the *k* processor will execute all the iterations with *i* = *k*. Regarding the data mapping, let us assume that we are making a distribution in order for processor *k* to store the *k* line of the matrix in its local memory (the notation of the line is *A*[*k*,∗]) and the element *B*[*j*] is stored in the processor's memory (*j* **mod** (*m*+1)).
In such a situation (figure 2) the *k* processor will execute *n*-*k*+1 iterations, but at least (*n*-⌈*n*/(*m*+1)⌉) elements of vector B must be brought from the processors where they have been distributed. In addition to this, each *k* processor must bring line *A*[*k*-1,∗] from the (*k*-1) processor.

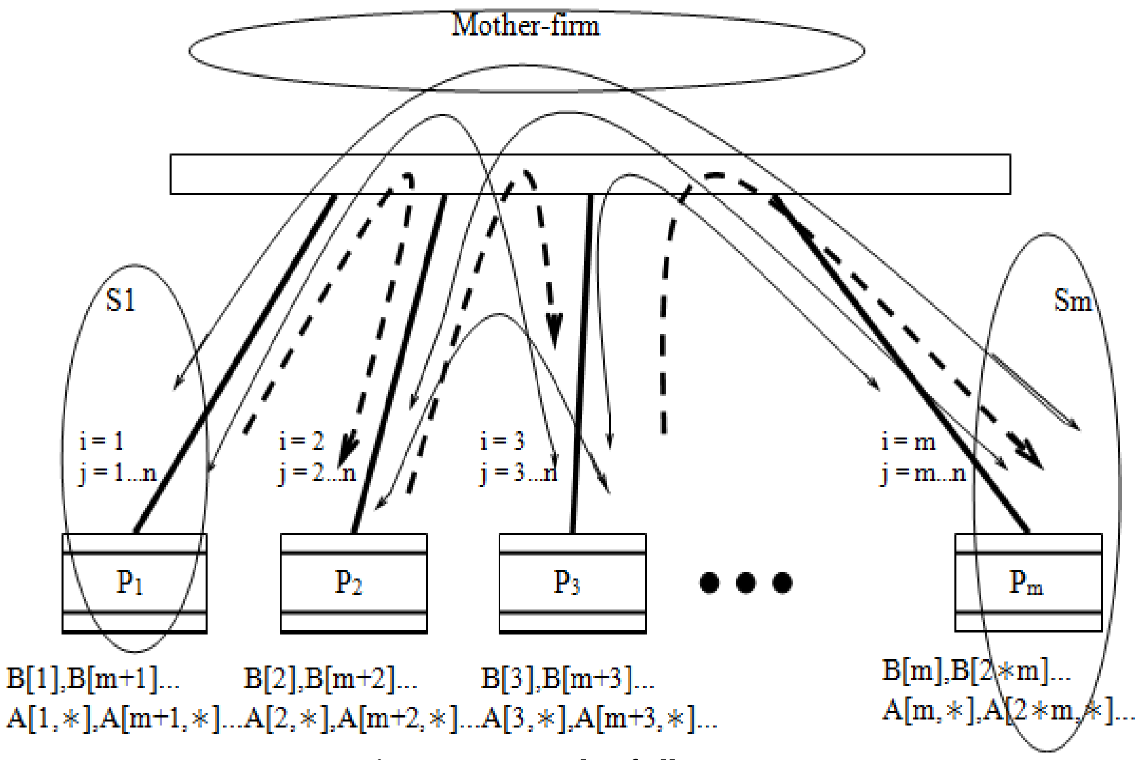

Fig. 2. An example of allocation

The thin lines correspond to moving the elements of vector *B* and the thick dotted ones, correspond to movements of elements of *A*. Let us now take an example that is semantically equivalent to the sequence above, in which we assume that *n* processors are available:

```
for j := 1 to n do
      for i := 1 to min(j,m) do
       A[i,j] := A[i-1,j] + B[j];
      end for
end for
```

Assuming that processor *k* executes all the iterations with *j* = *k* and stores element *B*[*k*] and column A[∗,*k*] in its local memory, then processor *k* will execute min(*k*,*m*)+1 iterations. Element *B*[*k*] is never changed, and can be stored even at the level of an available register. However, the most important change is the presence of all elements of *A* accessed by the processor at the level of its local memory (see figure 3).
The two analyzed mappings use *m* and *n* processors, respectively.
Since *n* ≥ *m*, the second mapping achieves a greater degree of parallelism as compared to the first case, but the greatest benefit is that all the data required for the calculations are locally stored and thus it is not necessary to perform any data transfer operation.

This makes operations be independent, allowing them to execute simultaneously. As opposed to this, the first version involves the necessity of many data exchanges among the processors, which considerably increase the execution time.
The second mapping also displays a better *static localization*, even perfect in this case. The elements of vector *B* can be stored in the cache memory or in an available register, to be used in each iteration, thus also obtaining a better *dynamic localization*.
In the first version, the access requests for the value of the same *B*[*k*] came from more processors.
In both cases analyzed above, processors do not perform the same number of iterations, which leads to a variation of the computational load, at processors level.
If the variation of this load is too big, there is a negative effect on the performance of the computer system.
Considering the two cases discussed, the second mapping has a smaller variation, thus displaying a greater degree of *balanced processor load* (IEP).

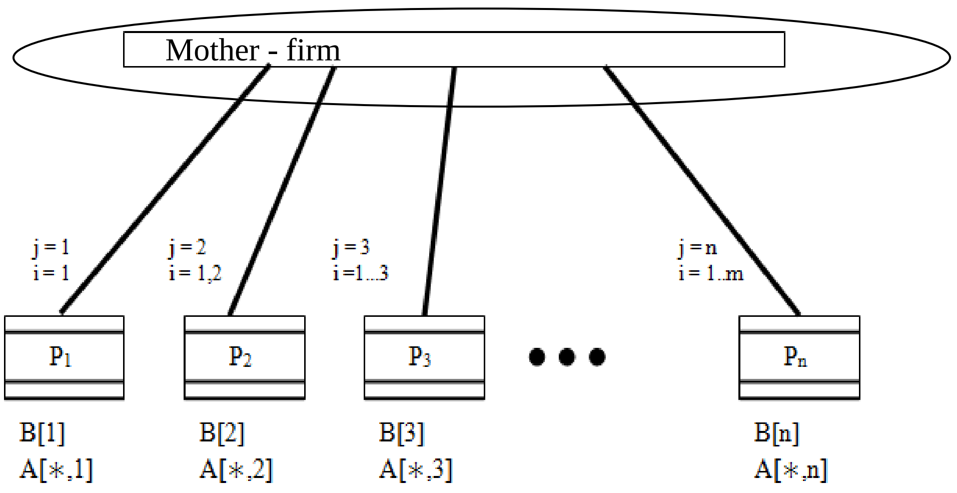

Fig. 3. An optimal allocation

These two examples show us that different structures of nested loops, semantically equivalent, can represent very different execution times, depending on the degrees of parallelism, localization and IEP displayed.
The transformation of a nested loop structure (SCI) in a semantically equivalent SCI, displaying possibilities of parallel execution is the main purpose of a restructuring translator and at the same time it is one of the main analysis and research objectives in groups of firms.
The theoretical basis of the transformation methodology of the SCI, is the data dependency mathematical concept. Data dependencies are a measure of parallelism that can be highlighted in the source code. Therefore, their accurate determination is of paramount importance for effective parallelization. Unfortunately, analysis of data dependencies is in the general case determining undecidable problem.
For example, in the code sequence:

```
read(n);
for i := 11 to 20 do
        A[i] := A[i-n] + 3;
end for
```

(in) dependence of the two references to elements of array A depends on the value of n which is not unknown at compiling time.
That means that in the management of the mother – firm, for each subsidiary we can consider the dependence between all *i-n* companies (see figure 4).

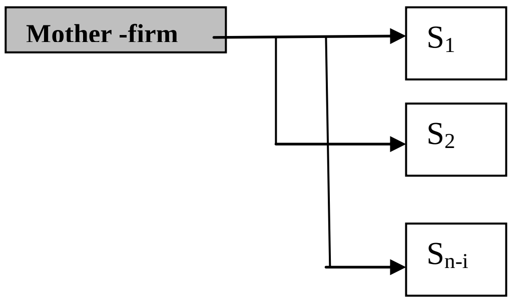

Fig. 4. The dependence between i-n subsidiaries

This is necessary to implement dynamic analysis methods of data dependencies. Even if we limit ourselves only to the static aspect of the problem, complications persist. Let us consider for example the following code, we want to analyze whether the array indexing expressions V will denote the same element or not:

```
if n > 2 then
    if a > 0 then
        if b > 0 then
      if c > 0 then
          V[a^n] := V[b^n + c^n] + 3;
      end if
         end if
   end if
end if
```

We must solve the equation given of the condition of equality of the corresponding index expressions.

The problem of data dependences in this case would mean neither more nor less than the problem of demonstrating Fermat's great theorem (the existence of three positive integers a, b, c> 0 which satisfy the equation, $a^n = b^n + c^n$, $n \in \mathrm{N}$) theorem which until now was neither demonstrated nor contradicted (although was verified the absence of such numbers verified to very high values, the problem is that we don't have another *methodology for proving* such theorems than just checking!).

Also, the classical un-decidable problem of stopping a program (*halting problem*) can be formulated in such a framework of determining dependencies in a program.

For these reasons, the approach of the analysis of data dependencies is doing only in a relatively small area, namely the affine index expressions (ie those of the form *ax* + *b*, expressions linear plus a constant) which occur in the majority of situations in practice. The problem formulated in such a framework is decidable, but obtaining accurate solutions can be a very costly process. Therefore, if you cannot find the exact solution or finding it too expensive, it is assumed conservatively dependence.

On the other hand, the restricting of the analysis of affine equations, followed automatically by data dependencies conservative assumption in all other cases, we can lose a large amount of potential parallelism. For example, in sequence:

```
for i := 10 to n do
     A[i^2] := A[3] + 2;
end for
```

is not known value of n, we do not only make affine index expressions (ie data analysis will conservatively assumed dependence), but it is obvious that the equation $i^2 = 3$ has no solution in the set of integers and hence the dependence between these two references for array there.

The emergence of expressions un-affine therefore means loss of information, but not always necessarily forced to assume dependence. For example, if other dimensions of the array elements involved are affine, we can use these to demonstrate (in) dependence:

```
for i := 10 to 20 do
      A[i^2][2i] := A[3][2i+1] + 2;
end for
```

The first dimension contains un-affine term, but the terms of the second dimensions are both affine.

Data Dependency would require the simultaneous satisfaction of the equations representing the identity of reference in the two dimensions. Analyzing only what the second dimension but it appears immediately that equation $2i = 2i + 1$ has no solutions, so we do not have data dependencies.

As architectures become more complex, significantly increasing the number of directions of optimization and decision

making relative to the range of transformations applied is very complicated.
The problem of choosing an optimal sequence of transformations leading to the most efficient parallel version remains an open question. Relative to this, compilers moment only managed to incorporate a set of heuristic decision.
Before attempting to generate an optimal sequence of loop transformations, it is natural to ask whether in the general case, a program always supports optimal planning scheme execution.

5 Conclusions

In mathematics, computer science or management, mathematical optimization (alternatively, optimization or mathematical programming) is the selection of a best element (with regard to some criteria) from some set of available alternatives.
In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function.
The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics.
More generally, optimization includes finding "best available" values of some objective function given a defined domain, including a variety of different types of objective functions and different types of domains.
As machines become more complex, the number of optimization directions increases significantly, and the decision making process related to the transformations to be applied becomes very complicated.
The problem of choosing an optimal sequence of transformations that would lead to the most efficient parallel version remains an open one.
Regarding this aspect, the current compilers only manage to embed a certain set of heuristic decisions in the activity of the groups of firms.
This paper has attempted to connect the management group of companies and parallelization algorithms and proposes a parallelization of their activity.

References

[1] F.E.Allen "Program optimization", in Annual Review in Automatic Programming 5, *International Tracts in Computer Science and Technology and their Applications*, vol.13, Pergamon Press, Oxford, England, pp.239-307, 1969.

[2] Bane, D "Utpal Banerjee - Speedup of ordinary programs", *PhD thesis*, Report 79-989, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1979.

[3] S. Dasgupta "Computer Architecture", vol.1 and 2, John Wiley and Sons, New York, 1989.

[4] Y.Muraoka, "Parallelism exposure and exploitation in programs", *Ph.D. thesis*, Tech.Rep. 71-424, University of Illinois at Urbana-Champaign, 1971.

[5] R.M.Karp, R.E.Miller and S.Winograd, "The organization of computations for uniform recurrence equations", in *Journal of the ACM*, 14(3), pp.563-590, July 1967.

[6] D.Kuck, P.Budnik, S.C.Chen et al. ,"Measurements of parallelism in ordinary FORTRAN programs", in *Computer*, vol.7, nr.1, 1974, pp.37-46.

[7] L. Lamport "The parallel execution of DO loops", in *Communications of the ACM*, 17(2), 1974.

[8] D. Pountain and D. May, "A Tutorial Introduction to Occam Programming", McGraw Hill, 1987.

[9] M. Wolfe, "High Performance Compilers for Parallel Computing", Addison-Wesley, Redwood, 1996.

[10] L. Mocean, M. Ciaca, A. Vancea, "Possibilities of parallel processing at the level of economic programs in groups of firms", *Conference Proceedings of the 12 ICIEERBT*, 2013

Loredana MOCEAN has graduated Babes-Bolyai University of Cluj-Napoca, the Faculty of Computer Science, she holds a PhD diploma in Economics and she had gone through didactic position of assistant, lecturer and associate professor, since 2000 when she joined the staff of the Babes-Bolyai University of Cluj-Napoca, Faculty of Economics and Business Administration. Also, she graduated Faculty of Economics and Business Administration. She is the author of more than 20 books and over 35 journal articles in the field of Databases, Data mining, Web Ser-vices, Web Ontology, ERP Systems and much more. She is director or member in more than 20 grants and research projects, national and international.

Dr. **Monica Iuliana CIACA** obtained her bachelor's degree at Babes Bolyai University Cluj-Napoca, in the field of Computer Science. After graduation, she has worked as programmer at the Institute for Computation Techniques from Cluj-Napoca. In 1994 she started working at the Babes Bolyai University as teaching assistant, being interested in artificial intelligence, expert systems, business information systems and software engineering. She published various articles, the most important being the one written after her participation in a Tempus Phare project, in Perugia. In 2003 she got her PhD in Mathematics and Computer Science, with a thesis on parallel computing: "Implementation Techniques in Parallel Computing". In the last five years she looked to extend her knowledge in another field: theology. She obtained her Bachelor's Degree and Master's Degree in Biblical Studies and Iconographic exegesis, in 2012, at Babes Bolyai University. Since 2004 she is Associate Professor at Babes Bolyai University, Cluj-Napoca, Faculty of Economics, in the Department of Business Information Systems.

Dr. **Alexandru VANCEA** has graduated the Computer Science Department of "Babes-Bolyai" University Cluj-Napoca in 1986. Ph.D. in Computer Science in 2000. Research areas and domains of interests: Programming Languages Design and Analysis, Automatic parallelization of programs, Distributed Programming. Teaching: Operating Systems, Computer Architecture, Fundamentals of Programming Languages.

DOI: 10.12948/issn14531305/18.2.2014.11