Analysis of adaptive algorithm's parallelism

J.Sánchez* and H.Barral

ENST, Télécom Paris, Département Signal
46 rue Barrault, 75634 Paris Cedex 13, France.
E–mail: sanchez@sig.enst.fr, Tel:++33 1 45817782, Fax:++33 1 45887935.

RÉSUMÉ

Ce papier analyse le problème de l'implantation d'algorithmes adaptatifs sur des machines parallèles MIMD ayant un faible nombre de processeurs. Le graphe de flot de signaux de l'algorithme et un ensemble de paramètres de l'architecture cible nous permetent de prédire la performance maximale des implantations, prenant en compte les délais dûs aux communications. Finalement, on calcule des limites à l'accélération des implantations ayant un nombre donné de processeurs.

ABSTRACT

In this paper we study the problem of implementing adaptive algorithms in non-massively parallel MIMD message-passing computers. We use the algorithm's signal flow graph and a set of parameters of the target architecture to predict implementation's maximal possible performance, accounting for inter-processor communications. We also detect critical tasks, to which we can apply classical parallelisation techniques in order to enhance performance. Finally, we calculate speedup bounds for implementations with a given number of processors.

I Introduction

In digital signal processing most of the time we have to deal with real-time problems. When complicated algorithms are used and a lot of data has to be processed, conventional processors are not powerful enough, so special Digital Signal Processors (DSP's) must be used. If more computational power is needed we have two choices: a VLSI or a parallel machine implementation. The first choice doesn't allow us to change final specifications and it is cost attractive only if we are expecting to use it in a large scale. The second option is becoming cheaper and easier to develop as new parallel MIMD machines and processors [4] appear. There is still a lack, however, of implementation tools, so implementing algorithms is a difficult task. Furthermore, we usually don't know in advance the performance that we obtain because we can no more consider only the algorithm's arithmetic complexity: communication volume and precedence relationships must be taken into account. In this paper we'll see how to predict implementation's performance from a set of algorithmic and architectural parameters.

II Problem Formulation

Given an algorithm A and a set of architectural restrictions R, we would like to find a parallel machine (i.e. the number of processors and the network's topology) and a task allocation that minimize execution time. This is not an easy problem since we know [5] that the task allocation problem is a NP-complete one. Nevertheless, a lot of heuristics have been developed [6] to find sub-optimal solutions.

In this paper, we restrict this difficult problem in various ways, so as to make it easier to solve. First, we are going to focus in a special kind of algorithms, the adaptive filters, which are *iterative* (figure 2). Second, we are not going to solve the restricted problem from the beginning. We'll find the performance in an *unlimited resource parallel machine* (URPM) and, if this implementation does not fulfill the restrictions, we'll predict the performance in a real machine.

II.1 Algorithm's model

Algorithms will be modeled as a set of sequential tasks running concurrently. Let N_t be the number of tasks in which the algorithm has been partitioned. The granularity of this partitioning is an important factor. *Fine grain* partitionning is well suited to SIMD and VLSI implementations, where each computational unit has a small charge and communication is reduced or local. This approach is used, for example, in [1]. With fine grains we can see all of the algorithm's parallelism and communications. *Medium* or *large grain* partitionning can be used if the number of processors of the target machine is small. Some of the algorithm's parallelism and communications are eliminated in this way, so this kind of partitionning must be carefully used.

Algorithm partitionning and data flow can be represented by a *Signal Flow Graph* (SFG). Nodes, denoted by circles, represent tasks. Edges represent data flow between tasks: we will call them *communication channels*.

Fig. 1, for example, depicts the SFG of the FTF-MC algorithm [2] in several granularities. An edge i-j with a D indicates that task i sends data that will be used in the next iteration by task j. Physically, this is implemented introducing a delay (memory) in the communication channel with a size equal to the communication volume C_{ij}. Inputs to and outputs of the algorithm are represented by edges not coming or not going to another task.

Each task i has an associated set of instructions which, applied to the inputs, produces outputs. According to the specific processor used, these instructions will take T_i CPU cycles. To simplify the analysis, we'll consider that, in every task:

- Computation is made until all inputs have arrived.
- Outputs are sent simultaneously to the next tasks.

II.2 Architecture's model

The target multiprocessor system considered will be a message-passing MIMD machine with N_p homogeneous processors connected in a fixed topology restricted by R. Each processor has a CPU and L_{max} communication units (CU). CPU and CU's share a local memory and work concurrently. A CU links one processor to another in a point-to-point connection. We'll call Γ the set of communication parameters of the machine: $\{t_e, g, t_{ei}, ig, \bar{d}\}$. g is the maximal commu-

*Work supported by Conacyt (Mexique)

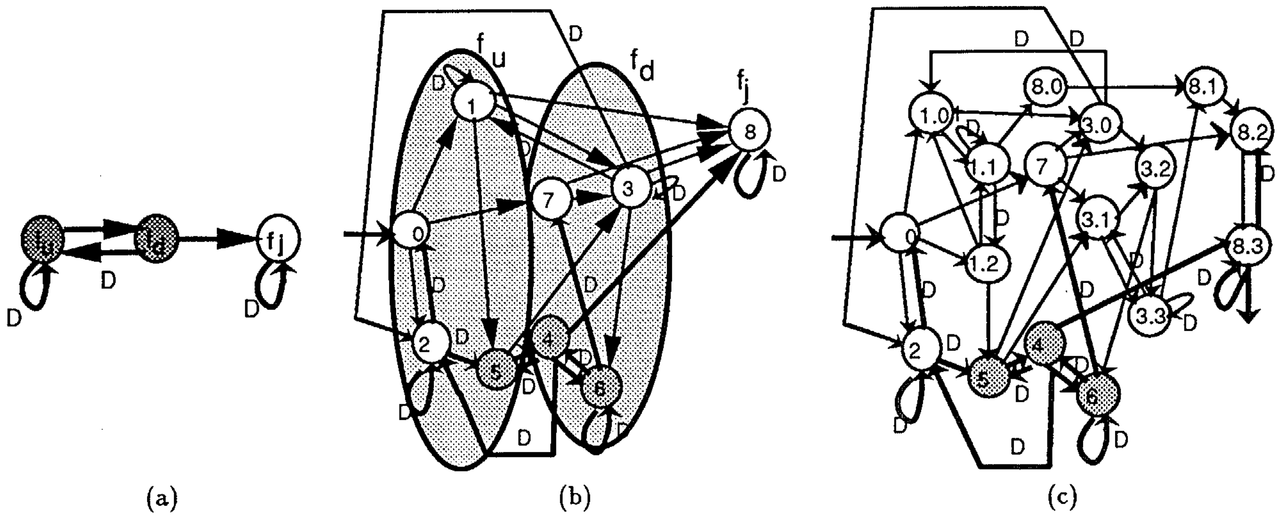

Figure 1: *Different granularities for algorithm FTF-MC, N=1.*

nication rate of a link:

$$g = \frac{CPU\ speed}{external\ link\ speed} \tag{1}$$

Let d_{ij} be the communication time between two tasks i and j. If these tasks are located in neighbor processors (*point-to-point communication*), we'll have:

$$d_{ij} = t_e + gC_{ij} \tag{2}$$

where t_e is the time necessary to establish a communication. If tasks are located in non-neighbor processors (*far communication*), the communication time will depend on the interconnection's network topology. Let $\bar{d}$ be the *average diameter* of the network, i.e. the expected number of processors a message has to cross to arrive to its destination. In this case, the communication time will be:

$$d_{ij} = \bar{d}(t_e + gC_{ij}) \tag{3}$$

Finally, if tasks are located in the same processor (*internal communication*), we'll have:

$$d_{ij} = t_{ei} + ig \times gC_{ij} \tag{4}$$

where t_{ei} is the time needed to establish an internal communication and ig is the rate between internal and external communication speeds:

$$ig = \frac{external\ link\ speed}{internal\ link\ speed} \tag{5}$$

The values of t_{ei} and ig depend on the way we program the internal communications. If we use Occam, for example, communication between tasks located in the same processor will require a data transfer in memory. A more efficient way to implement internal communications is by making the communicating tasks use the same memory locations: ig will be then null because no transfer will be made. From now on, we'll consider this last case.

III Analysis of parallelism

Several quantities have been proposed in the literature to measure implementation's performance [7, 8]. A classical one is *speedup*, defined as the ratio between algorithm's execution time in a sequential and a parallel machine. We can slightly modify this definition for iterative algorithms:

$$s(n) = \frac{IP(1)}{IP(n)} \tag{6}$$

where $IP(x)$ is the implementation's *iteration period* [3, 1] in a parallel machine with x processors, defined as the time between the end of two successive iterations (figure 2). In real-time processing IP must be smaller than the sampling rate T.

We'll show in the next section how to predict the iteration period bound (IPB) of an algorithm implemented in an unrestricted resource parallel machine (URPM). When we use IPB in (6) and we consider zero communication delays, we obtain the *algorithm's maximal speedup* [7], which represents the average number of busy processors in the URPM.

III.1 Critical loop

Let's define the distance between tasks i and j as the time elapsed from the beginning of the computation of task i to the end of the data transfer to task j, i.e. $T_i + d_{ij}$. The total distance in a loop B of the SFG is then:

$$T_B = \frac{\sum\limits_{i,j\in B} T_i + d_{ij}}{\sum\limits_{i,j\in B} z_{ij}} \tag{7}$$

where z_{ij} is the number of delays in edge i-j. Note that this distance depends on the SFG, and in the architectural and implementation parameters.

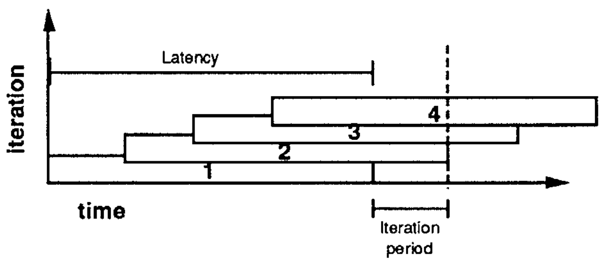

Figure 2: *Iterative algorithm's latency and IP.*

It has been shown [9], in the context of electrical circuits, that the minimal output period (i.e. the inverse of the maximal sampling rate) of a signal entering a circuit G is equal to the *critical loop* distance $\max_{B\in G} T_B$. This result can be extended to MIMD implementations if we apply equation (7) in the algorithm's SFG and we choose the same conditions than in electrical circuits, i.e. every task and communication channel is able to work in parallel. This condition is fulfilled in the URPM, when we allocate each task in a separate processor and each channel in a separate CU, i.e. when the machine graph is *isomorphe* to the SFG: we'll call

Task i	Variable	T_i (C40's CPU cycles)	C_{ij}
$(A,\alpha^{-1},C_{M+1},\nu_{M+1}^{-1}) = f_u(A,\alpha^{-1},C,\nu^{-1},U)$			
$k.0$	aux	$4M+5$	8
$k.1$	$\nu_{M+1}^{-1},\alpha^{-1}$	26	4
$k.2$	A	$5M+16$	$4M$
$k.5$	C_{M+1}	$5M+6$	$4M$
$(B,\beta,C,\nu^{-1}) = f_d(B,\beta,C_{M+1},\nu_{M+1}^{-1},U)$			
$k.7$	aux	$4M+5$	4
$k.4$	C	$5M+6$	$4M$
$k.3$	ν^{-1},β	42	4
$k.6$	B	$5M+6$	$4M$
$(W,\epsilon) = f_j(W,C,\nu,d,U)$			
8	W,ϵ	$9M+14$	$4M$

Table 1: *Tasks of the k-th channel in the FTF-MC algorithm.*

this implementation M_1. The *iteration period lower bound* of M_1 is equal to:

$$IPB_1(\Gamma) = \max_{B\in SFG} \frac{\sum\limits_{i,j\in B} T_i + t_e + gC_{ij}}{\sum\limits_{i,j\in B} z_{ij}} \tag{8}$$

where we have used the fact that each task is executed in a different processor to substitute (2) in (7) for point-to-point communications. M_1 is a realistic implementation in an URPM as we take into account communication delays.

IV Task clustering

We can sometimes obtain a better implementation than M_1 trading communication time by parallelism. In an MIMD implementation we have to deal with these opposite factors: tasks should be allocated in the same processor (*task clustering*) to minimize communication time and, on the other hand, tasks should be allocated in different processors to allow them to work in parallel.

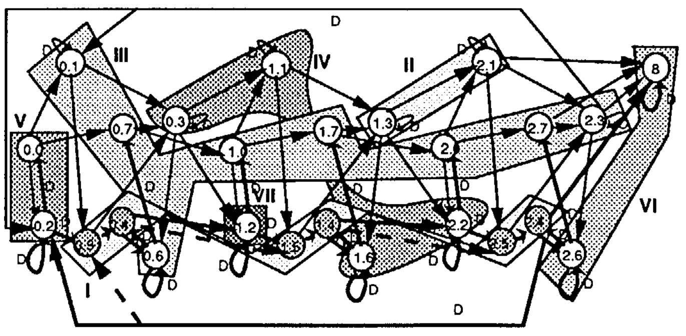

Figure 3: *SFG for the algorithm FTF-MC (N=3) and its M_2 implementation with $N_c = 7$ processors.*

There are several ways to cluster tasks. We've already seen one, namely changing SFG's granularity. In figure 1c, for example, tasks 1.i, 3.i, and 8.i have been put together to form tasks 1, 3, and 8 respectively (figure 1b). Unfortunately, we don't still know how to change granularity to obtain better implementations. We present here a technique to cluster tasks, based in the critical loop concept and in the algorithm used in [10] to schedule iterative algorithms in a parallel machine.

The SFG allows us to see precedence relationships between tasks. According to our data flow model (Section II.1), if an edge exists between tasks i and j, task j can't begin execution until task i ends. They can't run in parallel, even in the URPM, so we won't lose parallelism if we place these tasks in the same processor. Furthermore, this

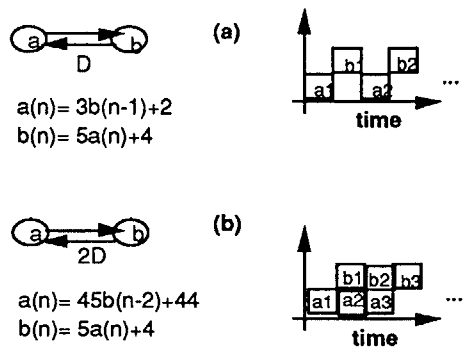

Figure 4: *Execution of loops in the UPRM.*

will eliminate communication time between them because we'll have internal rather than external communications.

As the *IPB* depends on loop distances, we must focus on critical loops. We can place the tasks of a loop in the same processor without losing parallelism and decreasing loop distances at the same time because of the reasons stated formerly, as we show in figure 4a. We'll call M_2 the implementation using N_c processors obtained after task clustering of M_1.

We must point out that, in order to perform task clustering, we must have, as stated in [10], a *perfect rate* signal flow graph, i.e. a SFG in which all loops have exactly one delay. We show in figure 4b why this restriction is introduced: we can't place tasks of a loop having more than one delay in a processor as we'd loose parallelism.

The M_2 iteration period bound can be predicted from the SFG as:

$$IPB_2(\Gamma) = \max_{B\in SFG} \left\{ \frac{\sum\limits_{i,j\,diff\,cluster\in B} T_i + t_e + gC_{ij}}{\sum\limits_{i,j\,diff\,cluster\in B} z_{ij}} + \frac{\sum\limits_{i,j\,same\,cluster\in B} T_i + t_{ei} + ig \times gC_{ij}}{\sum\limits_{i,j\,same\,cluster\in B} z_{ij}} \right\} \tag{9}$$

where we have substituted (4) as the communication time between tasks in the same cluster.

The M_2 implementation is a good compromise between an implementation with maximum parallelism, represented by M_1 and the one with no communication delays, when $t_e = g = 0$ (Γ_0):

$$IPB_1(\Gamma_0) \le IPB_2(\Gamma) \le IPB_1(\Gamma) \tag{10}$$

To show the use of these techniques, we'll analyze the implementation of the FTF-MC algorithm [2] in a network of TMS320C40 processors. Figure 3 depicts the SFG for a number of channels $N = 3$, and table 1 the task computation charges and the communication volume for a filter of order M. Critical loop of M_1 implementation will be, for any N, 0.5-0.4-1.5-1.4-...-(N-1).5-(N-1).4-0.5, with an iteration period bound of:

$$\begin{aligned} IPB_1(\Gamma) &= \frac{N(T_{k.5} + d_{k.5,k.4} + T_{k.4} + d_{k.4,(k+1).5})}{\sum_{k=0}^{N-1} z_{k.5,k.4} + z_{k.4,(k+1).5}} \\ &= N(18M + 120) \end{aligned}$$

where we have substituted the communication values of the C40 processor [4]: $t_e = 54$, $g = 1$.

The task clustering algorithm gives the clusters shown in boxes in figure 3, and an *IPB* of:

$$IPB_2(\Gamma) = \frac{N(T_{k.5} + T_{k.4})}{\sum_{k=0}^{N-1} z_{k.5,k.4} + z_{k.4,(k+1).5}} = N(10M)$$

If we compare this time with the sequential one, we obtain the M_2 implementation's speedup:

$$s(N_c) = \frac{N(37M+126)}{N(10M)} \approx 3.7 \quad for\ big\ M \tag{11}$$

V Predicting implementation performance

We have seen that it is possible to predict the performance of M_1 and M_2 implementations from the algorithm's SFG. Unfortunately, we cannot fix the number of processors they use.

In this section we'll see how to calculate iteration period bounds for implementations with a given number of processors. We'll do this by using the *parallelism profile* of the M_2 implementation, defined in [8] as the plot of parallelism degree, i.e. the number of busy processors during execution, versus time.

The *activity histogram*, obtained from the parallelism profile, shows the time percentage p_k when k processors are busy in the M_2 implementation for $k = 0 \ldots m$.

To calculate IP bounds we'll consider the following hypothesis:

H_1: **M_2 is the fastest parallel implementation of an SFG's algorithm.** Being an implementation in the URPM using all of the available parallelism in the SFG, this hypothesis is often verified if communication delays are not excessive.

H_2: **If an M_2 implementation uses k processors to perform a work in a time IP_k:**
- An implementation with $N_p > k$ processors won't do better.
- An implementation with $N_p < k$ processors will take, at least, a time $\frac{k}{N_p} IP_k$ to perform the same work.

Using hypothesis H_1 and H_2 we can find *iteration period bounds* for $N_p < N_c$:

$$IP(N_p) \geq IP_0(N_p) + IPB_2(\Gamma)\left\{\sum_{k=1}^{N_p} p_k + \sum_{k=N_p+1}^{m} \frac{kp_k}{N_p}\right\} \tag{12}$$

where IP_0 is the time when all processors are waiting for communication. We have separated it because we can't use hypothesis H_2 as it would mean that the communication time doesn't change as the number of processors decreases. IP_0 is a fraction of the total communication time:

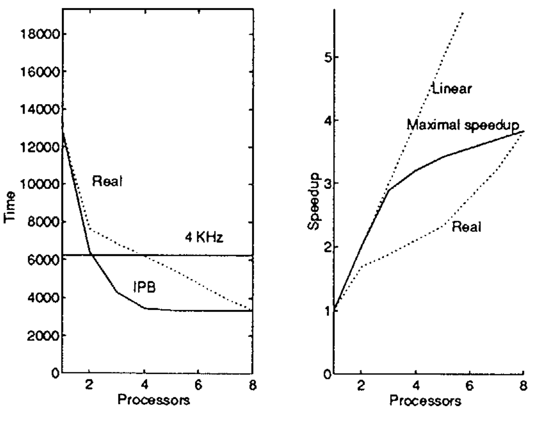

Figure 5: *a) Iteration period and b) Speedup bounds of the FTF-MC algorithm, N=3, in a C40's network.*

$$IP_0(N_p) = f(N_p) \sum_{i,j \in diff cluster} t_e + gC_{ij} \tag{13}$$

$f(N_p)$ is the *superposition factor* between execution and communication time in the parallel machine, bounded by:

$$f(1) = 0 \leq f(N_p) \leq f(N_c) = \frac{IP_0(N_c)}{\sum_{i,j \in diff cluster} t_e + gC_{ij}} \tag{14}$$

As variations of $f(N_p)$ depend on the algorithm, we can try several bounds:

H_3: **(Optimistic).** $IP_0(N_p) = IP_0(N_c)$.

H_3: **(Pesimistic).** $IP_0(N_p)$ changes linearly according to (14) and (13).

Using equations (12...14) and (6), we can calculate the iteration period and speedup bounds for an implementation with a given number of processors, as we show in figure 5 for our example. We can see that speedup bounds calculated using hypothesis $H_1 \ldots H_3$ are, even in the optimistic case, a lot smaller than the generally used linear speedup bound. They are also more realistic, as we can see by comparing these bounds with real iteration periods found when using several allocation algorithms [6].

IP bounds can be useful to predict the minimum number of processors needed to run an algorithm in real-time. In figure 5a, the horizontal line indicates the maximum number of CPU cycles allowing a real-time implementation with a given frequency sampling. We can see that we need at least two C40's to make a real-time implementation of the SFG in figure 3 when $f_s = 4000$ Hz. If we want to implement the algorithm using a greater sampling frequency, we must modify the SFG's critical tasks to show more parallelism.

VI Conclusion

We have presented a method to predict the maximal performance of implementations in parallel MIMD machines from the algorithm's SFG and a set of architectural parameters. Approximations are also proposed to predict performance when we have a fixed number of processors. This enables us to have realistic bounds in the performance of parallel implementations. Critical tasks are identified in the analysis, so we know where we have to parallelize the algorithms.

References

[1] P.R.Gelabert, T.P.Barnwell III, "Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs." *IEEE Trans. on Signal Processing*, Vol. SP-41, pp.858-888, Feb 1993.

[2] D.Slock, L.Chisci, H.Lev-Ari and T.Kailath, "Modular and numerically stable fast transversal filters for multichannel and multiexperiment RLS." *IEEE Trans. on Acoustics, Speech and Signal Processing*. April 1992, vol.40, no.4, pp. 784-802.

[3] J.Silva, "Contribution à l'optimisation d'implantation parallèle d'algorithmes itératifs récursifs. Application au filtrage adaptatif en traitement du signal." Thèse de doctorat de l'Université Bordeaux I. 1992.

[4] "TMS320C4x. User's guide," Texas Instruments. 1992.

[5] S.Bokhari. "On the mapping problem." *IEEE Trans. on Computers*. Vol. C30, no.3, march 1981, pp. 207-214.

[6] F.Andre et J.Pazat. "Le placemente de tâches sur des architectures parallèles." *Technique et science informatique*. vol.7, no.4, 1980. pp. 385-401.

[7] D.L.Eager, J.Zahorjan and E.D.Lazowska. "Speedup versus efficiency in parallel systems." *IEEE Trans. on computers*, vol.38, no.3, march 1989, pp. 408-423.

[8] X.Sun and J.L.Gustafson. "Toward a better parallel performance metric." *Parallel computing*. 1991, no.17, pp. 1093-1109.

[9] M.Renfors, Y.Neuvo. "The maximum sampling rate of digital filters under hardware speed constraints." *IEEE Trans. on Circuits and Systems*. Vol. CAS-28, no.3, march 1981, pp. 196-202.

[10] K.Parhi, D.Messerschmitt. "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding." *IEEE Trans. on Computers*. Vol. 40, no.2, february 1991, pp. 178-195.