STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVII**, Number 3, 2012

BUILDING GRANULARITY IN HIGHLY ABSTRACT PARALLEL COMPUTATION MODELS

VIRGINIA NICULESCU

Abstract. The purpose of this study is to investigate the necessity, the existence, or the possibility to introduce mechanisms for specifying and building granularity into parallel computation models with high level of abstraction. If such mechanisms exist in this kind of models they are very useful since they allow a better evaluation of the performance, and finally an easier and more efficient implementation.

1. Introduction

One key to attaining good parallel performance is choosing the right granularity for the application. An important goal is to determine the right granularity for parallel tasks. In parallel setting the need for a unifying parallel model or a set of models is heightened by the greater demand for performance and the greater diversity among machines. Parallel computation models with high degree of abstraction usually do not have mechanisms for specifying and building granularity. If such mechanisms could be introduced they are very useful since they allow a better evaluation of the performance, and finally an easier implementation. From a practical point of view, algorithms that allow an adjustment of the granularity are preferable. It is more realistic to assume that the number of available processors is limited. A model of parallel computation, to be useful must address both issues: abstraction and effectiveness, which are summarized in the following set of requirements: abstractness, software development methodology, architecture independence, cost measures, no preferred scale of granularity, efficiently implementable [10].

We intend to focus our analysis on the models with high level of abstraction and to evaluate how the last two criteria are fulfilled. The possibility of improving this kind of models based on granularity building is going to be analyzed.

Received by the editors: June 26, 2012.

2010 *Mathematics Subject Classification.* 68Q85 , 65Y20.

1998 *CR Categories and Descriptors.* D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming*; D.2.10 [**Software Engineering**]: Design – *Methodologies*.

Key words and phrases. parallel computation, models, granularity, performance, design, abstraction.

2. Level of parallelism and granularity

In modern computers, parallelism appears at various levels both in hardware and software. Levels of parallelism can also be based on the lumps of code (grain size) that can be a potential candidate for parallelism. The main categories of the code granularity for parallelism are described in Figure 1.

Grain Size	Code Item	Parallelised by
Very Fine	Instruction	Processor
Fine	Loop/Instruction block	Compiler
Medium	Standard One Page Function	Programmer
Large	Program-Separate heavyweight process	Programmer

Figure 1. The main categories of the code granularity for parallelism.

The granularity is often related to how balanced the work load is between tasks. While it is easier to balance the workload of a large number of smaller tasks, this may cause too much parallel overhead in the form of communication, synchronization, etc. Therefore, one can reduce parallel overhead by increasing the granularity - amount of work - within each task by combining smaller tasks into a single task.

Cost improvement

The size of work in a single parallel task (granularity) of a multithreaded application greatly affects its parallel performance. When decomposing an application for multithreading, one approach is to logically partition the problem into as many parallel tasks as possible. Next step is to determine the necessary communication between the parallel tasks. Since partitioning tasks, assigning the tasks to threads, and communicating (sharing) data between tasks are not free operations, one often needs to agglomerate, or combine partitions, to overcome these overheads and achieve the most efficient implementation. The agglomeration step is the process of determining the best granularity for parallel tasks [4].

If we are able to specify and to build granularity from the first levels of design, which are based on models with high degree of abstractions, then the chances to obtain good improvements of the resulted costs increase very much. This is due to the fact that the grain sizes are included in the models as parameters, and the best values for the actual parameters could be computed based on the concrete values of the target architecture.

3. Models of parallel computation

A model of parallel computation is an interface separating high-level properties from low-level ones. The role of a model is to provide a single machine - an independent, simplified target for the development and specification of programs. Unfortunately parallel computing lacks a single universally accepted model, because parallel architectures can vary drastically and because the performance of parallel algorithms depends very much on the details of the architecture. More concretely, a model is

an abstract machine providing certain operations to the programming level above and requiring implementations for each of these operations on all of the architectures below. It is designed to separate software-development concerns from effective parallel-execution concerns and provides both abstraction and stability. Since a model is just an abstract machine, models exist at many different levels of abstraction [10].

3.1. **Parallel Programming Models Classification.** Models could be classified based on their order of abstraction, in the following categories [10]:

- **(1):** Models that abstract from parallelism completely. Such models describe only the purpose of a program and not how it is to achieve this purpose. Software developers do not even need to know if the program they build will execute in parallel. Such models are necessarily abstract and relatively simple, since programs need to be no more complex than sequential ones.
- **(2):** Models in which parallelism is made explicit, but decomposition of programs into threads is implicit (and hence so is mapping, communication, and synchronization). In such models, software developers are aware that parallelism will be used and must have expressed the potential for it in programs, but they do not know how much parallelism will actually be applied at runtime. Such models often require programs to express the maximal parallelism present in the algorithm and then the implementation reduces that degree of parallelism to fit the designated architecture, at the same time working out the implications for mapping, communication, and synchronization.
- **(3):** Models in which parallelism and decomposition must both be made explicit, but mapping, communication, and synchronization are implicit.
- **(4):** Models in which parallelism, decomposition, and mapping are explicit, but communication and synchronization are implicit. .
- **(5):** Models in which parallelism, decomposition, mapping, and communication are explicit, but synchronization is implicit.
- **(6):** Models in which everything is explicit. Here software developers must specify all of the detail of the implementation.

Within each of these categories, we may distinguish models according to the programming paradigm on which they are based on:

- **-:** relational/logic models;
- **-:** functional models;
- **-:** imperative models;
- **-:** object-oriented models;
- **-:** bio-inspired models.

3.2. **Models with high level of abstraction.** This category of models abstract from parallelism completely meaning that nothing related to parallelization is made explicit so the specification of the parallelism is implicit. We have different models in this category, some of them having a dynamic structure, and others using static structure as a basis. We enumerate some of them:

- **:** *with Dynamic Structure*
  - **-:** Higher-order Functional-Haskell, Concurrent Rewriting-OBJ, Maude

- **-:** Interleaving-Unity
- **-:** Implicit Logic Languages-PPP, AND/OR, REDUCE/OR, Opera, Palm, concurrent constraint languages
- **-:** Explicit Logic Languages-Concurrent Prolog, PARLOG, GHC, Delta-Prolog, Strand MultiLisp

: *with Static Structure*

- **-:** Algorithmic Skeletons-Cole, Darlington, P3L
- **-:** Data Parallelism Using Loops-Fortran variants, Modula 3*
- **-:** Data Parallelism on Types-pSETL, parallel sets, match and move, Gamma, PEI, APL, MOA, Nial and AT

: *with Static and Communication-Limited Structure*

- **-:** Homomorphic Skeletons-Bird-Meertens Formalism
- **-:** Cellular Processing Languages-Cellang, Carpet, CDL, Ceprol
- **-:** Data-Specific Skeletons-scan, multiprefix, paralations, dataparallel C, NESL, CamlFlight

Since these models are so highly abstract the implementer's job is very often extremely difficult; the transformation, compilation, and run-time systems must infer all of the structure of the eventual program. This means deciding:

(1) how the specified computation is to be achieved,
(2) dividing it into appropriately sized pieces for execution,
(3) mapping those pieces, and
(4) scheduling all of the communication and synchronization among them.

The second requirement is tightly connected with granularity.

Higher-order functional models. Higher-order functional models that use graph reduction for function evaluation uses parallelization for this reduction, so the granularity of the probable resulted task is not an issue of the programmer.

A similar situation is encountered for models based on *concurrent rewriting*.

Logic models. The logic models exploit the fact that the resolution process of a logic query contains many activities that can be performed in parallel. The main types of inherent parallelism in logic programs are OR parallelism and AND parallelism. OR parallelism is exploited by unifying a subgoal with the head of several clauses in parallel. Implicit parallel logic languages provide automatic decomposition of the execution tree of a logic program into a network of parallel threads. This is done by the language support both by static analysis at compile time and at run-time.

Skeletons models. The algorithmic skeletons models are based on the orchestration and synchronization of the parallel activities that are implicitly defined by the skeleton patterns. Programmers do not have to specify the synchronizations between the application's sequential parts. The communication/data access patterns are known in advance. Granularity could be an important issue for each skeleton that could be composed, but not for the entire program that just composes these skeletons.

BMF. In Bird-Meertens formalism (BMF) higher-order functions (functionals) capture, in an architecture-independent way, general idioms of parallel programming, which can be composed for representing algorithms [9, 2, 5]. Two functional are use intensively: *map* and *reduce*. The functional *map* is highly parallel since the computation of f on different elements of the list can be done independently if enough processors are available.

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n] \tag{1}$$

Tree-like parallelism is expressed by the functional *red* (for reduction) with a binary associative operator $\oplus$:

$$red\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n \tag{2}$$

Reduction can be computed on a binary tree with $\oplus$ in the nodes. The time of parallel computation depends on the depth of the tree, which is $\lceil \log n \rceil$ for an argument list of length n.

Other functionals may be defined, but these two are the most important and the most used.

Functions are composed in BMF by means of functional composition $\circ$, such that $(f \circ g)\ x = f\ (g\ x)$. Functional composition is associative and represents the sequential execution order.

BMF expressions - and, therefore, also the programs specified by them - can be manipulated by applying semantically sound rules of the formalism. Thus, we can employ BMF for formally reasoning about parallel programs in the design process. BMF is a framework that facilitates transformation of a program into another equivalent program. A simple example of BMF transformation is the map fusion law:

$$map\ (f \circ g) = map\ f \circ map\ g \tag{3}$$

If the sequential composition of two parallel steps on the right-hand side of Eq. (3) is implemented via a synchronization barrier, which is often the case, then the left-hand side of the equation is more efficient and should be preferred.

A very important skeleton in BMF is the homomorphism that reflects the divide-and-conquer algorithms [2, 5].

The BMF programs usually reflect the unbounded parallelism. The functional map, for example, has the time complexity equal to $O(1)$ with a processor complexity n (n is the list's length). This leads to a very fine granularity which is not practical.

In order to transform BMF programs for bounded parallelism, we may introduce the type $[\alpha]_p$ of lists of length p, and affix functions defined on such lists with the subscript p, e.g. map_p [5]. Partitioning of an arbitrary list into p sublists, called *blocks* may be done by the *distribution* function:

$$dist^{(p)} : [\alpha] \to [[\alpha]]_p \tag{4}$$

The following obvious equality relates distribution with its inverse, flattening: $red(|) \circ dist^{(p)} = id$, where $|$ means concatenation.

Using these, we obtain the following equality for the functional *map*:

$$map\ f\ = flat \circ map_p\ (map\ f) \circ dist^{(p)} \tag{5}$$

where the function $flat : [[\alpha]]_p \to [\alpha]$ transforms a list of sublists into a list by using concatenation ($flat = red(|)$).

For reduction, the transformation for bounded parallelism is as it follows:

$$red(\oplus)\ = red_p(\oplus) \circ map_p\ (red(\oplus)) \circ dist^{(p)} \tag{6}$$

We may consider that only the functions with subscription p will be distributed over the processors. The others will be sequentially computed.

PARES. For the model PARES [8] we encountered a similar approach. Functions and operators which represent programs are defined on special data structures (PowerList, ParList, PList, PowerArray, etc) based on structural induction. Two important constructor operators are used: tie $(p|q)$ and zip $(p\natural q)$, yielding, respectively, the concatenation and interleaving of two similar lists.

Functions are defined based on the structural induction principle. For example, the high order function *map*, which applies a scalar function to each element of a *PowerList* is defined as follows:

$$\begin{array}{l} map : (X \to Z) \times PowerList.X.n \to PowerList.Z.n \\ map.f.\,[a] = [f.a] \\ map.f.\,(p\natural q) = map.f.p \natural map.f.q \text{ or } map.f.\,(p \mid q) = map.f.p \,|\, map.f.q \end{array} \tag{7}$$

In order to increase granularity distribution functions are defined [7]; they depend on the same operator (tie or zip) that the initial non-transformed function uses.

A transformation of a function to accept distributions means that granularity could be increased for that function (program). The transformation for *PowerList* funstions is based on the following theorem.

Theorem

Given

- a function f defined on $PowerList.X.n$ as

$$f.(u|v) = \Phi^f(u, v), \tag{8}$$

  where Φ is an operator defined based on scalar functions and extended operators on *PowerList*s.
- a corresponding distribution $distr^l.p, (p \leq n)$, and
- a function f^p – *the bounded parallel function* defined as

$$\begin{array}{l} f^p.(u|v) = \Phi^{f^p}(u, v) \\ f^p.[l] = [f^s.l] \\ f^s.u = f.u \end{array} \tag{9}$$

then the following equality is true

$$f = flat_1 \circ f^p \circ dist^l.p \tag{10}$$

A similar theorem is proven for cyclic distributions, based on $\natural$ operator too. The proof could be find in [7].

In BMF of lists the granularity analysis is done only for homomorphisms. Bounded parallelism is discussed there only for concatenation operator (so only for linear distribution or block). The advantage of the PARES model is that we may use not only the distributions based on concatenation operator, but also the cyclic distributions, or combinations of distributions.

UNITY. UNITY (unbounded nondeterministic iterative transformations) [3] is both a computational model and a proof system. UNITY model uses Interleaving approach that derives from multiprogramming ideas in operating systems via models of concurrency such as transition systems. If a computation can be expressed as a set of subcomputations that commute, that means, it can be evaluated in any order and repeatedly, then there is considerable freedom for the implementing system to decide on the actual structure of the executing computation. It can be quite hard to express a computation in this form, but it is made considerably easier by allowing each piece of the computation to be protected by a guard, that is, a Boolean-valued expression. Informally speaking, the semantics of a program in this form is that all the guards are evaluated and one or more subprograms whose guards are true are then executed. When they have completed, the whole process begins again. Guards could determine the whole sequence of the computation, even sequentializing it by having guards of the form step = i, but the intent of the model is rather to use the weakest guards.

A UNITY program consists of declaration of variables, specification of their initial values, and a set of multiple-assignment statements. In each step of execution, some assignment statement is selected nondeterministically and executed. For example, the following program,

$$\begin{array}{l}\text{Program P}\\ \text{ initially } x = 0\\ \text{ assign } x := a(x) [] x := b(x) [] x := c(x)\\ \text{end } \{\text{P}\}\end{array}$$

consists of three assignments that are selected non-deterministically and executed. The selection procedure obeys a fairness rule: every assignment is executed infinitely often.

In order to initialize a matrix - $U(n \times n)$ - to unity matrix; three variants could be used:

$$\begin{array}{l}< [] \; i,j : 0 \leq i < n \wedge 0 \leq j < n :\\ \quad U(i,j) := 0 \; if \; i \neq j \; \sim \; 1 \; if \; i = j\\ >\end{array}$$

or

$$\begin{array}{l}< \| \; i,j : 0 \leq i < n \wedge 0 \leq j < n : U(i,j) := 0 >\\ [] \; < \| \; i : 0 \leq i < n : U(i,i) := 1 >\end{array}$$

or

$$
\begin{array}{l}
< [] \; i : 0 \leq i < n : U(i,i) := 1 \\
[] \; < \| \; j : 0 \leq j < n \wedge i \neq j : U(i,j) := 0 > \\
>
\end{array}
$$

The first solution has n^2 statements, one for each matrix element; the second solution has two statements, and the third has n statements, one for each matrix row.

A mapping of a UNITY program expresses a way in which that program can be executed on an architecture; such a mapping may transform the connector [] into sequential or parallel composition [3].

A possible mapping onto a synchronous multiprocessor architecture that we may define is based on the following steps:

: choose one statement based on a certain order;
: execute the components of a statement (multiple assignments); the componets are executed in parallel by using the set of processors that are synchronized before other statement starts.

In this case, the connector [] means sequential composition, and the granularity analysis of the variants that initialize the unity matrix is the following:

: the first solution has the biggest granularity (nothing is executed in parallel);
: the second variant has fine granularity (two statements: the first with n^2 components and the second with n components);
: the third has a medium granularity (n statements each with two sub-statements, and the second sub-statement has n components).

If a mapping on an asynchronous architecture is considered then we have the following steps:

: each statement is allocated to a processor, and
: a sequence of execution (control flow) is defined for each processor (if more than one statement is allocated to a processor).

Statements allocated to different processors are executed in parallel. The components of a statement could be executed in parallel by using a set of processes that are synchronized before other statement starts. If we are able to transform the program in such a way that the number of statements is equal to the number of processors, then the granularity is perfect for the target architecture.

As we may see from the last variant for matrix initialization, a statement could be defined as a multiple assignment or as a composed statement (being composed from two or more sub-statements).

The granularity of a UNITY program can be adjusted by changing the way in which the assignment components are organized in the same or in different assignment statements. An important requirement is that a transformation like this has to preserve the correctness.

So, we may conclude that even if this model is so highly abstract we may change the granularity by interchanging the separation symbol [] with the symbol $\|$, by following some rules that preserve the result.

4. Conclusions

A programming model provides a set of rules or relationships that defines the meaning of a set of programming abstractions. These abstractions are manifested in a programming language which is an instantiation of that programming model. A primary objective is to allow reasoning about program meaning and correctness. Still, since the main reason for using parallelism is achieving performance, cost evaluation is also very important for a parallel computation model. Granularity is a factor that could influence the cost in a great measure. The importance of building granularity from the first stages of parallel programs development justifies the introduction of some mechanisms for specifying granularity even in models with high degree of abstraction.

For serial algorithms, the time complexity is expressed as a function of n – the problem size. The time complexity of a parallel algorithm depends on the type of computational model being used as well as on the number of available processors. Therefore, when giving the time complexity of a parallel algorithm it is important to give the maximum number of processors used by the algorithm as a function of the problem size. This is referred to as the algorithm's *processor complexity*.

The synthesis and analysis of a parallel algorithm can be carried out under the assumption that the computational model consists of p processors only, where $p \geq 1$ is a fixed integer. This is referred to as *bounded parallelism*. In contrast, *unbounded parallelism* refers to the situation in which it is assumed that we have at our disposal an unlimited number of processors. But the next question would be "how we may distribute the work across these processors". The answer is easy if the model allows different granularity scales.

Parallel programs specified using a computational model with high degree of abstraction usually reflect the unbounded parallelism, and so usually we have very fine size of granularity. But, some of these models could be adjusted to accept any scale of granularity. This represents an important advantage since based on this a more accurate cost model could be defined, too.

From the above analysis we may conclude that for the models with high degree of abstraction, which are based on domain decomposition we may define more easily formal methods for building granularity. Another important thing is that these methods have to be formally defined in order to assure the correctness preservation; the main reason of using models is to assure a software development method that assures the correctness, since debugging is extremely difficult in parallel setting.

References

[1] R. Bird. *Lectures on Constructive Functional Programming*. In M. Broy editor, Constructive Methods in Computing Science, NATO ASI Series F: Computer and Systems Sciences, Vol. 55, pg. 151-216. Springer-Verlag, 1988.

[2] M. Cole. *Parallel Programming with List Homomorphisms*. Parallel Processing Letters, 5(2):191-204, 1994.

[3] Chandy, K.M. AND Misra, J. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA.1988.

[4] Foster, I. *Designing and Building Parallel Programs*, Addison-Wesley 1995.

[5] Gorlatch, S., *Abstraction and Performance in the Design of Parallel Programs*, CMPP'98 First International Workshop on Constructive Methods for Parallel Programming, 1998.
[6] Misra, J.: PowerList: A structure for parallel recursion. ACM Transactions on Programming Languages and Systems, Vol. 16 No.6 (1994) 1737-1767.
[7] Niculescu V., *Data Distributions in PowerList Theory*. Lecture Notes of Computer Science, Vol. 3722: Theoretical Aspects of Computing, Proceedings of ICTAC 2007, Springer-Verlag, 2007: 396-409.
[8] Niculescu, V. *PARES - A Model for Parallel Recursive Programs*, Romanian Journal of Information Science and Technology (ROMJIST), Ed. Academiei Romane, Volume 14(2011), No. 2, pp. 159-182, 2011.
[9] D.B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.
[10] Skillicorn, D.B. and Talia, D.: Models and Languages for Parallel Computation. ACM Computer surveys, 30(2): 123-136, June 1998.

Department of Computer Science, Babeş-Bolyai University, Cluj-Napoca
E-mail address: vniculescu@cs.ubbcluj.ro