The case for Globally Irregular Locally Regular Algorithm Architecture Adequation

Pierre Boulet and Ashish Meena
Laboratoire d'Informatique Fondamentale de Lille
Cite scientifique
59 655 Villanelle d'Asce Ceded, France
Email: Pierre.Boulet@lifl.fr, Ashish.Meena@lifl.fr

***Abstract*— In modern embedded systems, parallelism is a good way to reduce power consumption while respecting the real-time constraints. To achieve this, one needs to efficiently exploit the potential parallelism of the application and of the architecture. We propose in this paper a hybrid optimization method to improve the handling of repetitions in both the algorithm and the architecture. The approach is called *Globally Irregular Locally Regular* and consists in combining irregular heuristics and regular ones to take advantage of the strong points of both.**

I. Introduction

When designing Systems-on-Chip (SoCs) for embedded applications, one often has to deal with hard optimization problems. Dimensioning the hardware platform in such a way that the constraints of the system are satisfied is a challenging problem. Indeed, these constraints are functional, but also non functional such as real-time or power consumption related. This problem is called "Algorithm Architecture Adequation" [1].

In many computation intensive applications such as signal processing or consumer multimedia these requirements are particularly strong. Hopefully these applications exhibit parallelism and can benefit from multiprocessor architectures. The goal of this paper is to propose a better way to handle parallelism in the optimization heuristic. We advocate a *Globally Irregular Locally Regular* approach to this problem. All the point here is to exploit as well as possible the repetitions in the algorithm and in the hardware architecture.

After some motivation and related work in Section II, we will present our proposal in Section III and present a small experiment to demonstrate the benefits of this approach in Section IV. We will finally conclude and propose some future works in Section V

II. Motivation

A. Parallelism to reduce power consumption

When designing applications for embedded systems, one often wants to reduce power consumption. The usually admitted formula for power consumption of a SoC is [2]:

$$\alpha C {V_{\text{dd}}}^2 f + I_{\text{off}} V_{\text{dd}}$$

where f represents the frequency of the chip, V_{dd} is the supply voltage. α is a coefficient that takes different values for logic and memory.

From this formula we can deduce that parallelism is a good way to reduce power consumption. Indeed, for a given work amount, W and a given duration (or real-time constraint), τ, one can use a single fast processor of frequency $f_{\text{seq}} = W/\tau$ or n slower processors of frequency $f_n = W/n\tau$. This is in an ideal world where the work can be split in n parts without overhead (communications, etc). As V_{dd} can be lowered when f decreases and appears squared in the power consumption equation, using parallelism allows to decrease the frequency and thus the supply voltage and the power consumption of the chip.

One of the main problems that reduces the efficiency of parallel algorithms is the overhead caused by communications. This is particularly true when considering clusters of workstations or massively parallel processors. Thus, to program efficiently such computers, one needs coarse grain parallelism to be able to overlap communications by computations.

This problem is much less crucial in SoCs because on chip communications are very fast, nearly as fast as computations. Communicating one data-element often takes only one or few cycles. Thus, to be able to overlap communications by computations, small grain computations may be enough.

From these two remarks, we can conclude that parallelism (and even fine grain parallelism) is a promising solution to reduce power consumption in SoCs. Some regular architectures have already appeared on SoCs [3], [4], [5], [6].

B. Available parallelism in applications

There are two sorts of parallelism available in applications: control parallelism and data parallelism. The former is usually modelled using task graphs while the latter corresponds to loops in the code or repetitions in tools such as SynDEx [7].

Irregular heuristics such as list heuristics [8], [9], [10], [11], [12], [13] are good tools to exploit control parallelism. On the other hand, these heuristics don't handle easily data parallelism. These heuristics are general, they can handle a very large class of problems but they may ignore some high level information that is available in their input (loop structure for example).

On the other hand, regular heuristics such as the ones developed in the automatic parallelism domain [14] are much more adapted to deal with data parallelism. These heuristics are well adapted to distribute loops onto regular architectures. Their

main drawback, which is also why they are so performant, is their limited application domain.

Though heterogeneous extensions to these heuristics have been developed [15], we propose here a different approach based on the combination of irregular heuristics and regular ones in the hope to take advantage of the strong points of both and to cancel their weak points.

III. Proposal

We show here a way to enhance irregular heuristics by using regular ones. We call this approach *Globally Irregular Locally Regular* optimization. After the explanation of the proposal (see Section III-A, we will stress the expected benefits of this approach (see Section III-B). In Section IV we will detail an experiment confirming these benefits.

A. GILR heuristic

Our proposal is structured as follows:

1) Extract the regular parts of the application and the hardware.
2) Optimize with a regular heuristic each of the regular application parts onto each of the regular architecture parts.
3) Replace the extracted parts by atomic parts encapsulating them in the description of the application and the architecture.
4) Characterize these atomic parts with the results of the regular optimization.
5) Use an irregular heuristic to optimize this new problem.

Thus the optimization of the irregular task graph is still done using a well suited list heuristic while the data parallel parts of the application have a chance to be scheduled in a regular way onto the regular parts of the architecture. In this way, we hope to be able to combine to strong points of the different optimization techniques.

B. Expected benefits

The expected benefits of such an approach are three-fold: better optimization, reduced optimization time and reduced code size and complexity.

1) Better optimization: Though irregular heuristics such as list heuristics or meta-heuristics can be well tailored to the AAA problem, they are not guaranteed, meaning that we have no way to assess the quality of their result in the general case. On the other hand, many optimality results exist concerning regular heuristics [16]. This alone should lead to optimization results that are, at least, not far from the ones obtained by an irregular heuristic alone, though this can not be guaranteed.

Furthermore, another point in favor of the regular heuristics is that they have the repetition information available whereas the irregular ones forget about it and try to optimize using lower level information.

2) Reduced optimization time: The most impressive benefit of using a regular heuristic is the reduction of the optimization time. This comes from the fact that such heuristics are parametrized. They give their result in function of the size parameters of the problem. So their running time is independent on the size of the data manipulated by the application. On the contrary, list heuristics look at all the elementary operations independently and thus their complexity is factor of the size of the computation domain (that is directly related to the data size). Actually, we will see that the computation time and memory requirements of such heuristics may become overly large for large data sizes.

This computation time reduction is very important because the architecture exploration is often an interactive task and waiting several hours for an adequation result may lead to less thought out decisions.

3) Reduced code size and complexity: Another big benefit of keeping the factorization available for the heuristic is that this heuristic can then generate regular code (containing loops). The nice thing about code with loops is that it is compact and thus takes less place in memory and thus allows for less power consuming architectures.

As the generated communication patterns are regular, communication behavior can be better understood and communication media dimensioning is eased.

All in all, even in the case when the obtained latency is not guaranteed to be better than when using pure irregular heuristics, the GILR method has clear expected benefits in terms of reduced optimization time and reduced code size and complexity. These benefits allow for a more thorough architecture exploration and in the end to less power consuming architectures.

IV. Experiment

A. Problem description

The application we will study here is very simple. It is a matrix-vector product. We will show how it can be mapped on a regular 2x2 processor grid. Several methods will be compared:

- the SynDEx [7] tool that implements a list heuristic (with two ways to handle repetition)
- and a GILR heuristic (here mainly the regular part).

This example is fully regular and stresses the benefits of using the regularity to improve the algorithm architecture adequation.

1) Repetitive Architecture: The architecture we are considering is consists of homogeneous multiprocessors in the form of a 2x2 grid. The interconnections among the 4 processors are point to point (see figure 1). Here every processor is directly connected to the other three processors. The designed architecture is drawn on SynDEx. As said above our designed architecture is fully repetitive i.e there are many instances of same type of processors. We will see in later sections, how we have taken benefit of this repetitiveness in architecture and got good results in comparison with SynDEx generated schedules.

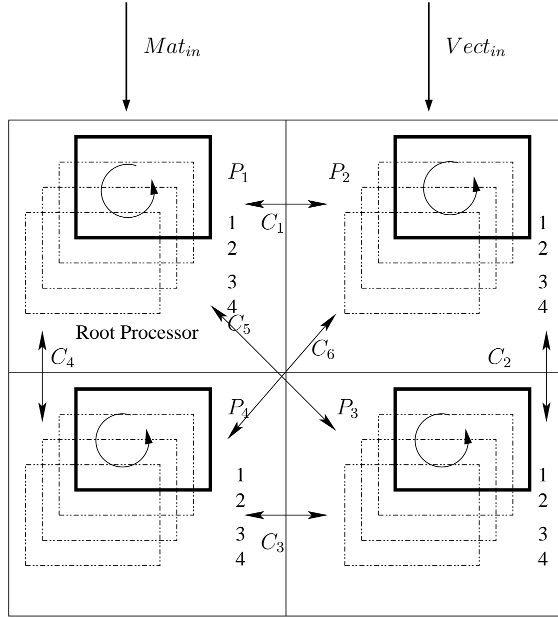

Fig. 1. 2x2 grid connection network

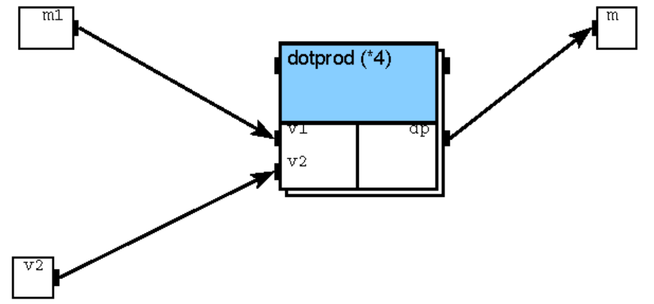

Fig. 2. Dot product modelling in SynDEx

2) Repetitive Computation Application: As an application we have taken the *"matrix vector product"* which comes as example with SynDEx CAD tool. The very choice of this example is well suited, as it presents regular computation on different array data. This application is repetitive in the form of nested loops, as often appear while manipulating multidimensional arrays. For examples such computations can be seen in signal and image processing applications. So, what interests us more in this type of application is to schedule regular repetitive parts of the application on repetitive parts of architecture. Having this as our basic theme we aim at computing task as fast as possible, exploring the data parallelism approach with less possible overhead in communications of data.

So for formulation we can say: *"matrix vector product"* of one matrix $M \in R^{m \times n}$ by vector $V \in R^n$ which gives a resultant vector W=R^n, and can be formalized as follows:

$$W = \left[\sum_{j=1}^{n} m_{ij} v_j \right]_{i=1}^{m} \quad (1)$$

Where m is the number of lines of matrix M, n is the number of columns of M and the size of vector V, m_{ij}: i, j-th element of matrix M, v_j: j-th element of vector V.

As can be seen on equation 1, this computation is repetitive and can be seen as a depth two loop nest. The outer loop iterating over the rows of M and the inner loop computing the dot product of such a row with the vector. This application is modelled this way in SynDEx. So, the staring dependence graph has this two repetition level structure where the outer level is fully parallel while the inner loop is sequential to accumulate the element by element multiplications. Figure 2 shows how this inner level is modelled in SynDEx.

B. Data availability and timing assumptions

To make meaningful comparisons, we take the same assumptions as those in the SynDEx example. Data are coming from sensors, so they are available at a fix cost on the processor connected to the sensor (3 time units). In the same way, output to the actuator of the result takes 3 time units. As shown on figure 1, the matrix is stored on processor P_1 and the vector is stored on processor P_2. The result will be stored on processor P_1.

The computation times on all processors are: addition and multiplication are 2 units each. The communication of an element is 2 units and the communication time is proportional to the data size. So the single processor timing for the computation of the matrix vector timing is $4nm$ for nm additions and nm multiplications.

C. Optimization methodologies

We present here the different heuristics we compare.

1) Heuristic used in SynDEx: The optimization heuristic provided by the AAA methodology implemented in SynDEx allows users to choose between two variants: *NO FLATTEN* or *FLATTEN*. The difference between NO FLATTEN and FLATTEN is as follows.

With FLATTEN, references corresponding to explicitly hierarchical definitions are replaced by the graph of this definition, conditioning and repetitions are expanded. This process is iterated until all vertices's of the transformed graph become atomic (no hierarchy remains). And later it explode all atomic tasks on different processor accordingly to the SynDEx scheduling algorithm (list heuristic) and proposes fine grain optimization. The drawback is that no regularity is seen by the optimization heuristic. Thus the generated code exploits all the additions and all the multiplications causing a code size explosion.

The NO FLATTEN option is an attempt to use the repetition to keep the code size compact and avoid the optimization time explosion caused by the hierarchy flattening. The method is to ignore the explicit hierarchy and keep only the top level of the algorithm graph. This approach leads to a coarse grain optimization that completely disallows the parallilization of repetitive constructs. The result for our simple example is a

mapping of the whole matrix vector product as a unique task on a single processor.

2) Regular optimization: Our aim here is to produce some allocation that could be produced by a regular heuristic. The idea is to make a data dependence analysis of the loop nest. Such an analysis would easily discover that the computation of the dot products of the rows of the matrix by the vector are independent and can be computed in parallel. On the other hand each of these dot products is sequential (if we suppose that in SynDEx, the regular heuristics does not use the knowledge that the addition is commutative and associative). Once the parallelism extracted, the mapping of these sequential dot products onto the processor grid can be done with a clustering heuristic such as tiling. We have chosen here to allocate the rows in a cyclic way to the processors. The communication allocation is then done in such a way to reduce the communication overhead by overlapping at most the communications by computations. The resulting allocation is detailed below.

a) Allocation description: Following our assumptions, matrix M is available on processor P_1 after 3 units. It is the same for vector V on processor P_2. As the vector has to be broadcast to all processors for the first round of dot product computations, this first round is different from the others. For the other rounds, the vector is already present, one only needs to distribute the rows of the matrix to the processors computing their product with V. Thus this is the steady state of the algorithm and can be coded as a loop. The last round has to handle the computation of the $m \mod 4$ last dot products and to gather the results on p_0.

To overlap the communications, all vector and matrix row transfers are pipelined. This allows to begin the computation of the dot product as soon as an element has been received. As the communication delay of one element of M or V takes 2 units and the computation of a multiplication and an addition together takes 4 units, all the data transfers can be overlapped except the first.

The details of the communication scheme is shown on figure 3. On the X-axis we have increasing computation time and on the Y-axis we have available processors and corresponding communication links. The scheduling graph shows coupled data and task scheduling. As you can see how we managed to have earliest possible starting of computation. So to bootstrap the computation on the available processors, one has to exchange data elements. During time steps 3 to 5 the first vector element is broadcast from P_2 to P_1, P_3 and P_4. Every processor has it now. We then have to distribute the first elements of the matrix rows from P_1 to P_2, P_3 and P_4. As the 4 processors are fully connected, this can be done in parallel with the broadcast of the first element of V, except for the communication from P_1 to P_2 as it would use the same link for two communications. That is why P_0 starts to compute 2 time units later than the others. The rest of the first round of dot product computation goes without communication delays as all communications can be overlapped by communication.

For the steady state of the algorithm, V is already available, so only the matrix elements have to be distributed. This is possible without delay and can be done at the end of the previous round so that no time is lost between rounds. One still has to send back all the computed results to P_1 this can be done during the next round without any additional delay except for the last round where the result gathering can be done during the last addition computation on P_1.

Finally, the total processing time for this regular schedule is

$$\begin{aligned} \delta &= 3 + 2 + \left\lceil \frac{m}{4} \right\rceil (4n) + 2 + 3 \\ &= 10 + 4n \left\lceil \frac{m}{4} \right\rceil \quad . \end{aligned} \qquad (2)$$

b) Efficiency of this schedule: This schedule is optimal considering that all processors compute all the time and only wait for data on the critical path of the dependence graph.

This schedule has a number of other advantages concerning resource usage. Indeed, the vector is sent only once to all processors and reused in all other computation rounds and no data communications are generated during one dot product computation. So, the communication network usage is known and well quantified. The code, containing loops is also very compact and so has a low memory usage. It is the same for the accumulator allocation, indeed one can reuse them from one round to the other because their life-time is well defined. This controlled memory usage is very beneficial to power consumption.

D. Simulation results

TABLE I

SIMULATION RESULTS OF MATRIX VECTOR PRODUCT PERFORMED ON 2X2 GRID.

Matrix Size	SynDEx(unit time)		Regular(Unit time)
	Flatten	Non-flatten	
4x4	40	78	26
8x8	98	288	74
16x16	303	1062	266
32x32	1176	4166	1034
64x64	4367	16518	4106

Table I shows the optimization results of the 3 heuristics by unit time taken to execute task on a 2x2 grid. Further, this unit time can be understood as number of cpu cycles taken to execute the given task. So, to clear the confusions, we are using "Unit time" taken.

From this table we can see that the non-flatten heuristic of SynDEx is a very bad choice because it schedules all the computation as a big block on an one processor only and can not use multiple processors. The flatten one quite good with respect to our optimal schedule. Without knowing anything about the repetitive structure of the application nor the hardware platform, it is able to stay no far from the optimal. But the gain on latency is not the most impressive result of the regular heuristic.

Indeed, table II shows the computation time of SynDEx for the flatten heuristic. The workstation used for SynDEx simulation is P4 2.6Ghz, 512DDR RAM. So now, one can imaging what will happen, if array size will be 1024x1024. SynDEx

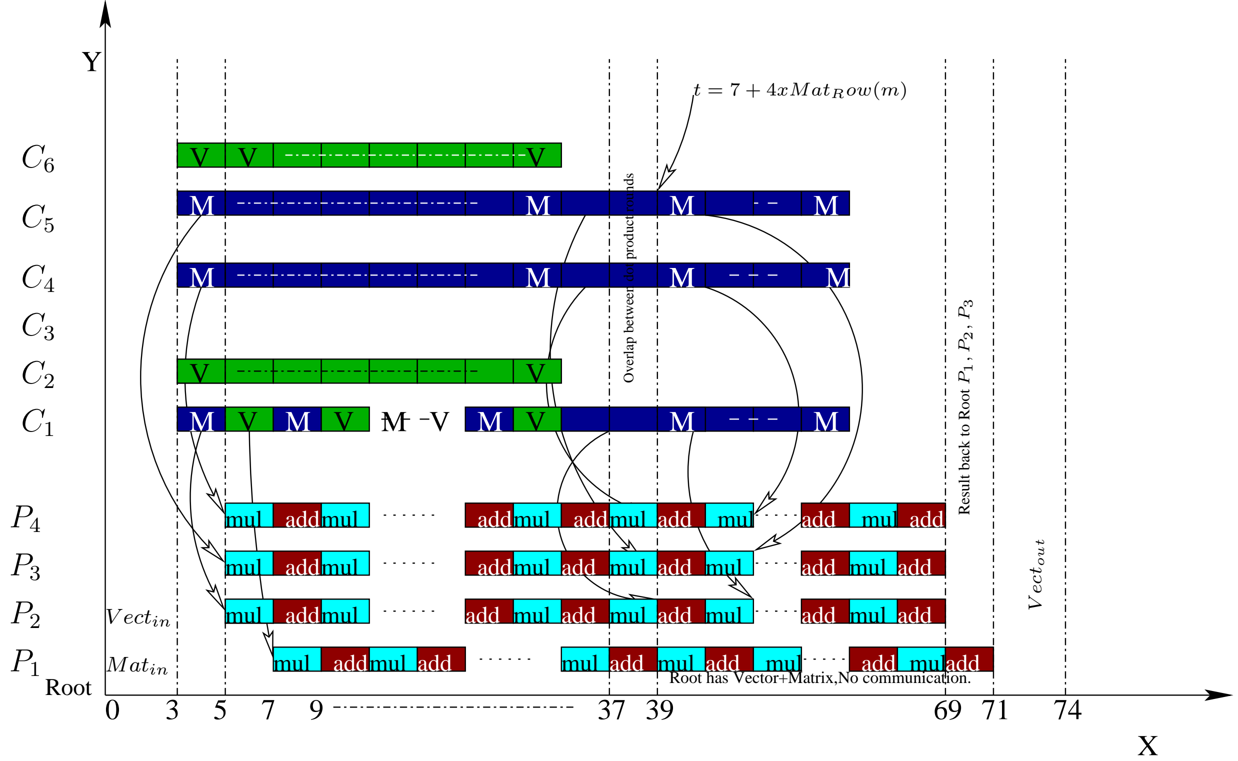

Fig. 3. Proposed regular schedule for Mat-Vect dimension 8X8

TABLE II
TIME CONSUMPTION IN GENERATING THE "FLATTEN" SCHEDULE BY SYNDEX.

Size	Timings
4x4	0.10 s
8x8	0.30 s
16x16	1 min
32x32	25 min
64x64	11 h

will take days to show the schedule. As the computation is regular, it can be parameterized on the size of the matrix. The regular heuristic, being parameterized will always take the same amount of time to give its result. Indeed, it is the same result for all sizes.

The third very big advantage of the regular heuristic is the size of the generated code that is the same for all sizes also. In the case of the flatten heuristic of SynDEx, all the elementary computations are present individually in the code leading a code size that is proportional to the number of operations, namely $2nm$.

So all the expected benefits of using a regular heuristic for the regular parts of the code are shown by this example: better optimization, lower computation time and better resource usage.

V. CONCLUSION

We have proposed in this paper a new way to use efficiently the parallelism that is present in computation intensive embedded application on SoCs with regular hardware units. The idea is to combine irregular heuristics with regular ones to compute a mapping of the application onto the architecture. By this way, we are able to combine the strengths of both kinds of heuristics, namely the versatility of irregular heuristics with the efficiency of the regular ones. The benefits of such an approach are a better optimization, obtained in less time with a better resource utilization.

The potential of this *Globally Irregular Locally Regular* approach has been illustrated by a simple experiment. The expected benefits have been well demonstrated by this experiment, comparing an irregular heuristic coming from the SynDEx tool and a hand-crafted regular one on a matrix vector product. The result show that the irregular heuristic was quite good in terms of optimization result but that regularity handling is crucial concerning resource usage.

More complex experiments need to be done to fully assess the method. In particular, we will formalize the regular heuristic and try it in a systematic way with more complex applications.

References

[1] Y. Sorel, "Massively parallel computing systems with real time constraints - the "Algorithm Architecture Adequation" methodology," in *Proceedings of the 1st International Conference on Massively Parallel Computing Systems*. Los Alamitos, CA, USA: IEEE Computer Society Press, May 1994, pp. 44–54.

[2] ITRS, "Design, 2003 edition," http://public.itrs.net/, 2003. [Online]. Available: http://public.itrs.net/

[3] M. Berekovic and P. Pirsch, "A scalable, distributed netwrk-on-chip architecture for digital signal processing based on simultaneous multithreading(SMT)," in *5th Workshop on Media and Streaming Processor*, San Diego, Dec. 2003.

[4] NEC Corporation, "NEC unveils a new class of system LSI solutions, the dynamically reconfigurable processor LSI architecture, at microprocessor forum," http://www.nec.co.jp/press/en/0210/1601.html, San José, California, 2002. [Online]. Available: http://www.nec.co.jp/press/en/0210/1601.html

[5] picoChip, "PC101 and PC102 datasheets," http://www.picochip.com/technology/picoarray, 2003.

[6] W. Wolf, "The future of multiprocessor systems-on-chips," in *41st Conference on Design Automation (DAC'04)*, San Diego, California, USA, June 2004.

[7] Y. Sorel and C. Lavarenne, *SynDEx Documentation Index*, INRIA, 2000, http://www-rocq.inria.fr/syndex/doc/.

[8] P. Hoang and J. Rabaey, "Hierarchical scheduling of dsp programs onto multiprocessors for maximum throughput," in *Proc. ASAP92 International Conference on Application Specific Array Processors*, Berkeley, California, Aug. 1992.

[9] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for dsp applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, Oct. 1995.

[10] T. Grandpierre, "Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés," Ph.D. dissertation, Université Paris XI, Paris, France, 2000. [Online]. Available: http://www.inria.fr/rrrt/tu-0666.html

[11] N. K. Jha, "Low power system scheduling and synthesis," in *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, San Jose, California, 2001, pp. 259 – 263.

[12] T. Wolf and M. Franklin, "Locality-aware predictive scheduling of network processors," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Tucson, AZ, Nov. 2001, pp. 152–159.

[13] J. Hu and R. Marculesu, "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints," in *Proceedings of the conference on Design, automation and test in Europe*, vol. 1, Paris, Feb. 2004.

[14] A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2000, http://www.birkhauser.com/detail.tpl?isbn=0817641491.

[15] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, "Scheduling strategies for mixed data and task parallelism on heterogeneous clusters," *Parallel Processing Letters*, vol. 13, no. 2, pp. 225–244, 2003.

[16] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien, "Loop parallelization algorithms: From parallelism extraction to code generation," *Parallel Computing*, vol. 24, no. 3-4, pp. 421–444, May 1998.