Massively Parallel Computing Systems with Real Time Constraints The "Algorithm Architecture Adequation" Methodology

Y. Sorel
INRIA, Domaine de Voluceau, Rocquencourt
B.P.105 – 78153 LE CHESNAY CEDEX – FRANCE
email: yves.sorel@inria.fr

Proc. Massively Parallel Computing Systems, the Challenges of General-Purpose and Special-Purpose Computing – Ischia Italy, May 1994.

Abstract

Massively Parallel Computing Systems (MPCS) provide high performance computing generally used to accelerate numerical computation applications. We present a methodology called "Algorithm Architecture Adequation" used to take advantage of the computation power of these systems in the case of real-time applications. With this methodology, application algorithm as well as MPCS are specified with graphs, then the implementation of an algorithm on a MPCS in respect with real-time constraints may be formalized in terms of graphs transformations. This allows to optimize, taking into account inter-processor communications which are critical, the real-time performances of the implementation. As a result, real-time distributed executives are produced automatically without dead-lock and with minimum overhead. This reduces drastically the development cycle of real-time applications running on MPCS.

1 Introduction

Massively parallel computing systems are mostly used in the field of number crunching in order to accelarate the execution of numerical computation programs. However, since they provide high performance computing, it should be interesting to use these systems under real-time constraints.

We intend to show how it is possible to exploit the potential power brought by these systems in a real-time environment. In this peculiar domain of MPCS, there are two main issues. In one hand, such systems must be able to handle external events coming from environment through sensors (analog or digital) and to produce output events as a reaction to input events, through actuators. In an other hand, because a lot of processors are cooperating in MPCS, communications are critical and must be efficiently supported in order to minimize the overhead they introduce.

Typical applications requiring large amount of computations as well as interactions with their environment under real-time constraints, may be found for example in the following domains: biomedical (EEG and ECG processing...), telecoms (modems, echo cancelling, telephone network regulation...), automobile (engine control, ABS...), aerospace (aircraft and spacecraft control ...), process control (industrial and power plant regulation...), arms systems (radar, sonar, multi-sensor fusion...). These applications belong to the field of process control, image and signal processing, that we will call later in a broad sense, signal processing.

Each *application* is composed of a computer based *system* and of the *environment* it interacts with. The system itself is composed of a *hardware* part executing a *software* part. Finally the software part includes an application program, coding an *algorithm* independently from hardware considerations, and an *executive* allocating hardware resources in order to execute the application program. The system is said to be *reactive* because, in order to keep control over its environment it must interact permanently with it, i.e. it must produce reactions to every stimulus coming from the environment. If not already discrete, input stimuli are sampled and called *input events* and reactions generate *output events*. Strictly speaking the system is said to be *real-time*, when reactions are produced within *bounded delays*.

Although new monoprocessor architectures (workstations), provide ever increasing computation power, they cannot cope with the ever increasing complexity of some signal processing applications. Parallel ar-

chitectures that we call *multiprocessor* are needed for such applications. In this case, it must be pointed out that the executive, which supports the execution of the application algorithm, has more resources to handle. Therefore it induces more overhead which reduces the power benefit brought by multiprocessing. That is the reason why the application developper must be particularly careful for the choice and the use of the executive.

This paper is organized as follows. We first present the graph models used to exhibit both the *potential parallelism* of the algorithm and the *available parallelism* of the multiprocessor. We focus on the multiprocessor model called *Macro-RTL* in reference with Register Tranfer Languages [9] introducing a new level for hardware description. This level is accurate enough to represent the architecture features necessary to modelize and optimize the implementation. This one, which consists in *distributing* and *scheduling* the algorithm on the multiprocessor, is formalized in terms of graphs transformations. In order to obtain an efficient implementation that we call "Algorithm Architecture Adequation", A^3 for short, we state and give a solution for a performance optimization problem. Such an adequation is performed by choosing among all the possible graphs transformations one which optimizes the real-time performances. Real-time performances are mainly characterized by the input data rate and the response time, both expressed in our models in terms of critical path calculi. We present briefly a response time optimization heuristics which is a "schedule flexibility" based greedy algorithm, upgraded to take into account interprocessor communication delays and critical path increase. At this point, a real-time distributed executive, which is the result of graphs transformation, may be built from an *executive generic kernel*. The global approach, from algorithm and architecture specifications to optimized executives generation, formalized with the graph oriented methodology is also called A^3. Finally, before concluding, we describe SynDEx, an interactive graphical software implementing the A^3 methodology.

2 Algorithm Model

In order to take advantage of the available parallelism of the multiprocessor, it is necessary to exhibit the potential parallelism of the algorithm. This may be obtained either directly from a data-flow program or extracted by data-dependence analysis of a sequential program.

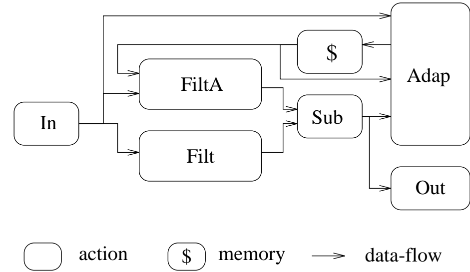


Figure 1: Software graph example

The algorithm is therefore modelized by a *conditioned data-flow graph* (oriented hypergraph), that we call *software graph*, where each vertice is an *action* and each edge is a *data-flow*. This is a generalization of the data-flow graph model defined by Dennis in [4], extended with three kinds of vertices. The first one, that we call *delay*, exhibits the initial tokens, that is the state memory of the algorithm. The two other ones are used in combination to condition a subgraph by a boolean data-flow [2]. Hence, each action stands for either a computation or a conditionning or a memory or an external I/O whereas each edge stands for an iterative data transfer which induces a data dependence between the producing and consumming actions it connects. Moreover, the inputs of a computation action must precede its outputs (output values depend only on input values, there is no internal state variable and no other side effect), this involves also a dependence. A memory action holds data in sequential order between iterations, its output precedes its input. Therefore a memory action may be seen as two separate vertices, one with no predecessor and the other with no successor, then a graph with cycles only through memory vertices may be considered *acyclic*. Actions with no predecessor stand for the external input interface handling the events produced by the environment. Symetrically the actions with no successor stand for the external output interface generating the reactions to the events. The dependences of the acyclic graph induce a partial order on its actions. This model exhibits the potential parallelism of the algorithm.

3 Multiprocessor Model

The multiprocessor, of SIMD, MSIMD, MISD, MIMD or SPMD type according to the "well-known" classification, is a processor network modelized by a

non oriented hypergraph, that we call *hardware graph*, where each vertice is a processor and each edge is a physical bidirectionnal communication *link* allowing data transfers between the memories of the processors connected to the link.

Since we intend to optimize the implementation of an algorithm on a multiprocessor, we need a model of the multiprocessor, accurate enough to predict its real-time behavior, but macroscopic enough to avoid unnecessary details of second order importance. For example, fetch-decode pipelines may not be neglected when the execution duration of an isolated instruction is considered, but they are of second order importance when a sequence of instructions is considered, because the sequence has an integrator effect.

At our macro-RTL level, a processor is composed of several *execution units* sharing a *local memory*, through which inter-unit communication and synchronization are performed. Each execution unit is an automaton with its own state memory (private registers). Each transition of such automata consumes and/or produces data in *macro-registers*. A macro-register is a set of contiguous memory elements which size depends on the type of the macro-register (integer, real ...) and on the size of the memory element for each architecture (8 bits, 32 bits ...).

Among these automata, some are used to control the others: we call them *macro-sequencers* because they generate commands for the controlled automata to initiate transitions. These commands are produced by decoding *macro-instructions* fetched by the macro-sequencer from a program memory. Moreover, state transitions of a macro-sequencer automaton may be conditionned by the state of the controlled automata (conditionnal jumps or interrupt arbitrations for example).

In practice, at a lower level, several processor clock cycles may be required to perform a macro-instruction because the corresponding automaton transition may involve in turn several internal transitions. These internal transitions may be the result of the execution of a single native instruction of the processor or the result of the execution of a subroutine. Therefore, the execution duration of a macro-instruction depends on the type of the execution unit which executes it.

The macro-sequencer, as well as the local memory of each processor and the physical inter-processor communication links, are each shared among several units. An *arbiter* is necessary to schedule the use of each of these shared resources. These arbitrations may induce waiting delays which increase the execution duration of macro-instructions. At the macro-RTL level, these delays are integrated on the duration of the macro-instruction. This is modelized by a square matrix that we call *interference matrix*, where each element (i, j) is a coefficient proportional to the slow-down of the unit i when the unit j is active. When the arbitration is equitable between two units i and j, the coefficients (i, j) and (j, i) are equal. To characterize an architecture, it is therefore sufficient to measure separately these coefficients and the execution duration of macro-instructions with no interference [5].

Execution units may be divided in four main categories:

- *computation units* execute computation macro-instructions coding computation actions of the software graph
- *I/O units* execute input and/or output macro-instructions, one for each I/O action of the software graph, to read data from sensors or write data to actuators
- *communication units* execute communication macro-instructions, one for each communication action of the distributed software graph, to transfer data between macro-registers located on different processors
- macro-sequencer units execute conditionning macro-instructions coding conditionning actions of the software graph and sequence other types of macro-instructions

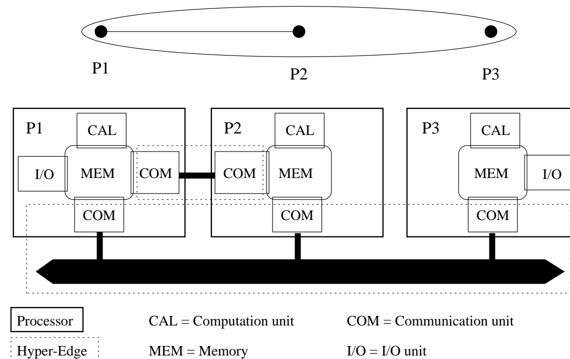


Figure 2: Hardware graph example

Note that there are multiple memory spaces in a multiprocessor:

- the state memory of each automaton is not visible in our macro-RTL model because it is accessed only inside macro-instructions

- the local data memory shared by the execution units of a processor is visible through macro-registers
- the global memory shared between several processors (there may be several global memory spaces) is modelized by a degenerated processor with no computation unit

In order to illustrate our macro-RTL model, here are two examples. The Transputer may be decomposed into one computation unit, five communication units (four full-duplex links and the memory bus which may be used for globally shared memory communications or for communications with I/O units), and only one macro-sequencer.

The TMS320C40 may be decomposed into one computation unit, eight communication units (six half-duplex links and two memory buses which may be each used for globally shared memory communications or for communications with I/O units), and only one macro-sequencer.

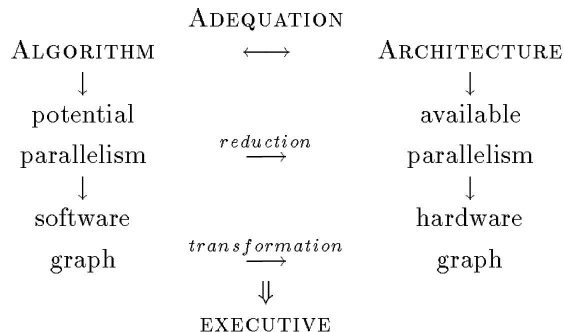
4 Implementation Model

In this paper, we assume that each processor has only one computation unit and only one macro-sequencer, as in the examples of the Transputer and of the TMS320C40, and no more than one I/O unit. Moreover, we assume that each processor has enough program and data memory to ignore memory allocation problems in the implementation.

The graph models used for the algorithm and the architecture specifications lead to a formalization of the implementation in terms of graphs transformations. The optimization aspects of the adequation between both graphs will be the subject of the next section. Whatever type of optimization, the implementation consists in reducing the potential parallelism of the software graph into the available parallelism of the hardware graph. This graphs transformation consists in:

- distributing the actions on the processors (computation actions are assigned to the computation unit of the processor, conditionning actions are assigned to the macro-sequencer of the processor, and I/O actions are assigned to the I/O unit of the processor), this leads to inter-processor communications which are themselves distributed on the inter-processor links (communication actions are assigned to the set of communication units which are connected to the link and cooperate to transfer data between the local memories of their respective processors)
- scheduling the actions which have been assigned to a processor and scheduling the communications which have been assigned to a link

From this graphs transformation may be automatically generated a real-time distributed executive allowing the execution of the algorithm on the multiprocessor. This is discussed in section 6.



More formally, the distribution consists in partitionning the software graph in as many partition elements as there are processors. The edges crossing the partition boundaries imply inter-processor communications. Though the hardware graph is inevitably connex, each processor is not necessarily directly connected to all the others. In order to allow communications between non directly connected processors, the hardware graph is first transformed into a completely connected graph where the vertices are the same as in the hardware graph (the processors) and where the edges are all the acyclic paths of the hardware graph, called *routes*. Therefore any communication between any two processors will be assigned to a route.

The hardware graph is a pair $(\mathcal{P}, \mathcal{L})$ where $\mathcal{P}$ denotes the set of processors and $\mathcal{L}$ the set of inter-processor physical communication links. We then denote $\mathcal{R}$ the set of all the paths in $(\mathcal{P}, \mathcal{L})$. Note that routes, as links, are bidirectionnal. We call *routing* this transformation of the hardware graph into a completely connected hardware graph:

$$(\mathcal{P}, \mathcal{L}) \stackrel{routing}{\longrightarrow} (\mathcal{P}, \mathcal{R})$$

The software graph is a pair $(\mathcal{A}, \mathcal{D})$ where $\mathcal{A}$ denotes the set of actions and $\mathcal{D}$ the set of inter-action data-dependences. We then denote $\mathcal{A}_p$ the set of the actions assigned to $p \in \mathcal{P}$, by $\mathcal{D}_p$ the set of the data-dependences between actions belonging to $\mathcal{A}_p$ and by

$\mathcal{D}_r$ the set of the inter-processor data-dependences generating communications assigned to $r \in \mathcal{R}$. We call *distrib* the distribution of the software graph onto the routed hardware graph and call $\mathcal{G}_{d\mathcal{R}}$ the resulting graph:

$$((\mathcal{A},\mathcal{D}),(\mathcal{P},\mathcal{R})) \stackrel{distrib}{\longrightarrow} \mathcal{G}_{d\mathcal{R}}$$

where

$$\mathcal{G}_{d\mathcal{R}} = \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \mathcal{D}_p), \bigcup_{r \in \mathcal{R}} \mathcal{D}_r \right)$$

In order to describe more accurately inter-processor communications, each $d_r \in \mathcal{D}_r$ is subsituted by a linear graph with as many vertices as there are links on the route. We call *com* this transformation:

$$\forall r \in \mathcal{R}, \quad \forall d_r \in \mathcal{D}_r, \quad d_r \stackrel{com}{\longrightarrow} (c_p, a_l, c_{p'}, \ldots a_{l''}, c_{p''})$$

Each such new vertice a_l, that we call *communication action*, corresponds to a data transfer between the memories of two directly connected processors. Actually all the processors connected to the physical link cooperate (in a way that depends on the hardware) to execute the transfer, so we can consider that each communication action is distributed over the communication units sharing the link. Therefore the new edges c_p are intra-processor but inter-units (computation-communication or communication-communication or communication-computation).

By grouping all the a_l of a same $l \in \mathcal{L}$, and all the c_p of a same $p \in \mathcal{P}$, we obtain the sets $\mathcal{A}_l$ and $\mathcal{C}_p$. Then *com* transforms $\mathcal{G}_{d\mathcal{R}}$ into $\mathcal{G}_{d\mathcal{L}}$ where:

$$\mathcal{G}_{d\mathcal{L}} = \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \mathcal{D}_p \cup \mathcal{C}_p), \bigcup_{l \in \mathcal{L}} \mathcal{A}_l \right)$$

The transformations *routing*, *distrib* and *com* do not modify the software graph partial order $\mathcal{D}$. On the computing unit of each processor, a scheduling of the actions is a total order $\bar{\mathcal{D}}_p$ which includes the partial order restricted to $\mathcal{A}_p$, that is $\mathcal{D}_p$. Identically on each link, a scheduling of the communication actions is a total order $\bar{\mathcal{D}}_l$ which includes the partial order restricted to $\mathcal{A}_l$. We call *sched* this transformation and $\mathcal{G}_s$ its result:

$$\mathcal{G}_{d\mathcal{L}} \stackrel{sched}{\longrightarrow} \mathcal{G}_s$$

where

$$\mathcal{G}_s = \left(\bigcup_{p \in \mathcal{P}} (\mathcal{A}_p, \bar{\mathcal{D}}_p \cup \mathcal{C}_p), \bigcup_{l \in \mathcal{L}} (\mathcal{A}_l, \bar{\mathcal{D}}_l) \right)$$

In short, the implementation of an application algorithm on a multiprocessor is the composition of the four above described transformations:

$$(\mathcal{P},\mathcal{L}) \stackrel{routing}{\longrightarrow} (\mathcal{P},\mathcal{R})$$

$$((\mathcal{A},\mathcal{D}),(\mathcal{P},\mathcal{R})) \stackrel{distrib}{\longrightarrow} \mathcal{G}_{d\mathcal{R}} \stackrel{com}{\longrightarrow} \mathcal{G}_{d\mathcal{L}} \stackrel{sched}{\longrightarrow} \mathcal{G}_s$$

We call this global graph transformation *dist/sched*:

$$((\mathcal{A},\mathcal{D}),(\mathcal{P},\mathcal{L})) \stackrel{dist/sched}{\longrightarrow} \mathcal{G}_s$$

We present now a basic example of graphs transformations involving the software graph of fig.3 and the hardware graph of fig.4. First the routes $R_1 = (L_1)$, $R_2 = (L_2)$, $R_3 = (L_3)$, $R_4 = (L_1, L_2)$ are built.

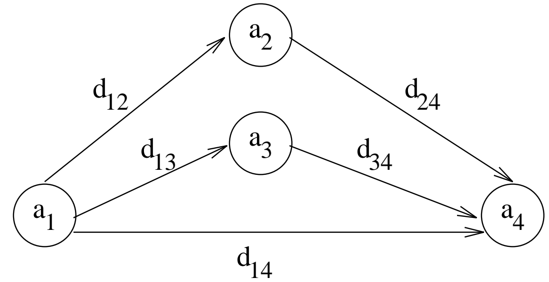


Figure 3: Software Graph Example

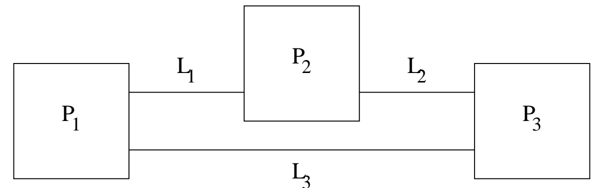


Figure 4: Hardware Graph Example

We assume that actions a_1 and a_3 have been assigned to P_1, a_2 to P_2 and a_4 to P_4. We obtain the partition of the software graph as shown on fig.5. Thus d_{12}, d_{24}, d_{14} and d_{34} are edges between partition elements. Similarly, we assume that the data dependence d_{12} is assigned to R_1, d_{24} to R_2, d_{14} to R_3 and d_{34} to R_4. Note that this latter dependence is assigned to a route introducing an intermediate processor which will have to relay the communication.

In order to complete the distribution c/P's and a/L's are added as shown fig.6. Then the actions are scheduled on the processor they have been assigned

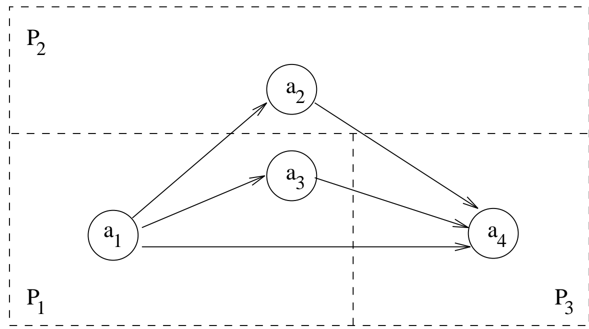


Figure 5: Distribution Example

to, and identically the inter-processor communication actions are scheduled on the link they have been assigned to. The computation unit of P_1 will execute the sequence of computation (a_1/P_1;a_3/P_1). In parallel with this sequence, the communication unit of P_1 attached to L_1 will cooperate with the one of P_2 also attached to L_1 in order to execute the sequence of communications (a_1/L_1;a_2/L_1). This sequence is imposed by the dependence edge $\bar{d}_1/L_1$. Similarly the communication unit of P_2 attached to L_3 will cooperate with the one of P_4 also attached to L_3 in order to execute the communication a_1/L_3 in parallel with the other units of P_2. And so on for the rest of fig.6.

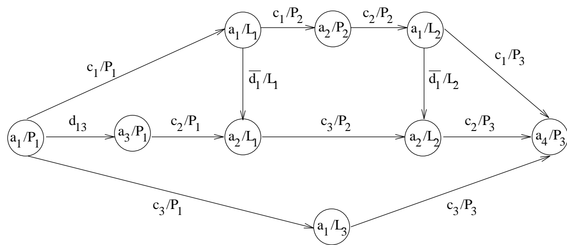


Figure 6: Scheduling Example

5 Performance Optimization

Several *dist/sched* transformations may be obtained from a given software/hardware graphs pair. As stated in the introduction, we look for one which minimizes the response time of the transformed graph. The response time is the critical path of the graph, computed from the execution durations of the actions (including communication actions).

We denote $\Delta : \mathcal{A} \times \mathcal{P} \rightarrow \mathbb{R}$ the function giving the execution duration of every action. To simplify the notations, we assume in the following that the multiprocessor is homogeneous, that is that it is built with only one type of processor and only one type of communication link. Then the function Δ is simply $\Delta : \mathcal{A} \rightarrow \mathbb{R}$.

Note that usual optimization schemes do not take into account communication delays, whereas in our model each routed communication is detailed enough so that the duration of each communication action may be determined in function of the amount of data it transfers [5].

We then denote $\bar{\mathcal{D}}$ the partial order between actions after the *dist/sched* transformation:

$$\bar{\mathcal{D}} = \mathcal{D} \cup \left(\bigcup_{p \in \mathcal{P}} \bar{\mathcal{D}}_p \right) \cup \left(\bigcup_{l \in \mathcal{L}} \bar{\mathcal{D}}_l \right)$$

We also denote $s(a)$ and $e(a) = s(a) + \Delta(a)$ the start and end execution dates of an action $a \in \mathcal{A}$. The partial order $\bar{\mathcal{D}}$ is easily translated into relations between execution dates:

$$\forall (a, a') \in \bar{\mathcal{D}} \quad e(a) \leq s(a')$$

Then by denoting $\Gamma(a) = \{a' \in \mathcal{A} \mid (a, a') \in \bar{\mathcal{D}}\}$ the successors of an action $a \in \mathcal{A}$ and $\bar{\Gamma}(a) = \{a' \in \mathcal{A} \mid (a', a) \in \bar{\mathcal{D}}\}$ its predecessors, we may define its *earliest start from start* $S(a)$ and its *earliest end from start* $E(a)$ and finally the response time R:

$$\begin{aligned} S(a) &= \begin{cases} 0 & \text{if } \bar{\Gamma}(a) = \emptyset \\ \max_{a' \in \bar{\Gamma}(a)} E(a') & \text{otherwise} \end{cases} \\ E(a) &= S(a) + \Delta(a) \\ R &= \max_{a \in \mathcal{A}} E(a) \end{aligned}$$

The optimization problem, minimizing R, as other resource allocation optimization problems, is known to be NP-complete. Among the heuristics used to cope with this problem, the ones based on *schedule flexibilities* are known to give almost as good results as simulated annealing but are much faster [3]. For every action, its schedule flexibility $F(a)$ is defined using R and its *latest end from end* $\bar{E}(a)$ and its *latest start*

from end $\bar{S}(a)$:

$$\bar{E}(a) = \begin{cases} 0 & \text{if } \Gamma(a) = \emptyset \\ \max_{a' \in \Gamma(a)} \bar{S}(a') & \text{otherwise} \end{cases}$$

$$\bar{S}(a) = \bar{E}(a) + \Delta(a)$$

$$F(a) = R - S(a) - \bar{S}(a)$$

Schedule-flexibility based heuristics are greedy algorithms where an action is assigned to a processor at each step. This action is chosen among the ones, called *candidates*, which have all their predecessors already assigned. The schedule flexibility is used as a cost function to select the best action/processor pair.

We have developped a new schedule-flexibility based heuristics using our models to take into account communication delays (see the discussion on the Δ function at the beginning of this section). Moreover we have extended the cost function to take into account the critical path increase of candidate assignments. Indeed, assigning a candidate action to a processor may lead to delay the candidate's execution. When comparing the assignments of a candidate on different processors one may obtain different earliest start dates. The later an action is executed, the least schedule flexibility it has, until it becomes critical. Then the schedule flexibility remains null (it is a piecewise continuous function) but the critical path begins to increase. We therefore introduce the *schedule penalty* $P(a)$ which is also a piecewise continuous function and is the counterpart of the schedule flexibility which is a slack before any response time increase. The difference between the schedule penalty and the schedule flexibility is then a continuous function that we call *schedule pressure* $\sigma(a)$:

$$\forall a \in \mathcal{A} \quad \sigma(a) = P(a) - F(a)$$

With our models, the heuristics consists in building step by step the *dist/sched* transformation by composing elementary graphs transformations. Each step transformation consists in assigning and scheduling a selected candidate action on a selected processor. Therefore, the number of steps is equal to the number of actions. The pair (candidate, processor) selected at step i is the one which verifies:

$$\exists (a,p) \in V^i \times \mathcal{P} \quad | \quad \sigma^{iap} = \max_{a' \in V^i} \left(\min_{p' \in \mathcal{P}} \sigma^{ia'p'} \right)$$

where $V^i \subset \mathcal{A}$ denotes the set of candidates which are the actions not yet selected and having no predecessor not yet selected. This heuristics is detailed in [8].

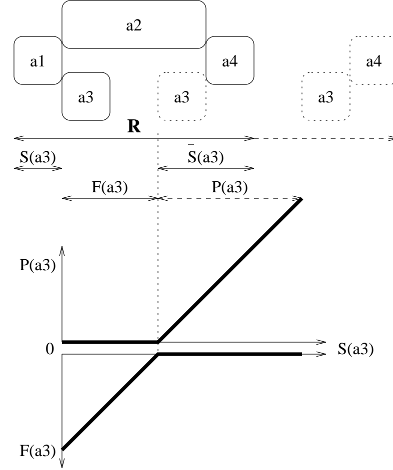


Figure 7: Schedule pressure components

The number of processors may also be optimized. If the response time constraint is satisfied, the user may try to decrease the number of processors, otherwise he may try to increase it. A reasonable initial number of processors is the integer immediately above the ratio between the maximum response time (sum of the durations of the actions excepted the communication actions, corresponding to a mono-processor execution) and the minimum response time (critical path before implementation, that is with no inter-processor communication action).

6 Executive Generation

The executive supports the implementation described in section 4, that is mainly the distribution and the scheduling of the algorithm on the multiprocessor. Distribution is a spatial allocation of hardware resources, whereas scheduling is a temporal allocation of these resources.

An executive is said to be *dynamic* when the decisions and optimizations it has to carry out in order to satify the real time constraints are done at execution time using the real-time clock. Conversely it is said to be *static* when the decisions and optimizations are done at compile time with the knowledge of the execution duration of actions. So, dedicated or general purpose real-time distributed executives currently used, may be simply classified according to the three main resource types (computation units, local memories and communication units) and the two allocation

types as depicted in fig.8 where the letter S stands for static and the letter D for dynamic.

Executive examples	comput dist	comput ord	mem dist	mem ord	comm dist	comm ord
Meiko	D	D	D	D	D	D
CHORUS	S	D	S	D	D	D
SynDEx v1	S	D	S	D	S	D
SynDEx v2	S	S	S	D	S	D
SynDEx v4	S	S	S	S	S	S

Figure 8: Executives classification

Note that a static executive introduces an overhead which is less expensive than in the case of a dynamic executive. It is important to use as much as possible static executive and to use dynamic executive only when it is necessary. The parts of the application (environment and/or algorithm and/or architecture) for which enough informations are available will be supported by a static executive whereas the other parts will be supported by a dynamic executive. A current research axis consists in improving the A^3 optimization heuristics presented in [8] so that it may take these implementation choices. These choices are exhibited in our implementation model through the scheduling orders $\bar{\mathcal{D}}_p$ (total order implying a sequential execution of actions) and $\mathcal{D}_p$ (partial order implying a pseudo-parallel execution of actions).

In any case, with or without pseudo-parallelism in each execution unit, there is also a true parallelism between units. The executive must support both kinds of parallelism in respect with the causality imposed by the partial order involved by the distribution and the scheduling of the implementation (partial order which is itself, let's recall it, only an enforcement of the original partial order involved by the software graph). Obviously, when this causality is respected, dead-locks are avoided.

The causality is initiated by the inputs received from the environment. Another current research axis concerns executives able to handle several parallel inputs involving a partial order between these inputs. Here we assume that the software graph has only one input action.

Intra-unit dependences involving the total orders $\bar{\mathcal{D}}_p \supset \mathcal{D}_p$ and $\bar{\mathcal{D}}_l$ (d_{ij} and $\bar{d}_i/L_j$ in our example of fig.6), as well as the conditionning actions, are naturally implemented as sequential and conditionnal execution of actions on macro-sequencers. But the inter-unit dependencies $\mathcal{C}_p$ involve inter-unit synchronizations through the local memory and the macro-sequencer and require special care for their implementation. Inter-processor communication actions $\mathcal{A}_l$ also require special care for their implementation when the cooperation they require between communication units is not completely supported at the hardware level.

These carefully implemented subroutines, which coding is strongly dependent on the architecture, compose the executive generic kernel. The executive is built by instanciating (calling sequentially or conditionnaly) these subroutines for each $\mathcal{C}_p$ or $\mathcal{A}_l$ and by instanciating the user's subroutines which code the algorithm computation and I/O actions.

We present now an example of static executive generation. Fig.9 shows an architecture with two processors, both with one computation unit, one communication unit and one macro-sequencer, and the left one also with an I/O unit. Below this hardware graph is represented the distribution and scheduling of the example algorithm given fig.1. The vertical arrow below each execution unit represents the sequence of actions it executes. The big horizontal arrows within boxes represent the inter-processor communications and the small horizontal arrows represent inter-unit synchronizations.

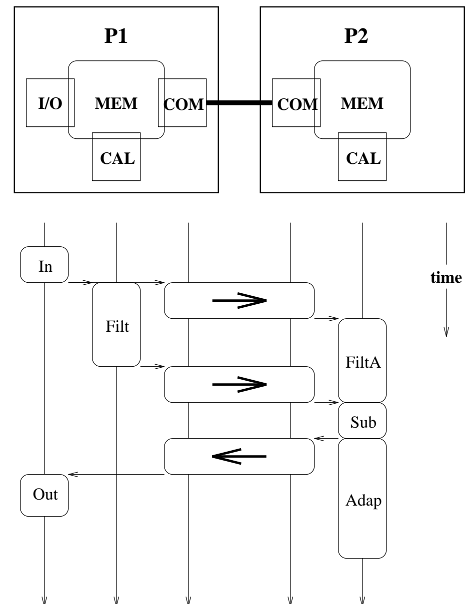


Figure 9: Static executive example

7 The SynDEx Software

SynDEx [6] supports the A^3 methodology discribed previously. Its graphical interactive user interface allows to input both harware and software graphs. The latter can be imported from a file which is the result of the compilation of a source program written in the synchronous language SIGNAL [1].

Then the user executes the A^3 heuristics which finds out a distribution and scheduling minimizing the response time. As a result, the predicted real-time behaviour of the algorithm on the multiprocessor is visualized. This feature is very important because the user can evaluate real-time applications even before the hardware is available. It is sufficient to know enough informations about it to experiment its real-time capabilities for a given algorithm. Moreover, using the performance prediction diagram, the user can iterate at this level, trying different hardware solutions to size the architecture. Namely, he can find the minimum number of processors and of physical links he needs.

Finally, a real-time distributed executive is generated, built from the application library including computation and I/O actions given by the user and the executive generic kernel written in C, including conditionning, memory and communication actions. This generated executive is garanteed to be dead-lock free. Presently executive generic kernels have been developped for multiprocessor architectures based on the Tranputer T800 and on the Texas TMS320C40.

The executive may be generated with chronometric instructions [5]. Hence, it is possible to know the beginning and ending dates of every action during a real-time execution. These informations may be compared with those calculated by the A^3 heuristics.

Since the executives are automatically generated with SynDEx, low level hand coding and multiprocessor real-time code debugging are eliminated, consequently the development cycle duration of real-time applications is tremendously reduced. Moreover, as the optimization heuristics is fast, several *hardware sizes* may be rapidly compared in order to find the best trade-off between real-time and hardware size constraints.

Fig.10 shows a screen snapshot of the SynDEx graphical user interface displaying a didactic example application with two tranputers executing an adaptive filter algorithm.

The left window displays a topological viewpoint of the application, with the hardware graph on the top (processors root and p with communication units root/l2 and p/l1 connected) and the software graph on

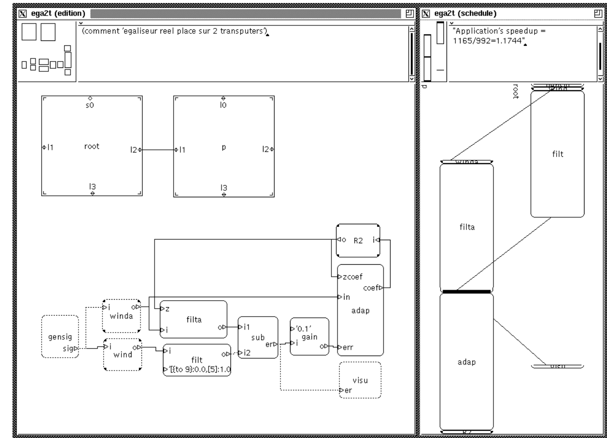


Figure 10: SynDEx software: a didactic example

the bottom. Each action with a dotted frame is constrained to be executed on a given processor. Each graph component may be moved or sized within a mouse-drag or may popup its own menu of context dependent edit commands. Commands may also be typed and interpreted in the top text view which is also used to display informations which are not represented graphically, such as execution durations or dataflow types.

The right window displays, after a run of the A^3 heuristics, a temporal viewpoint which represents the execution schedule of one reaction to a stimulus. The time is flowing from top to bottom with one column for each processor. The position and height of each action frame represent respectively its execution date and its duration. The top and bottom of each slant line represent respectively the starting and ending dates and processors of an inter-processor communication. More details may be obtained numerically in the top text view, such as the schedule flexibility of any action, the idle periods of any hardware resource, or more global informations such as the reaction's duration or the application's speedup.

To generate the executive, once satisfied with the predicted performances, the user has just to select the corresponding option from the popup menu of the top-left view (which is also used to zoom the bottom graphs view). Among the generated source files, only one, the makefile, has to be modified to locate the user's functions coding the algorithm actions.

On a Sun-Sparc10 workstation, the adequation takes less than one second and the executive is generated in about two seconds for this example. For a

more complex example with an hypercube of 16 processors executing 269 actions as shown fig.11, the adequation takes less than 5 minutes and the executive is generated in about 10 seconds.

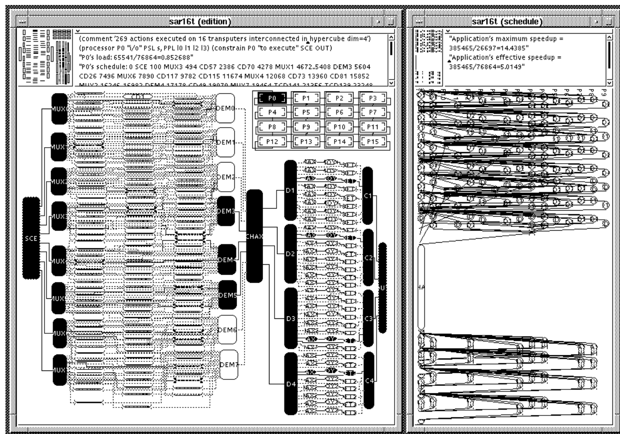


Figure 11: SynDEx software

8 Conclusion

We have developped a methodology called A^3 (Algorithm Architecture Adequation) which unifies through graph models the algorithms and the massively parallel architectures and therefore formalizes in terms of graphs transformations the implementation of an algorithm on an architecture. In the frame of this methodology, we have developped an heuristics to optimize the real-time performances of the implementation because efficient use of architecture resources is needed for economical reasons or to satisfy real-time constraints. The practical interest of this methodology is that it finally allows the automatic generation of a dead-lock free executive from an optimized implementation. This drastically reduces the development cycle duration of real-time applications on MPCS.

To support this methodology, we have developped the SynDEx graphical interactive software which has been tested on multi-Transputer and multi-TMS320C40 architectures.

We are presently refining our hardware model in order to obtain a more accurate determination of the execution durations of computations and inter-processor communications. We are also generalizing our executive models to support the combination of static and dynamic distribution and scheduling and to support algorithms with multiple inputs. The heuristics will be consequently modified. Future research axis are planned in the field of memory optimization, input rate minimization and its relationship with response time optimization.

References

[1] A. Benveniste, G. Berry:
The Synchronous Approach to Reactive and Real-Time Systems.
Proc. of the IEEE vol79, n.9, pp.1270–1282, 1991.

[2] A. Benveniste, P. Le Guernic, Y. Sorel, M. Sorine
A Denotational Theory of Synchronous Reactive Systems. Information and Computation, vol. 99, n° 2, pp. 192–230 (1992).

[3] C. Coroyer, Z. Liu:
Effectiveness of heuristics and simulated annealing for the scheduling of concurrent tasks: an empirical comparison.
INRIA Research Report n°1379, 1991.

[4] J. B. Dennis:
First Version Data Flow Procedure Language.
Technical Memo MAC TM61, MIT laboratory for Computer Science, 1975.

[5] F. Ennesser, C. Lavarenne, Y. Sorel:
Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs SynDEx.
INRIA Research Report n°1769, 1992.

[6] C. Lavarenne, O. Seghrouchni, Y. Sorel, M. Sorine:
The SynDEx software environment for real-time distributed systems design and implementation.
Proc. of the European Control Conference, 1991.

[7] C. Lavarenne, Y. Sorel:
Specification, Performance Optimization and Executive Generation for Real-Time Embedded Multiprocessor Applications with SynDEx.
Proc. of Real-Time Embedded Processing for Space Applications, CNES International Symposium, 1992.

[8] C. Lavarenne, Y. Sorel:
Performance Optimization of Multiprocessor Real-Time Applications by Graphs Transformations.
Proc. of PARCO93 conference, 1993.

[9] C.A. Mead, L.A. Conway:
Introduction to VLSI systems.
Ed. Addison-Wesley, 1980.