Parallel Computer Algebra[1]

Jean-Louis Roch and Gilles Villard

LMC-IMAG
Institut Fourier, BP 53X
100, rue des Mathématiques
38041 Grenoble Cedex 9, France
Email: `[Jean-Louis.Roch, Gilles.Villard]@imag.fr`

July 15, 1997

[1]Updated versions may be found at `http://www-lmc.imag.fr/ gvillard/CFPAR/`

Building and implementing parallel algorithms in the area of computer algebra has become an important thread of research for more than a decade with the increasing availability of various parallel architectures, from dedicated machines to network of workstations. New algorithms have been built and implemented to solve high performance computing challenges.

The aim of this tutorial is to give an introduction to parallel algorithms in computer algebra, from the building of an efficient algorithm to its effective implementation on a given architecture. Parallel computer algebra systems, that exploit the parallelism of an algorithm on a given architecture, play a central role to ensure efficient executions. Due to the variety of parallel programming models, several such systems propose various approaches to express parallelism, from data distribution to functional parallelism.

After an introduction to algorithmic techniques and classical programming models, the tutorial will focus on parallel computer algebra systems, parallel linear algebra algorithms and their effective implementations. The tutorial is organized in four parts :

1. **Parallel efficient algorithms.** The major techniques used to build efficient algorithms on theoretical machine models are presented. They are illustrated by various basic computer algebra algorithms. Due to the non-uniformity of memory access, communication complexity is a key point to take into account in the analysis of the algorithm.

2. **Programming models and scheduling.** To combine expressive power and portability, several programming models have been proposed, from message-passing to bulk-synchronous programming and functional languages. The inherent overhead due to their emulation makes each of them suited to a specific range of applications.

3. **Parallel computer algebra systems.** Different parallel systems are proposed that are based on the coupling of a sequential system and a parallel programming model. They are often guided by the classes of applications on which they have been experimented.

4. **Parallel linear algebra.** The parallelization techniques introduced before are illustrated on various research problems in parallel linear algebra : system solving, gcd, rank and normal forms.

Chapter 1

Parallel efficient algorithms

Parallel algorithmic is a successful theory. Several methods, techniques and paradigms, which are presented in several books and surveys [60, 5, 30, 38, 35, 20, 41, 28, 39, 45] have been developed to build powerful theoretical algorithms. Furthermore, they stand as a basis for implementation of performant programs on effective parallel architectures. Those general techniques overflow computer algebra framework even if arithmetic and algebraic computations are of specific interest.

In this chapter, we introduce the main techniques involved in the building of parallel algorithms. They are illustrated on elementary computer algebra problems. The underlying model is PRAM but the data-flow graph representation is also introduced. It is used to describe executions of a parallel algorithm and to define its cost. Three factors are here preponderant: parallel execution time, number of operations and granularity which is related to the required volume of communications. An efficient algorithm realizes a compromise solution between those three factors.

The organization of the chapter is as follows. Section 1 describes the local PRAM model, the data-flow graph representation and cost analysis. Following sections illustrate, using simple examples, the main techniques involved in the building of:

- section 2: a coarse granularity algorithm from a fine grain optimal one;
- section 3: a fast optimal algorithm from a very fast but non optimal one;
- section 4: an very fast optimal randomized algorithm from a deterministic but non optimal one.

Finally, in the last section, we give an overview of parallel time complexity, focusing on boolean-arithmetic circuits which are commonly used in computer algebra.

1.1 PRAM, DFG and cost analysis

The Parallel Random Access Machine (PRAM) [18, 4] is the most common execution model used to build and analyze parallel algorithms. Its major feature is to be independent from the number of processors used. In this section we focus on the local PRAM model introduced in [38]. Cost analysis takes into account both arithmetic and communication complexities.

In the following, A denotes an algorithm and A_n its restriction for input of size $O(n)$.

1.1.1 The PRAM model

A Local Parallel Random Access Machine (PRAM) is setted of:

- an (infinite) number of processors $P_0, \ldots, P_N, \ldots$, each indexed by an integer (*processor identifier* or `pid` in short). Each processor is a RAM (Random Access Machine [2]) and gets its own local memory which contains its own pid.
- a global (or shared) memory. Each processor can copy data from the global memory into its own local memory: this operation is called `global read` or `read` in short. Conversely, each processor can copy a data from its own local memory into the global one: this operation is a `write` operation.
  Initially, input variables are available in global memory. At the end of computation, final outputs are also stored there.

- A program that consists in a finite sequence of RAM elementary instructions, extended by the global elementary (i.e. single word location) read and write instructions.

- a global clock that ensures a synchronous mode of computation. After initialization (first top), processors are ready to execute the first instruction of the program. At each top (or *step*), each processor executes the next RAM instruction in the program. Thus it performs either an elementary arithmetic operation within its local memory or an access to the shared memory (read or write).
  The program terminates when processor with pid 0 executes the `halt` instruction.

Note that the program may contain branching instructions eventually depending on the pid value. Due to branching instructions, at a given top, processors may execute different instructions (Multiple Instruction Multiple Data – MIMD – type).

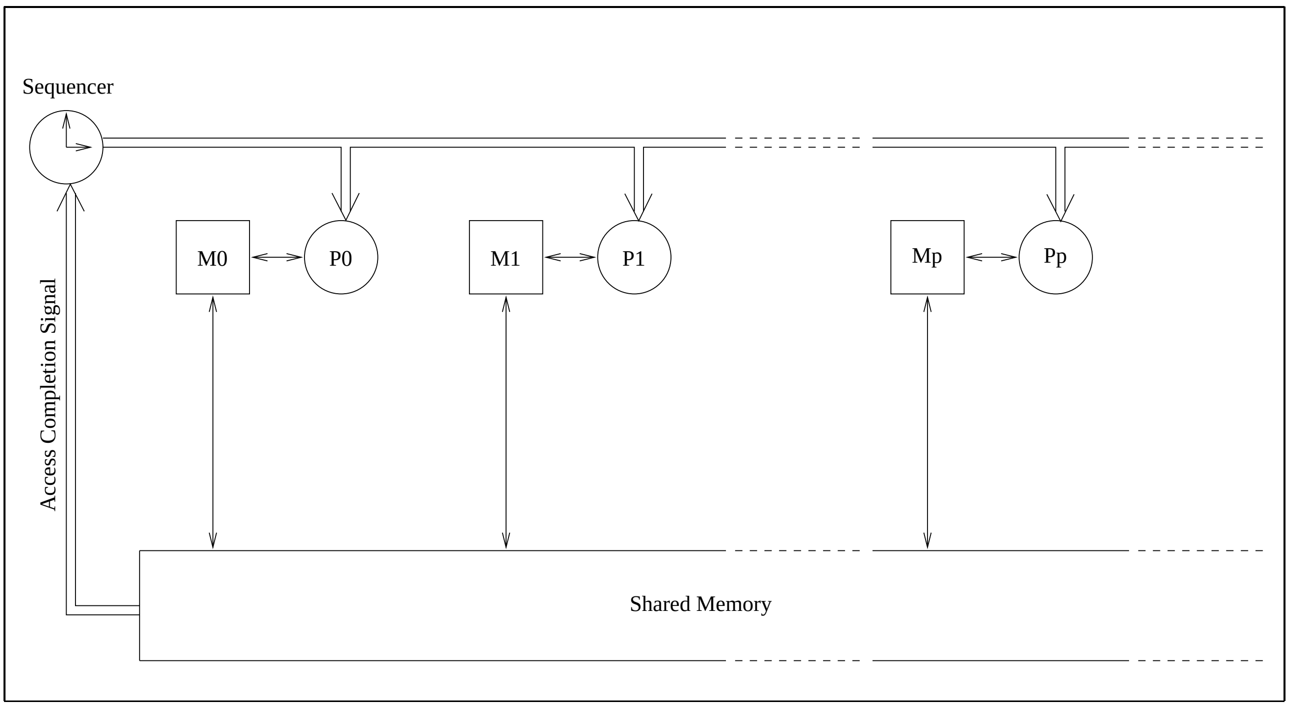

Figure 1.1: The local PRAM execution model

Semantics of access in shared memory. Due to the synchronous mode of computation, semantics of global memory access is simple and only depends on the behavior when, at a same top, several processors concurrently accede to a same single location in the shared memory.

At a same top, two processors can't perform both a read and a write in the same location. But concurrent read (or concurrent write) access may be allowed, depending on the PRAM:

- an EREW-PRAM (Exclusive Read Exclusive Write) does not allow concurrent access to a single location.

- a CREW-PRAM (Concurrent Read Exclusive Write) allows only concurrent read access.
- a CRCW-PRAM (Concurrent Read Concurrent Write) allows concurrent access (all in the same mode, either read or write).

When a concurrent write operation is performed into a single location in the shared memory, different semantics are considered depending on the reduction operation performed to produce the final value:

- COMMON: all processors have to write the same value. If not, an error is produced.
- ARBITRARY: an arbitrary processor writes its value.
- PRIORITY: the processor with the minimum pid writes its value.
- CUMULATIVE: the sum of all the concurrent values is written. The addition operation (defined between single location values) is assumed to be associative. Furthermore, it is assumed to be commutative to have, like concurrent read and common or arbitrary write operations, a semantic independent from the pids of the writing processors. This concurrent write mode is also called *combining* [41].

As detailed further, those different variants of the PRAM are relatively closed to each others: each one can simulate the other one with small overheads [14, 41, 28].

Dynamic task creation The above definition presents two drawbacks:

- it assumed that, after initialization, an unbounded number of processors start execution;
- dynamic creation of parallelism has to be described in the program using busy-waiting; this means that the scheduling of the program is completely described in the program.

In the initial definition from [18], only the processor with pid 0 starts execution of the program. To generate parallelism, an elementary `fork <e>` instruction is defined. When a processor P executes this instruction, an inactive processor P' is reset. The accumulator of P (which may contain an address in the shared memory where some parameters are stored) is first copied into the one of P'. The pid of P' is then put into the accumulator of P. This allows P and P' to later communicate via the shared memory.
At the next step, P executes the following instruction (the one that follows the `fork`) and P' starts the execution of the program at the instruction labeled `e`.
Using `fork`, dynamic task creation is made possible, scheduling (allocation of inactive processors) being ensured by the PRAM machine. However, this modification implies that any PRAM program that uses a polynomial number $n^{O(1)}$ of processors takes a time $\Omega(\log n)$, forbidding the building of constant time algorithms; if an algorithm is involved during the execution of a program (e.g. inside the body of a loop), this overhead may easily be avoided. Analysis of costs in this chapter are made under the previous model, thus without taking into account task allocation overhead.

Randomized PRAM To support execution of randomized algorithms, the PRAM is extended in the following way. A new `random` instruction is introduced that allows each processor to generate (in one top) a random bit (or a random number that fits in a single memory location).

Random generations (i.e. `random` instructions) performed by a processor during the execution are assumed to be independent realizations of an uniform law. Moreover, generations performed in parallel at a given top by different processors are also assumed to be independent.

1.1.2 Execution of a PRAM program and data-flow graphs

Being given the input data, the execution of a PRAM program may be represented as a direct acyclic graph. Vertices correspond to instructions that are executed (one vertex, one instruction) and edges to precedence relations between instructions. Basically, if v (resp. w) is the vertex representing an instruction executed[1] at step i (resp. $i+1$), then there is an edge from v to w.

However, the finest representation of a parallel algorithm is given by the data-flow graph (DFG) of any of its executions. DFG is direct acyclic and bipartite with node sets $J = \{j_1, \ldots, j_n\}$ corresponding to instructions (j meaning *job*) and $T = \{t_1, \ldots, t_m\}$ corresponding to single assignment data (t meaning transition). An edge goes from t_k (resp. j_i) to j_i (resp. t_k) if j_i is a read (resp. write) instruction of the global data related to t_k.

In the DFG, any memory access, either global or local, is represented by an edge between a location (represented by a transition node) and an instruction (a job node) that requires the access. Except for transitions related to input, immediate ancestors of each transition t_k are write instructions: only one on an exclusive-write PRAM, eventually more on a concurrent-write one. Conversely, its immediate successors (except for transitions related to output) are read instructions: only one on an exclusive PRAM, eventually more on a concurrent-read one. This means that when all immediate successors (job nodes) of a transition have been executed, the location related to it in global memory may be garbaged.

Let us considered the DFG related to a tree computation scheme. As an illustration, we consider two algorithms that solve the *iterated product*[2] problem: it consists in computing the product of n elements. In order to exhibit parallelism, multiplication is assumed to be associative and commutative. A balanced binary tree scheme gives an algorithm that works on an EREW PRAM; related DFG is shown in figure 1.2. On a CUMULATIVE-ERCW PRAM all products may be performed concurrently and cumulated on a shared location (fig. 1.2.b).

This graph defines a precedence relation, denoted $\prec$, between instruction nodes in J. Let j_1, j_2 be two nodes in J; $j_1 \prec j_2$ if there is a path in DFG from j_1 to j_2. In the following, we will consider the subgraph $DFG_a(J, \prec)$ of DFG, where only arithmetic instructions and their precedence relations are represented.

Remark 1. The data-flow description of the algorithm is roughly equivalent to a *straight-line program* [32].

Remark 2. Note that symmetry of input (resp. output) edges to a transition node assumes commutativity of access. This is verified for any concurrent write (resp. read) access defined on the

[1] Instructions corresponding to v and w may be executed by different processors.

[2] also called *iterated sum* when an addition law is considered

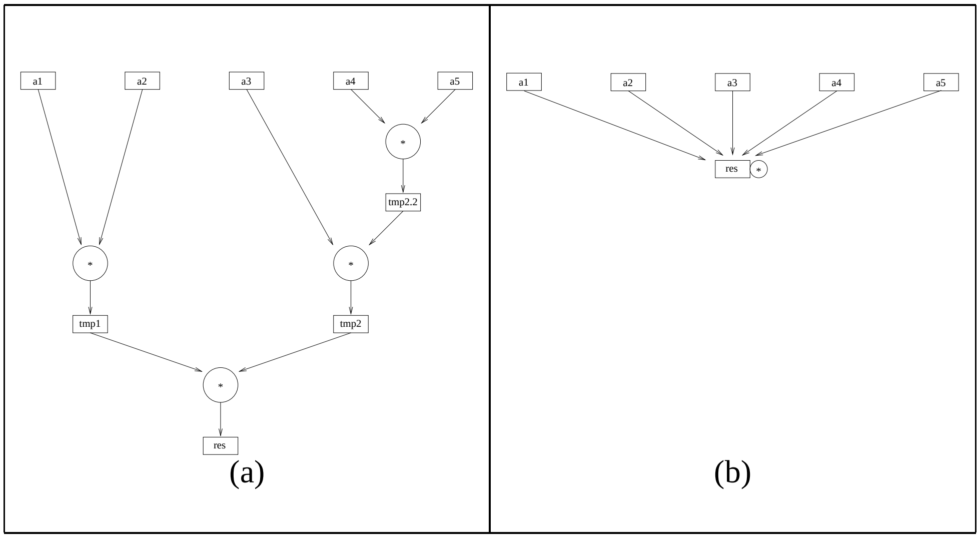

Figure 1.2: DFG of two iterated products: (a) EREW (b) cumulative-ERCW

PRAM.

1.1.3 Describing PRAM algorithms: ATH language

PRAM stands as an abstract model virtualizing any parallel architecture. In order to describe PRAM algorithms, we need an elementary programming language which leads to easy description of algorithms.

Sine evaluation of a parallel algorithm is directly related to the analysis of DFG, a sequential description should be sufficient since data-dependencies appear implicitly: each read access to a location gets the value put by the last write in a sequential execution. However, two characteristics, which not appear in a sequential description, are to be taken in account:

- two levels of memory access are distinguished: local and global. Global memory access support CUMULATIVE-CRCW semantics.

- the elementary unit of instruction is the block. A block is a sequence of elementary RAM instructions. A block is executed in sequential; it takes benefit of local access.

In the following, we consider an extension of the basic PRAM basic language introduced in [18] based on those two considerations. This abstract language is called *ATH*, an acronym for *Asynchronous Tasks Handling*.

Blocks of instructions are defined as procedures body. Execution of such a block is a *task*. Tasks may be ordered either in sequence using synchronous procedure call or in parallel using asynchronous procedure calls (prefixed by `fork`). In this last case, precedence relation between tasks is defined in a natural way, according to shared-data dependencies that appear in a sequential execution of the program. Data dependencies concerning local data are then not considered in the relative DFG.

Figure 1.3 gives two different recursive programs for the iterated product. Version (a) works on an EREW PRAM and is related to DFG presented in figure 1.2.a. Version (b) works on a CUMULATIVE-ERCW; corresponding DFG is presented in figure 1.2.b.

```
Product(a : in E,
        b : in E,
        c : out E)
begin
  c.Write( a.Read()*b.Read() );
end

IterProd( n       : in  integer,
          a[1..n] : in  array of shared E,
          res     : out shared E)
begin
  if(n==1)
    res.Write( a[1].Read() );
  else
    tmp1i, tmp2 : shared E;

    fork IterProd(n/2, a[1..n/2], tmp1);
    fork IterProd(n-n/2, a[n/2+1..n], tmp2);
    fork Product(tmp1, tmp2, res);
  end if
end
```

(a)

```
IterProd( n       : in  integer,
          a[1..n] : in  array of shared E,
          res     : out shared E )
begin
  if(n==1)
    res.Cumul<*>( a[1].Read() );
  else
    fork IterProd( n/2, a[1..n/2], res );
    fork IterProd( n-n/2, a[n/2+1..n], res );
  end if
end
```

(b)

Figure 1.3: ATH code of two iterated products: (a) EREW, (b) cumulative-ERCW. *Data in shared memory are explicitly declared by the prefix* `shared`. Notation `x.f()` *means that function* `f` is called on the data in shared memory `x`. *In program (b), the function call* `x.Cumul<*>( v )`specifies a cumulative concurrent write on the data in shared memory `x`*; the commutative and associative binary function implementing the operation is* `*`.

1.1.4 Time, work and communication costs

Consider a PRAM program. In the following, n denotes the size of the input. The arithmetic cost is characterized by:

- the *parallel time* $T(n)$ which corresponds to the number of executed steps;
- the *arithmetic work* $W_a(n)$, i.e. the whole number of operations performed.

Those quantities are independent of the number of processors and thus may be defined directly from the DFG description of the execution.

Definition 1 *The parallel time $T(n)$ is the maximal depth of DFG(x) for any input x of size n:*

$$T(n) = \max_{x, ||x||=n} Depth(DFG_a(x)) \tag{1.1}$$

The arithmetic work $W_a(n)$ is the number of instruction nodes of DFG(x) for any input x of size n:

$$W_a(n) = \max_{x, ||x||=n} \#V(DFG_a(x)) \tag{1.2}$$

The arithmetic cost is denoted:

$$O_a(T(n), W_a(n)) \tag{1.3}$$

Similarly, the communication cost is characterized by two factors:

- the *communication delay*[3] $C_d(n)$ is the maximal number of global memory access performed by a processor;
- the *communication work* $W_c(n)$, i.e. the whole number of global memory access performed.

The PRAM program implements a scheduling of the DFG on an infinite number of processors: any access to the local memory on each processor is not considered as a communication. Thus, the communication cost may vary depending on the number of processors used in the program.
To define communication cost with respect to a parallel algorithm (independent of a number of processors, and so more general than the program that implements it), we will refer to its DFG.

Definition 2 *The communication work $W_c(n)$ is the maximal number of edges for any input of size n:*

$$W_c(n) = \max_{x, ||x||=n} \#E(DFG(x)) \tag{1.4}$$

The communication delay $C_d(n)$ is the maximal length of a path in DFG from an input data to an output one:

$$C_d(n) = \max_{x, ||x||=n} Depth(DFG(x)) \tag{1.5}$$

The communication cost is denoted:

$$O_c(C_d(n), W_c(n)) \tag{1.6}$$

In order to compare arithmetic and communication costs, the granularity $g(n)$ is defined.

Definition 3 *The* granularity $g(n)$ *is the ratio between the arithmetic and communication works:*

$$g(n) = \frac{W_a(n)}{W_c(n)} \tag{1.7}$$

1.1.5 Efficient algorithms

Let $\mathcal{A}$ be an algorithm with cost $T(n), W_a(n), C_d(n), W_c(n)$. Let $W_s(n)$ the work of the best known (sequential) algorithm that solves the same problem.

The building of a parallel algorithm to solve a given problem may be aimed at different directions:

- either finding the smallest amount of time required to solve a problem. In this context, the class NC of problems that may be solved in parallel time $T(n) = \log^{O(1)} n$ using a polynomial number of processors $W_a(n) = n^{O(1)}$ plays a central role.

[3] $C_d(n)$ is called *communication complexity* in [28].

- or building an *efficient* program that leads to solve larger problems in a reasonable amount of time taking benefit of the ability to use several processors, let us say p. Here, arithmetic and communication overheads (i.e. $W_a(n)$ and $W_c(n)$) are to be carefully taken into account in order to guarantee efficient executions.

A common trade-off [38] consists in building parallel algorithms that:

- have *polynomial speed-up*, i.e.

$$T(n) = O(W_s(n)^\epsilon) \quad \text{with } \epsilon < 1 \tag{1.8}$$

- are *work-preserving*, i.e.

$$W_a(n) = \Theta(W_s(n)) \tag{1.9}$$

  The *inefficiency* ν measures the arithmetic overhead:

$$\nu(n) = \frac{W_a(n)}{W_s(n)} \tag{1.10}$$

- require *few communications*, i.e

$$W_c(n) = O(W_a(n)^\epsilon) \quad \text{with } \epsilon < 1 \tag{1.11}$$

Definition 4 *$\mathcal{A}$ is said:*

- fast *if it achieves poly-logarithmic parallel time with a polynomial number of operations, i.e. $T(n) = \log^{O(1)} n$ and $W_a(n) = n^{O(1)}$.*
- optimal *if it is fast and has constant inefficiency.*
- efficient *if it has a polynomial speed-up and a constant efficiency.*
- *of* coarse-granularity *if it has polynomial granularity, i.e. $g(n) = n^{O(1)}$.*

In order to not absolutely reject fast algorithms involving a small overhead in arithmetic operations, fast algorithms with poly-logarithmic inefficiency will be considered as efficient *also.*

In the following, some main techniques that lead to the building of an efficient *and* of coarse-granularity algorithm are overviewed. It turns out that minimizing time without preserving work (i.e. building NC algorithm) is of specific interest:

- algorithmic techniques involved for both are very close;
- it gives a lower bound on the best parallel time that may be achieved;
- an inefficient but fast algorithm may successfully be coupled to a slower but efficient one to build a faster program.

1.1.6 Example

We illustrate previous definitions on the iterated sum algorithm presented in figure 1.3.a. Scalar product of two vectors is directly reduced from iterated sum; it may be applied to perform matrix multiplication in a semi-ring.

Iterated sum

For the EREW algorithm presented in figures 1.3.a and 1.2.a (balanced tree computation scheme), we have assuming $n = 2^m$:

$$\begin{array}{ll} T(n) = \log n & C_d(n) = \log n + 1 \\ W_a(n) = n - 1 & W_c(n) = 2n - 1 \end{array} \tag{1.12}$$

This algorithm is optimal since its cost is – asymptotically – a lower bound.

As a consequence, the scalar product of two vectors is computed on an EREW with cost:

$$O_a(\log n, n) \quad \text{and} \quad O_c(\log n, n). \tag{1.13}$$

On a semi-ring, $+$ is commutative. Thus, on a cumulative-CRCW PRAM, this problem may be computed with parallel cost (fig. 1.2.a):

$$O_a(1, n) \quad \text{and} \quad O_c(1, n). \tag{1.14}$$

However, description of the computation scheme (cf program in fig. 1.3.b) may require $O_a(\log n, n)$.

Matrix product

Consider the problem of computing a square matrix product $C = AB$ in a semi-ring (i.e. using only $+$ and $\times$ operations).

Let n be the dimension of the matrices: since $C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$, the problem reduces to n^2 independent scalar products. Using 1.13, we obtain a parallel algorithm with cost:

$$O_a(\log n, n^3) \quad \text{and} \quad O_c(\log n, n^3). \tag{1.15}$$

Since $W_s(n) = \Theta(n^3)$ [37], this algorithm is efficient.

However, $g(n) = O(1)$ and it is not coarse-granularity. Besides, it can be seen that, if E is a field (or ring), the above algorithm is not efficient (polynomial inefficiency) neither theoretically since $W_s = O(n^{2.376})$ [15, 45] nor practically since $O(n^{2.81})$ algorithms are of practical use [3, 40, 17]. We will see in following sections how to overcome those problems.

1.1.7 Relations between PRAMs

We consider the cost of the execution of a parallel algorithm (defined on a CUMULATIVE-CRCW PRAM for instance) on a given PRAM with a fixed number of processors and with its own semantics for access in shared memory. Two cases are distinguished: when the number of processors is decreased and when memory access are restricted. We consider here only arithmetic costs. The main consequence is the existence of optimal – within a constant factor – simulations of a CRCW algorithm that uses an unbounded number of processors on an EREW machine with a fixed number of processors.

Theorem 1 Fine grain simulation with fewer processors - Brent's principle *[9, 28]. Let $\mathcal{A}$ be an algorithm that can be implemented to run in (arithmetic) parallel time T and work W_a on a given PRAM with an unbounded number of processors. If each local access corresponds to a global one, then $\mathcal{A}$ can be scheduled on the same PRAM, but with p processors, to run in (arithmetic) parallel time $T_p(n)$:*

$$\left\lceil \frac{W_a(n)}{p} \right\rceil \leq T_p(n) \leq \left\lfloor \frac{W_a(n)}{p} \right\rfloor + T(n) \tag{1.16}$$

It can be noted that this fine grain simulation does not take into account additive cost due to the computation of the schedule [12, 22].

Remark. In chapter 2, a constructive coarse grain simulation for DFGs where arithmetic nodes may represent a sequence of elementary instructions.

Theorem 2 Simulation with restricted access in global memory *[28, 38]. Let $\mathcal{A}$ be an algorithm that can be implemented to run in (arithmetic) parallel time T_p on a CUMULATIVE-CRCW PRAM with p processor. Then, $\mathcal{A}$ can be implemented on an EREW PRAM with p processors to run in time $O(T_p \log p)$.*

1.2 Increasing granularity

Efficient parallel algorithms require near-optimal work; obviously, the careful analysis of the smallest depth DFG induced by a sequential algorithm among the best is then of practical interest.

As a major example, sequential algorithms based on a partitioning of the problem into – many – independent subproblems have intrinsic parallelism if partitioning and merging (to recover the global solution) steps are either parallel or of neglected cost. This situation appears frequently in numerous divide&conquer algorithms (let us say *parallel divide&conquer*). As a computer algebra instance, modular methods based on Chinese remainder computations [2, 10] amounts to this scheme.

Once a fine grain fast parallel algorithm built, increasing granularity is required to obtain and efficient algorithm with coarse-granularity. In this section, the technique consisting in stopping recursivity is illustrated on the matrix product problem; we prove an optimal granularity for this problem.

1.2.1 Parallel divide and conquer

Let us consider the example of matrix multiplication using a standard bi-dimensional block algorithm:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}. \tag{1.17}$$

All block matrices products, of dimension $n/2$, can be multiplied in parallel. Applying recursively this splitting scheme leads to a parallel algorithm with cost:

$$O_a(\log n, n^3) \quad O_c(\log n, n^3) \tag{1.18}$$

Note that, since coefficient addition is associative, each entry in the output matrix may be computed as an iterated sum of n values. This allows the whole computation to take a time $\log n$ (instead of $\log^2 n$ if additions where performed naively at each step). This remark appears directly on the DFG description for a CUMULATIVE-CRCW PRAM 1.4: all final sums are made in O(1) time. But the splitting process, which involves no arithmetic operation but recursive forks (cf fig. 1.3.b), requires $O(\log n)$ time using recursive forks[4]. Another technique to obtain $O_a(\log n, n^3)$ consists in pipelining additions [1].

```
CumulProductTerm( a : in  E,
                  b : in  E,
                  c : out E )
begin
  c.Cumul<+>( a.Read()*b.Read() );
end

MatrixProduct( n : in  integer;
               a : in  array[1..n,1..n] of E,
               b : in  array[1..n,1..n] of E,
               c : out array[1..n,1..n] of E )
begin
  i, j, k : local integer;

  for i = 1..n loop
    for j = 1..n loop
      for k = 1..n loop
        fork CumulProductTerm( a[i,k], b[k,j], c[i,j] );
      end loop
    end loop
  end loop
end
```

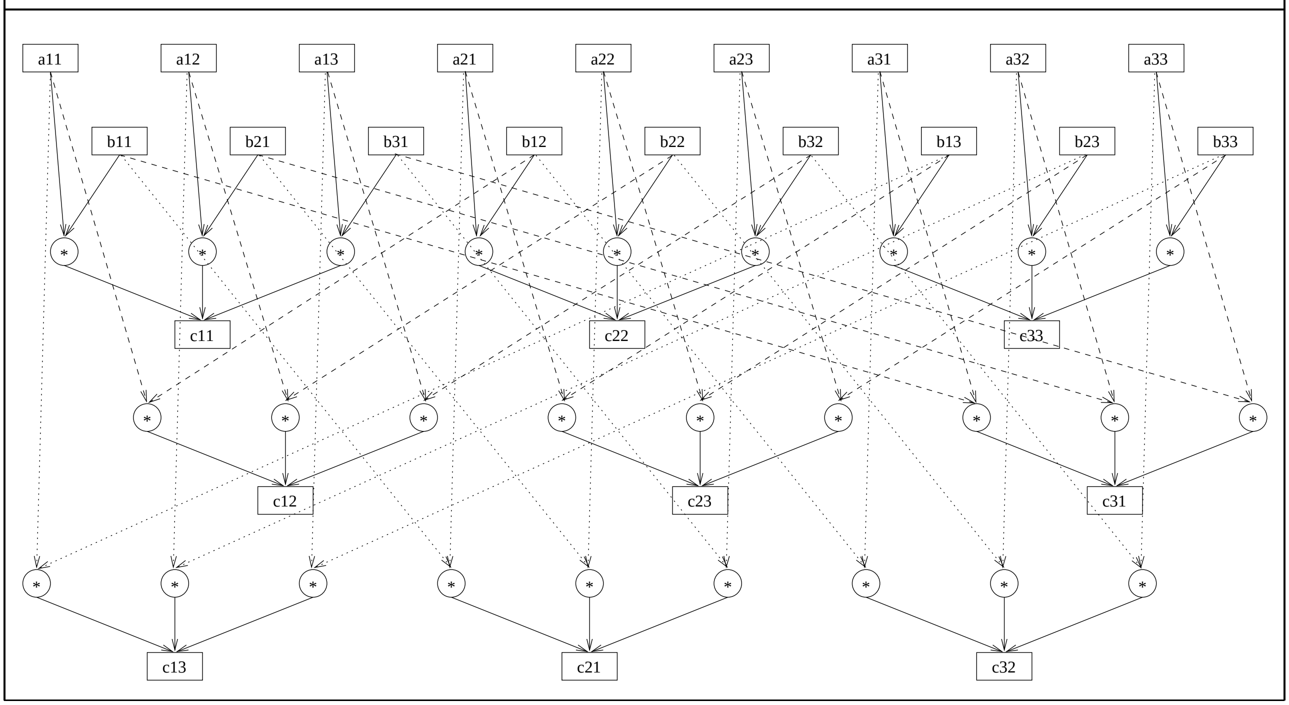

Figure 1.4: DFG of the multiplication of two 3×3 matrix (cumulative-CRCW)

[4]Note that the brute force program (fig. 1.4) which performs iteratively fork instructions requires $O_a(n^3, n^3)$!

Remark. The same strategy applied to Strassen's algorithm leads to a parallel algorithm with cost:

$$O_a(\log n, n^{\log_2 7}) \quad O_c(\log n, n^{\log_2 7}) \tag{1.19}$$

Optimal in work (on a semi-ring), this algorithm has granularity $g(n) = O(1)$: it is roughly equivalent to a recursive version of 1.15). In the next section, we detail how to increase granularity in order to build an efficient algorithm with coarse-granularity.

1.2.2 Minimizing communication work

Obtaining a coarse-granularity algorithm requires to minimize communications. This can be done by stopping the recursive parallel splitting process at a given depth, let us say when sub-matrices are of size lesser than k (i.e. depth $\log \frac{n}{k}$). Operations – resp. sums and products – on matrices of dimension k are then performed sequentially, using an optimal algorithm – resp. in time $O(n^2)$ and $O(n^3)$ –. The cost is then:

$$O_a\left(k^3 + \log n, n^3\right) \quad O_c\left(k^2 + \log \frac{n}{k}, \frac{n^3}{k}\right) \tag{1.20}$$

which gives an algorithm with granularity $g(n) = k$. We thus obtain a parallel efficient algorithm with arbitrary (polynomial) granularity.

Theorem 3 *For any g, $\log^{1/3} n \leq g \leq n$, two $n \times n$ matrices can be multiplied by an algorithm of granularity g with parallel cost:*

$$O_a\left(g^3, n^3\right) \quad O_c\left(g^2 + \log n, \frac{n^3}{g}\right).$$

The previous algorithm 1.20 proves the upper bound. $\Box$.

The following theorem gives lower bounds for communication costs. It shows that the previous algorithm achieves an optimal communication delay and an optimal granularity among algorithms that achieves an optimal communication delay.

Theorem 4 *Let $\mathcal{A}$ be an efficient parallel algorithm that multiplies two matrices of dimension n in time T using $(+, \times)$ only and performing $\Theta(n^3)$ operations. Then,*

$$C_d = \Omega\left(T^{2/3} + \log n\right) \qquad W_c = \Omega\left(\frac{n^3}{C_d^{1/2}}\right).$$

Since $\mathcal{A}$ is efficient, $T = O(n^\epsilon)$ with $\epsilon < 3$; by reduction from iterative sum, we thus have $C_d = \Omega(\log n)$.
Kerr [37, 1] shows the lower bound $\Omega(n^3)$ on the arithmetic work. Since $\mathcal{A}$ performs $\Theta(n^3)$ operations, its execution can be scheduled in time $\Theta(T)$ using $p = \frac{n^3}{T}$ processors. Let s_i, $1 \leq i \leq p$, be the number of shared memory access performed by processor i. We then have $W_c = \sum_{i=1}^{p} s_i$ and $C_d \geq \max_{i=1}^{p} s_i$. To obtain a lower bound on W_c and C_d, we use the following lemma [1, 25]: if a processor reads at most s elements of input matrices and computes at most s partial sums of their product, then this processor can compute no more than $s^{3/2}$ multiplicative terms for these

partial sums.
Applying this lemma to p_i which reads or writes at most s_i elements and since $\Omega(n^3)$ multiplicative terms are to be computed, we have:

$$\sum_{i=1}^{p} s_i^{3/2} = \Omega(n^3). \tag{1.21}$$

Bounding s_i by C_d and replacing p by $\frac{n^3}{T}$ leads to:

$$C_d = \Omega\left(T^{2/3}\right). \tag{1.22}$$

Noticing that $\sum_{i=1}^{p} s_i^{3/2} \leq C_d^{1/2} \sum_{i=1}^{p} s_i$, we obtain:

$$W_c = \Omega\left(\frac{n^3}{C_d^{1/2}}\right) \tag{1.23}$$

which concludes the proof $\square$.

Recursive multiplication algorithms. A similar study can be applied to other recursive matrix multiplication algorithms (e.g. Strassen). It leads also to efficient parallel algorithms with both polynomial speed-up and polynomial granularity that leads to performant implementations [17].

1.2.3 Conclusion

In this section, we have studied the DFG of a sequential algorithm, based on a divide&conquer scheme, that contains inherent parallelism. By halting the recursive process in order to minimize communications, we have exhibited a family of efficient parallel algorithms with arbitrary coarse-grain granularity.

Due to its practical interest, this technique has been successfully applied to various problems. One of significant interest in computer algebra is the discrete Fourier transform. The direct analysis of the FFT algorithm leads to a parallel algorithm with cost:

$$O_a(\log n, n \log n) \quad O_c(\log n, n \log n).$$

A clustering of elementary instructions (block clustering on the first $\frac{\log n}{2}$ steps and cyclic clustering on the last $\frac{\log n}{2}$ steps, cf fig. 1.5) leads to an algorithm with parallel cost [41, 39]:

$$O_a(\sqrt{n} \log n, n \log n) \quad O_c(\sqrt{n}, n).$$

This algorithm has polynomial speed-up, optimal work and achieves also optimal granularity [1].

The resulting algorithm is based on coupling a very fast parallel algorithm, optimal in time but requiring many communications, to a sequential one which minimizes communication. Such an algorithm is called “poly-algorithm”; the technique that underlies this coupling is called “cascading divide&conquer”.

Cascading divide&conquer may be applied in a more general context, by coupling a very fast parallel algorithm, yet requiring many operations, to a slower one which performs an optimal number of operations. This technique makes the building of very fast algorithms attractive even if the required number of operations is larger.

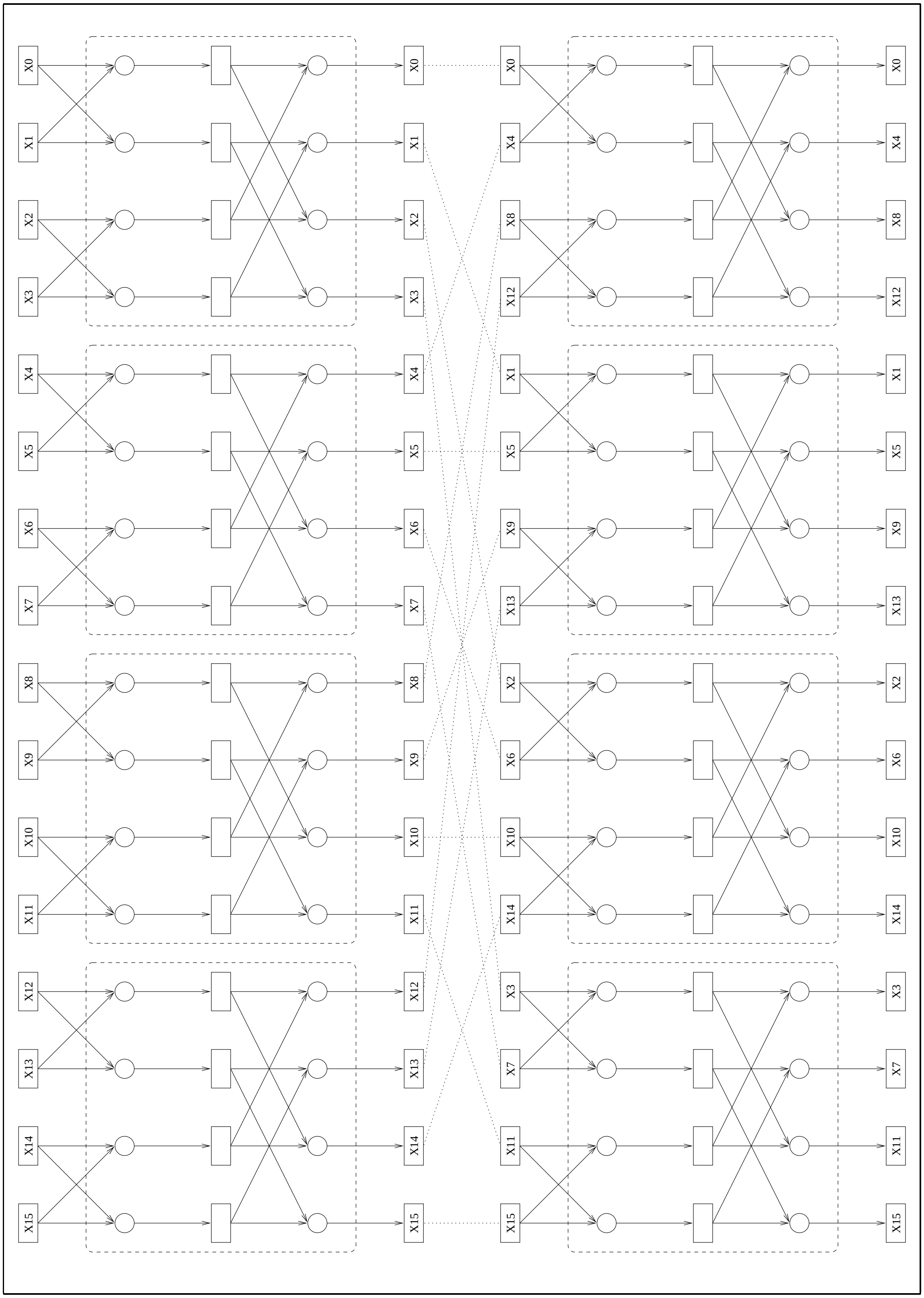

Figure 1.5: DFG of the EREW $O_a(\sqrt{n}\log n, n\log n)$ FFT algorithm of 16 points. There are $2\sqrt{n}$ arithmetic tasks (represented by square boxes embedding elementary operations and local dependencies), each corresponding to a sequential FFT computation on $\sqrt{n}$ points. For any task on the left, shared data dependencies imply a precedence relation with the $\sqrt{n}$ tasks on the right.

1.3 Breaking data-flow dependencies by redundancy and cascading divide&conquer

It may appear that DFGs related to a sequential algorithm contain data-dependencies that bound parallelism. Introducing redundant computations may then allow to break dependencies in order to minimize parallel time. Cascading divide&Conquer may then be used to obtain an optimal arithmetic work. In this section we illustrate this technique on the computation of the solution of a triangular linear system presented in [46]. We focus on communication costs.

Let A be an $n \times n$ nonsingular triangular matrix with coefficients in a field K. We assume by convenience $n = 2^m$. Let b a vector in K^n. We consider the computation of $x = A^{-1}b$.

1.3.1 DFG of the best sequential algorithm

The simple forward substitution algorithm has sequential cost $W_s(n) = \Theta(n^2)$. Direct analysis of its DFG (see fig. 1.6) gives its parallel cost:

$$O_a(n, n^2) \qquad O_c(n, n^2), \tag{1.24}$$

which leads to an algorithm with polynomial speed-up but small granularity $g(n) = O(1)$.

If entries of A are in global memory after initialization, we have $W_c(n) = \Omega(n^2)$. In a view to minimizing the communications involved by the algorithm itself, in the following we do not consider the access to A in the communication work $W_c(n)$.

In order to increase granularity, we consider a divide&conquer version of this algorithm [7]. Let A, b and x be divided into blocks:

$$A = \begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{1.25}$$

Here A_{11} is of size $h \times h$, x_1 and x_2 are of size h. We have:

$$A_{11}x_1 = b_1 \quad \text{and} \quad A_{22}x_2 = b_2 - A_{21}x_1. \tag{1.26}$$

where x_1 and x_2 are computed recursively using the same algorithm; $A_{21}x_1$ is computed using a scalar product (see 1.13). Note that the use of a pipeline scheme leads to the previous parallel cost 1.24.

We may then stop the recursive splitting when matrices are of size $k \times k$, and use sequential algorithms (triangular system inversion and matrix-vector product) on matrices of size lesser than k. The resulting parallel cost is:

$$O_a(nk, n^2) \qquad O_c\left(nk, \frac{n^2}{k}\right) \tag{1.27}$$

which leads to an algorithm with granularity $g(n) = O(k)$.

Theorem 5 *For any $\epsilon < 1$, a triangular nonsingular linear system can be solved by an efficient parallel algorithm of coarse granularity n^ϵ in time $O(n^{1+\epsilon})$.*

Choosing $k = n^\epsilon = o(n)$ in 1.27 proves the upper bound. □.

```
Update( x : out E,
        a : in  E,
        y : in E )
begin
  x.Cumul<+>( -a.Read()*y.Read() );
end


FinalDivision( x : in and out E,
               a : in         E )
begin
  x.Write( x.Read() / a.Read() );
end




TriangularSolve ( n : in  integer,
                  a : in  array[1..n, 1..n] of E,
                  b : in  array[1..n] of E,
                  x : out array[1..n] of E )
begin
  i,j : local integer;

  for i = 1..n loop
    x[i].Cumul<+>( b[i].Read() );
    fork FinalDivision(x[i], a[i,i]);
    for j = (i+1)..n loop
      fork Update(x[j], a[j,i], x[i]);
    end loop
  end loop
end
```

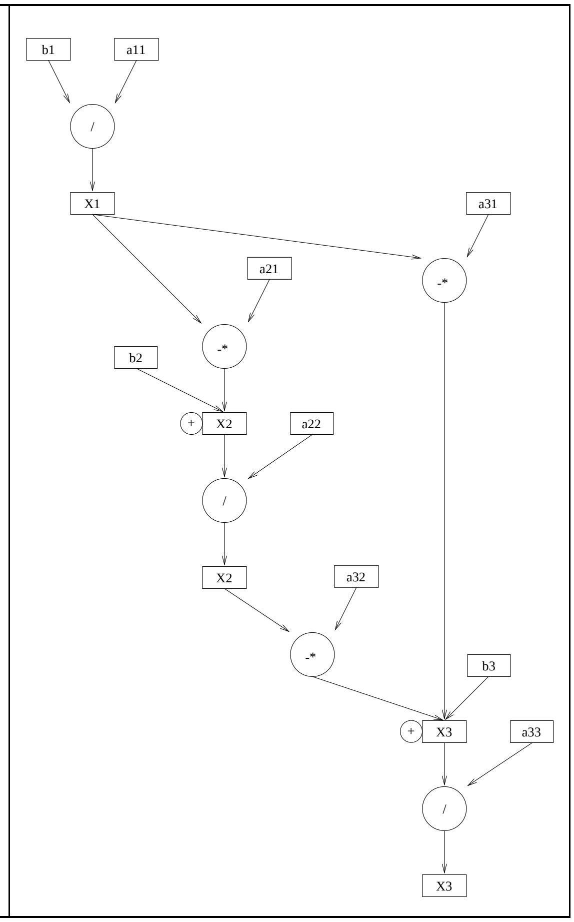

Figure 1.6: DFG for the solving of a 3×3 nonsingular triangular matrix

1.3.2 Breaking dependencies

The linear time lower bound on previous algorithm time comes from the dependency in formula 1.26 between computations of x_1 and x_2. This dependency may be broken by directly computing the inverses of the triangular nonsingular matrices A_{11} and A_{22}.

Consider the matrix A split in four blocks of dimension $n/2$ (1.25 with $h = n/2$). Then we have:

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & 0 \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{bmatrix} \tag{1.28}$$

From theorem 3, the product of two matrices of dimension n is computed with parallel cost $O_a(\log n, n^3)$. In the following, we will refer to this cost.

To compute the inverse of A from 1.28, we first compute recursively and in parallel A_{11}^{-1} and A_{22}^{-1}. Then we compute the last block of A^{-1} by performing sequentially two parallel matrix products. The parallel cost for inverting A is then:

$$O_A(log^2 n, n^3) \qquad O_c\left(log^2 n, \frac{n^3}{\log^{1/3} n}\right) \tag{1.29}$$

Once A^{-1} is computed, $x = A^{-1}b$ can be computed with the same cost. However, even if polylogarithmic in time, this algorithm has polynomial inefficiency. In the next paragraph, we use it on A_{11} in 1.26 in order to decrease parallel time.

Remark. The above algorithm is efficient for computing the inverse of a nonsingular triangular matrix. Note that by using fast matrix multiplication, the parallel cost is reduced to $O_a(\log^2 n, n^\omega)$ with $\omega < 2.38$ [46]. Besides, if computations are performed sequentially when the dimensions of the matrices are lesser than $k \geq n^\epsilon$, $(\epsilon < 1)$, the obtained algorithm is efficient and has polynomial speed-up and polynomial granularity.

1.3.3 Cascading divide&conquer to minimize time

The previous algorithm is not efficient but may be combined to the recursive sequential algorithm (formula 1.26). The trick is to use it on small dimension matrices (let us say h) when the overhead $O(h^3)$ due to the fast inversion of such a matrix becomes neglectible compared to coefficients updates (roughly nh). This leads to the following algorithm of Pan&Preparata [46].

Theorem 6 *The solution of a nonsingular triangular system can be computed in*

$$O_a(n^{1/2} \log n, n^2)$$

using a standard n^3 matrix multiplication algorithm.
If a fast n^ω multiplication is used then the parallel cost is:

$$O_a(n^{(\omega-2)/(\omega-1)} \log^2 n, n^2).$$

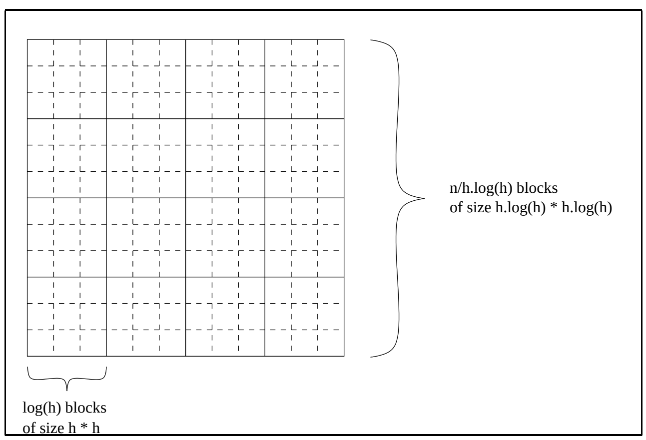

Figure 1.7: Splitting used for $h = 8$, $h \log h = 24$, $n = 96$

Following 1.27, let A be split in n^2/h^2 blocks of size $h \times h$. Though, note that a direct computation (see theorem 1.27) leads to a parallel time $O(n^{1/2} \log^2 n)$. To avoid the $\log n$ overhead factor in the parallel time, we proceed by gathering computation on $\log^2 h$ blocks.
Let $k = h \log h$; the matrix A may be seen as split in $(n/k)^2$ blocks, each block consisting in $\log^2 h$

sub-blocks of dimension h (cf fig. 1.7).
We use the sequential iterative algorithm on the $(n/k) \times (n/k)$ coarse grain matrix. At step i, we have to invert the triangular system corresponding to the diagonal block (i, i). For this computation, we first invert concurrently the $\log h$ diagonal sub-blocks of this block. Then, we update others sub-blocks of x_i. At the end of the step, blocks x_j, for $j > i$, are updated.
The algorithm is the following:

Initialization.

Let A be split into n/k blocks $M_{i,j}$ of dimension k ($k = h \log h$). For $1 \leq j \leq i \leq n/k$, let $M_{i,j}$ be split into $\log h \times \log h$ block $m_{i,j}^{k,l}$ of dimension h.
Let x be initialized to b and split according to A.

for $i = 1..n/k$ do

1. for $j = 1.. \log h$ do
   fork $\left(m_{i,i}^{j,j}\right)^{-1}$ = invert($m_{i,i}^{j,j}$).
   Using fast inversion and Brent's principle, the cost is $O_a(\log^2 h, h^3 \log h)$.
2. for $j = 1.. \log h$ do
   update x_i^j in parallel
   $x_i^j = \left(m_{i,i}^{j,j}\right)^{-1} \left(x_i^j - \sum_{l=1}^{j-1} m_{i,i}^{j,l} x_i^l\right)$
   Scalar product are performed in parallel: thus x_i is computed with a cost $O_a(\log^2 h, h^2 \log h)$.
3. for $j = i + 1..n/k$ fork update x_j in parallel
   $x_j = x_j - M_{j,i} x_i$
   Performing scalar product in parallel, the cost is $O_a(\log h, nh \log h)$.

The final cost is : $O_a(n \log^2 h/k, n/k \max(h^3 \log h, nh \log h))$. Since $k = h \log h$, it reduces to:

$$O_a(n \log h/h, \max(nh^2, n^2)),$$

and the optimal value for h is the larger one that leads to a work $W_a(n) = O(n^2)$. Thus, we choose $h = n^{1/2}$ and we obtain the upper bound.
The same technique is applied to obtain the upper bound when a fast matrix multiplication algorithm is used. □

1.3.4 Applications in linear algebra

Many linear algebra algorithms are based on a Gaussian elimination scheme: linear system solving, normal forms (Hessenberg, Smith, Frobenius, symbolic Jordan). Such a scheme provides parallel algorithms with polynomial speed-up: at each step, a transformation is computed that can then be applied in parallel to each coefficient of the matrix. For instance, solving a non-singular linear system using standard Gaussian elimination leads to a parallel algorithm with cost:

$$O_a(n, n^3) \quad O_c(n, n^2) \tag{1.30}$$

Moreover, very fast deterministic algorithms (polylogarithmic parallel time) are known for most problems [45, 24, 58, 57] but they are often inefficient ($W_a(n) = n^{O(1)} W_s(n)$). For instance, solving a non-singular linear system can be computed in parallel with cost:

$$O_a(log^2 n, n^{3+\alpha}) \tag{1.31}$$

with $\alpha = 1/2$ in characteristic zero [16, 50] and $\alpha = 1$ in the general case [11]. Applying the same cascading divide and conquer strategy leads to sub-linear parallel algorithms with optimal[5] work [46]:

$$O_a(n^{1/5} \log^2 n, n^3). \tag{1.32}$$

Remark. The same technique applied on Strassen formulation [56] (which may take benefit of fast $O(n^{2.376})$ matrix multiplication algorithms), does not succeed in the building of a sub-linear algorithm with parallel time n^β, $\beta < 1$.

1.3.5 Conclusion

In this paragraph, we have used bi-dimensional block matrix partitioning in order to:

- increase the granularity to build polynomial speed-up algorithms with polynomial granularity; the technique used is cascading divide and conquer with a sequential algorithm in order to decrease communication costs.

- decrease parallel time while preserving the work; the technique used is cascading divide and conquer with a very fast but inefficient algorithm in order to make the computation faster.

In [46], the same technique, called *work-preserving speed-up*, is applied to several linear algebra algorithms: LU factorization, inversion, quasi-inversion, solution of linear structured systems.

1.4 Randomization to decrease time or preserve work.

When an algorithm has a bounded degree of parallelism or a polynomial efficiency, randomization may help in order to either decrease time or preserve work, eventually both. This section illustrates both aspects on the computation of the rank of a matrix.

In computer algebra, randomization is most often introduced via the verification of a polynomial identity by evaluation on a random value. Testing whether a polynomial is identically zero can deterministically be solved by evaluating the polynomial, represented as a straight-line program, at a sufficient number of points. However, depending on the degree and on the number of indeterminates, such a deterministic test can require a huge number of evaluations. Following theorem, due to Schwartz [54], uses randomization in order to reduce this number while bounding the probability of failure.

[5] relatively to the standard $O(n^3)$ sequential algorithm

Theorem 7 *[54, 28] Let $P(x_1, \ldots, x_n)$ be a polynomial in the variables (x_i), $1 \leq i \leq n$, over a field K. Let I be a finite subset of K with cardinal c. Let $(\alpha_1, \ldots, \alpha_n)$ a vector selected at random in K^n. If P is not identically zero then*

$$\textit{Prob}\left(P(\alpha_1, \ldots, \alpha_n)\right) \leq \frac{\textit{deg}(P)}{c}.$$

Once a problem is reduced to the verification of a polynomial identity, this theorem allows to build a Monte-Carlo algorithm to solve it (for an introduction on Monte-Carlo and Las Vegas algorithms, see [36]). It is sufficient to build a parallel algorithm that evaluates the polynomial at a given input point. By choosing this point at random in a large enough finite subset[6] we obtain a Monte-Carlo algorithm whose probability of error is at most $1/2$. This technique may be applied in a very large framework [36, 28] and is commonly used in computer algebra [45] to build fast algorithms with optimal work. We illustrate it on the problem of computing the rank of a matrix.

In the following, A denotes a matrix of dimension $n \times n$ with coefficients in a field K. For the sake of simplicity, K is assumed infinite.

1.4.1 Randomization to suppress dependencies

The rank of a matrix can be computed using a standard pivoting Gaussian elimination. Similarly to 1.24, this results in an algorithm with parallel cost:

$$O_a(n, n^3) \quad O_c(n, n^2) \tag{1.33}$$

Contrary to triangular system solving, the computation scheme (DFG) is relatively unknown: coefficients to modify are determined at each step only once the pivot element has been chosen.

In [8], randomization is used in order to reduce the whole problem to a fixed DFG on which parallelization techniques can be applied. The algorithm is based on the following characterization of the rank: rank$(A) = r$ iff there exist two non-singular matrices L and C such that the principal minor of dimension r in LAC is non zero while principal minors of dimension larger than r are zero. Moreover, L and C can be taken at random with a high probability of success: the use of theorem 7 to evaluate this probability requires to express the problem as a polynomial identity.

Let $\delta_i(L, C)$ denote the principal minor of dimension i of LAC. Due to multi-linearity of the determinant, δ_i is a polynomial of degree $2n$ with indeterminates $L_{i,j}$ and $C_{i,j}$ ($1 \leq i, j \leq n$). Previous rank characterization leads to the following polynomial identities:

$$\begin{array}{ll} \delta_i \neq 0 & 1 \leq i \leq r \\ \delta_i = 0 & r < i \leq n \end{array} \tag{1.34}$$

This suggests the following Monte-Carlo algorithm to compute r:

1. Choose two random non-singular matrices L and C with coefficients in a finite subset of cardinal c of K;

2. Compute: $M = LAC$;

[6]Note that, if K is not large enough, this may require to work in an extension of K [24].

3. For $1 \leq i \leq n$, compute $d_i = \det(M_i)$ and let $d_0 = 1$;

4. Return $s = \text{Max}_{k=0,\ldots,n}\{k/d_k \neq 0\}$.
   (Note that step 3 and 4 may be replaced by a logarithmic search to compute s).

In any case, $s \leq r$. The probability of error, which occurs when $s < r$, corresponds to executions where the evaluation d_r of polynomial δ_r is zero although δ_r, of degree $2n$, is not identically zero. From theorem 7, this probability is bounded by $\frac{2n}{c}$. Choosing $c = 4n$ results in a Monte-Carlo algorithm with probability of error lesser than $\frac{1}{2}$.

Arithmetic cost is dominated by the computation of the n determinants. If Chistov's method [11] is used (see chapter 4), this cost is:

$$O_a(log^2 n, n^{\omega+1}) \tag{1.35}$$

In order to improve efficiency, determination of s may be computed using a logarithmic scheme instead of the previous brute force method. Using the efficient randomized algorithm of Kaltofen and Pan [33] to compute the determinant (see chapter 4), the parallel cost becomes

$$O_a(log^3 n, n^{\omega} \log n), \tag{1.36}$$

Note that such an algorithm uses mainly randomization in order to provide a parallel computation scheme for the rank.

1.4.2 From Monte-Carlo to Las Vegas

The building of a Las Vegas algorithm from a Monte-Carlo one consists mainly in verifying that the output is a correct solution to the initial problem. Such a verification is easy from the previous algorithm; it suffices to verify that all columns (resp. rows) of the matrix $M = LAC$ are linear combinations of s independent columns (resp. rows) in M, s being the output of the algorithm.

Consider the following splitting for M, the first block M_{11} being of size $s \times s$:

$$M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}. \tag{1.37}$$

M_{11} is a non-singular matrix. Let $X = M_{21}M_{11}^{-1}$ and $Y = M_{11}^{-1}M_{12}$; note that X and Y^t are of size $(n-s) \times s$. Since L and C are non-singular, A is of rank s iff the last $(n-s)$ rows and columns of M are respectively linear combinations of the s first ones. This relies on the following identities:

$$\left\{ \begin{array}{c} [M_{21} \;\; M_{22}] = X[M_{11} \;\; M_{12}] \\ \begin{bmatrix} M_{12} \\ M_{22} \end{bmatrix} = \begin{bmatrix} M_{11} \\ M_{12} \end{bmatrix} Y \end{array} \right. \tag{1.38}$$

Assuming a Las Vegas algorithm to compute M_{11}^{-1} with parallel cost $O_a(\log^2 n, n^{\omega} \log n)$ ([33], see chapter 4), those identities can be verified with a parallel cost:

$$O_a(\log^2 n, n^{\omega} \log n). \tag{1.39}$$

This results in an optimal randomized Las Vegas algorithm to compute the rank.

In this algorithm, randomization is strongly used for preconditionning the input (computation on LAC instead of A) in order to suppress data dependencies that bounds parallelism. A natural question is then the existence of a fast deterministic algorithm, i.e. with few dependencies. In [44], Mulmuley provided such a deterministic algorithm for computing the rank: it achieves parallel time $O(\log^2 n)$ but polynomial inefficiency. Then, randomization is required to provide efficiency.

1.4.3 Randomization to provide efficiency

Based on a generalization of a method developed in [27] for arbitrary fields, Mulmuley algorithm [44] reduces the problem of computing the rank to the computation of a characteristic polynomial in an extension of the ground field K.
In the following, A is assumed symmetric; this is done without loss of generality since

$$\text{rank}(A) = \frac{1}{2}\text{rank}\left(\begin{bmatrix} 0 & A \\ A^t & 0 \end{bmatrix}\right).$$

Theorem 8 *[44] Let A be a square symmetric matrix over a field K and let m be the highest integer such that x^m divides the characteristic polynomial $\xi_{A_Z}(x) = \sum_{i=0}^{n} a_i(z)x^i$ of the matrix A_Z over $K(z)$:*

$$A_Z = \begin{bmatrix} 1 & & & \\ & z & & 0 \\ 0 & & \ddots & \\ & & & z^{n-1} \end{bmatrix} A.$$

Then rank$(A) = n - m$.

Deterministic parallel algorithms for computing the characteristic polynomial in parallel time $O(\log^2 n)$ are known [16, 11] (cf chapter 4) but they have polynomial $O(n)$ inefficiency. Even if we assume an optimal algorithm for computing the characteristic polynomial, the cost of the above algorithm would be:

$$O_a(\log^2 n, n^\omega n log^{O(1)} n) \tag{1.40}$$

Since $a_i(z)$ are polynomials of degree $O(n)$, a way to obtain efficiency is to get rid off polynomial arithmetic on K using evaluation at a random value.
Moreover, efficient $O_a(log^2 n, n^\omega \log n)$ randomized algorithm are known for computing the minimal polynomial. Multiplying A_Z by a random non-singular matrix over K results, with high probability, in a matrix with distinct eigenvalues; then, minimal and characteristic polynomial are equal.
Those two steps of randomization results in the following efficient Monte-Carlo algorithm for computing the rank:

1. Choose a random non-singular matrix P;

2. Choose a random value z in K (or in an extension if K is too small);

3. Compute the minimal polynomial $\xi_{PA_z}(x)$ of the matrix PA_z;

4. Return $n - m$ where m is the highest integer such that x^m divides $\xi_{PA_z}(x)$.

The parallel cost is then:

$$O_a(\log^2 n, n^\omega \log n) \tag{1.41}$$

which results also in an efficient Monte-Carlo algorithm.

Remark. The above algorithm is very close to the one presented in 1.4.1; Mulmuley algorithm can effectively be considered as an inefficient deterministic version of 1.4.1. This is not surprising since both randomized algorithms solve efficiently the same problem. However, we point out two different motivations for the use of randomization.

1.4.4 Conclusion

In the above examples, randomization is used to provide work-optimal computations from either slow or fast but not efficient deterministic algorithms. Due to the fact that only randomized algorithms are known for computing efficiently the solution of linear systems in polylogarithmic time ([33] cf chapter 4), randomization is an important tool in parallel computer algebra.

1.5 Parallel time complexity and NC Classification

An efficient parallel algorithm achieves polynomial speed-up within an optimal (or near optimal) number of operations. Obtaining bounds on the parallel time required to solve a given problem within a reasonable number of operations is then of fundamental interest. Moreover, as detailed in previous sections, very fast parallel but inefficient algorithms may be of practical interest if they can be coupled to an efficient but slow algorithm.

In the framework of parallel complexity, NC class [13] which includes polynomial sequential time problems that have a polylogarithmic parallel time plays an important role [35]. The parallel model used in the formal definition of NC is log-uniform family of boolean circuits [53]. NC^k is the class of problems that can be solved by such a family with depth $O(\log^k n)$ and $n^{O(1)}$ boolean gates[7]. For instance, integer arithmetic ($+$, $-$, $\times$ and Euclidean division) lies in NC^1. Introduction of gates that deliver in output a random bit allows to define corresponding randomized classes: RNC for Monte-Carlo circuits and ZNC for Las Vegas ones. Problems P-complete [28, 49, 35] are in NC only iff $NC = P$; among them, the *monotone circuit value problem* (MCVP) consists in the evaluation of a boolean circuit, roughly equivalent to a DFG with boolean nodes as defined in this chapter. The integer greatest common divisor remains an open question; only sub-linear $O(\frac{n}{\log n})$ algorithms are known [34, 35].

The algebraic extension [61] of this primitive model allows to build circuits which gates compute compute arithmetic operations in an algebraic domain. A gate testing nullity ($? = 0$) is introduced in order to mix boolean and arithmetic operations. For instance NC^k_F (F stands for *field*) is the class of problems that can be solved by log-uniform family of circuits whose gates perform

[7] Gates compute bounded fan-in boolean operations (*or*, *and* and *not*) and have unbounded fan-out [26]. Extensions to unbounded fan-in gates leads to class AC [29].

arithmetic operations in any field, i.e. $+$, $-$, $\times$, $/$ and $? = 0$. Complexity of basic computer algebra problems has been extensively studied [8, 13, 59, 60, 35, 45]. Polynomial arithmetic ($+$, $-$, $\times$ and Euclidean division) lies in NC_F^1 [45]. An important class is DET_F which contains problems NC^1-reducible to the determinant of a matrix; matrix powering is complete for DET_F. DET_F is included in NC_F^2. Most of linear algebra problems lie in NC_F^2: rank, null-space, minimal and characteristic polynomial, gcd of many polynomials [8, 44], Hermite normal form of polynomial matrices [31], Smith and symbolic Jordan forms [52, 58, 57, 21]. Note that those problems admit an optimal $O_c(\log^2 n, W_s(n))$ parallel algorithm but using randomization [33, 23, 24, 45]. Though, in certain cases, some general techniques are known to remove randomness without increasing the work [42], no work optimal deterministic algorithms with poly-logarithmic time are known for those problems.

As it appears for most computer algebra problems studied in this chapter, parallel algorithms often appear as a restructuration of sequential ones, taking into account algebraic properties of the arithmetic operations involved. Although evaluation of a boolean circuit is P-complete, several algorithms have been developed to evaluate arithmetic DFGs (also called straight-line programs) taking benefit of the underlying structure. In a semi-ring, DFG that are trees can be evaluated in $O(\log n)$ time without increasing the number of operations performed [9]. Any DFG performing n operations in a semi-ring and whose outputs are of arithmetic degree[8] d can be evaluated in $O_a(\log n \log(nd), n^3)$ [32]. This result has been extended to DFGs performing operations in a lattice [51]. A more general simulation of a RAM machine on a PRAM one [43] shows that any DFG can be evaluate in parallel on an unbounded number of processors with polynomial speed-up.

1.6 Conclusion

This chapter overviews the PRAM framework (execution model and main algorithmic techniques) in which parallel algorithms are built and analyzed. The macro data-flow graph (DFG) related to the execution plays a central role: it describes data-dependencies between blocks of instructions.

Abstract measures used to analyze algorithms are *depth* and *work*; *arithmetic* and *communication* costs are distinguished. The one corresponds to operations performed (macro-instructions nodes) while the other to access in the shared memory (data dependencies nodes). Arithmetic work and depth are used for many years to analyze performances of parallel algorithms [9, 55, 35, 28, 6]. Due to experimental constraints, relevance of communications costs (i.e. total communication traffic – work - and total communications delay) has been pointed out to obtain practical performant programs [5, 19]. Since minimizing communications overhead and minimizing parallel time are antagonist, good trade-offs have been studied for several common algorithms [47, 1, 48]. Granularity, defined as the arithmetic-to-communication works ratio, appears as a good parameter.

[8] In such a DFG, any output may be equivalently seen as a polynomial whose indeterminates are the inputs. The arithmetic degree is then the maximal degree of polynomials corresponding to the outputs.

Bibliography

[1] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of PRAM's. *Theoretical Computer Science*, 71:3–28, 1990.

[2] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[3] D. Bailey. Extra high-speed matrix multiplication on the cray-2. *SIAM J. Sci. Sta. Comput.*, 9:603–607, 1988.

[4] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*. Springer-Verlag, Berlin, 1990.

[5] D. Bertsekas and J. Tsitsiklis. *Parallel and distributed computation*. Prentice-Hall, New York, 1989.

[6] G. E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[7] A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems* . Elsevier, New-York, 1975.

[8] A. Borodin, J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and gcd computations. *Information and Control*, 52:241–256, 1982.

[9] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974.

[10] T. Bubeck, M. Hiller, W. Küchlin, and W. Rosentiel. Distributed symbolic computation with DTS. In *Proc. of IRREGULAR'95, Lyon, France*, pages 231–248. Springer-Verlag LNCS 980, Sep. 1995.

[11] A. L. Chistov. Fast parallel calculation of the rank of matrices over a field of arbitrary characteristic. In *Proceedings of Fundamentals of Computation Theory'85*, pages 63–68. Springer-Verlag LNCS 199, 1995.

[12] R. Cole and U. Vishkin. Approximate Parallel Scheduling. Part I : The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM Journal on Computing*, 17(1), 1988.

[13] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[14] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines with simultaneous writes. *SIAM Journal on Computing*, 15:87–97, 1986.

[15] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[16] L. Csánky. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing*, 5:618–623, 1976.

[17] B. Dumitrescu, J.-L. Roch, and D. Trystram. Fast matrix multiplications algorithms on mimd architectures. *Parallel Algorithms and Applications*, 4(2), 1994.

[18] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, 1978. ACM Press.

[19] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, Reading, MA, 1995. http://www.mcs.anl.gov/dbpp.

[20] K. A. Gallivan, R. Plemmons, and A. H. Sameh. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Review*, 32(1), 1990.

[21] T. Gautier and J.-L. Roch. NC^2 computation of a gcd-free basis and application to parallel algebraic number computations. In E. Kaltofen, editor, *Parallel Symbolic Computation (PASCO'97)*, 1997.

[22] T. Gautier, J.-L. Roch, and G. Villard. Regular versus irregular problems and algorithms. In *Proc. of IRREGULAR'95, Lyon, France*, pages 1–26. Springer-Verlag LNCS 980, Sep. 1995.

[23] M. Giesbrecht. Fast algorithms for matrix normal forms. In *33rd IEEE Symposium FOCS, Pittsburgh*, pages 121–130, 1992.

[24] M. Giesbrecht. *Nearly optimal algorithms for canonical matrix forms*. PhD thesis, University of Toronto, Department of Computer Science, Canada, 1993.

[25] J. W. Hong and H. T. Kung. I/O complexity: the red-blue pebble game. In *Proc. 13th ACM Annual Symposium on Theory of Computing*, pages 133–139, 1981.

[26] H. Hoover, M. Klawe, and N. Pippenger. Bounding fan-out in logical networks. *J. ACM*, 31:13–18, 1984.

[27] O. Ibarra, S. Moran, and L. E. Rosier. A note on the parallel complexity of computing the rank of order *n* matrices. *Information Processing Letters*, 11:162, 1980.

[28] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Massachussets, 1992.

[29] D. Johnson. A Catalog of Complexity Classes. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 67–161. Elsevier, 1990.

[30] E. Kaltofen. Parallel algebraic algorithm design. Technical report, Rensselaer Polytechnic Institute, 1989. Lecture notes for a tutorial, ISSAC'89.

[31] E. Kaltofen, M.-S. Krishnamoorthy, and B. Saunders. Parallel algorithms for matrix normal forms. *Linear Algebra and its Applications*, 136:189–208, 1990.

[32] E. Kaltofen, G. Miller, and V. Ramachandran. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM Journal on Computing*, 17:687–695, 1988.

[33] E. Kaltofen and V. Pan. Processor efficient parallel solutions of linear systems over an abstract field. In *Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 180–191, Hilton Head, SC, 1991. ACM Press.

[34] R. Kannan, G. Miller, and L. Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM Journal on Computing*, 16-1:7–16, January 1987.

[35] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 869–932. Elsevier, 1990.

[36] R. M. Karp. An introduction to randomized algorithms. *Disc. Appl. Math.*, 34:164–201, 1991.

[37] L. R. Kerr. *The effect of algebraic structure on the computational complexity of matrix multiplications*. PhD thesis, Cornell University, New-York, 1970.

[38] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[39] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing:Design and Analysis of Algorithms*. Benjamin Cummings, Redwood City, 1994.

[40] J. Ladreman, V. Pan, and X.-H. Sha. On practical acceleration of matrix multiplication. *Linear Algebra and its Applications*, 1992.

[41] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays – Trees – Hypercubes*. Morgan Kaufmann, New-York, 1992.

[42] M. Luby. Removing Randomness in Parallel Computation without a Processor Penalty. *J. Computer and System Sciences*, 47:250–286, 1993.

[43] L. Mak. Parallelism always help. *SIAM Journal on Computing*, 26(1):153–172, 1997.

[44] K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7(1):101–104, 1987.

[45] V. Y. Pan and D. Bini. *Polynomial and Matrix Computations I*. Birkhauser, Boston, 1994.

[46] V. Y. Pan and F. P. Preparata. Work-preserving speed-up of parallel matrix computations. *SIAM Journal on Computing*, 24(4), 1995.

[47] C. H. Papadimitriou and J. D. Ullmann. A communication-time tradeoff. *SIAM Journal on Computing*, 16:639–646, 1987.

[48] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[49] I. Parberry. *Parallel Complexity Theory*. Pitman, London, 1987.

[50] F. P. Preparata and D. V. Sarwate. An improved parallel processor bound in fast matrix inversion. *Information Processing Letters*, 7:148–150, 1978.

[51] N. Revol and J.-L. Roch. Parallel evaluation of arithmetic circuits. *Theoretical Computer Science*, 162:133–150, 1996.

[52] J.-L. Roch and G. Villard. Fast parallel computation of the Jordan normal form of matrices. *Parallel Processing Letters*, 6(2):203–212, 1996.

[53] W. Ruzzo. On uniform circuit complexity. *J. Computer and System Sciences*, 22, 3:365–383, 1981.

[54] J. Schwartz. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM*, 27(4):701–717, 80 1980.

[55] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel Max-Flow algorithm. *Journal of Algorithms*, 3:128–146, 1982.

[56] V. Strassen. Gaussian elimination is not optimal. *Numerische Math.*, pages 354–356, 1969.

[57] G. Villard. Fast parallel algorithms for matrix reduction to normal forms. Appli. Alg. Eng., Comm. Comp., to appear.

[58] G. Villard. Fast parallel computation of the Smith normal form of polynomial matrices. In *International Symposium on Symbolic and Algebraic Computation, Oxford, UK*, pages 312 – 317. ACM Press, July 1994.

[59] J. von zur Gathen. Parallel algorithms for algebraic problems. *SIAM Journal on Computing*, 13:802–824, 1984.

[60] J. von zur Gathen. Parallel arithmetic computations : a survey. In *Proc. 12th Int. Symp. Math. Found. Comput. Sci., Bratislava*, pages 93–112. LNCS 233, Springer-Verlag, 1986.

[61] J. von zur Gathen. Algebraic complexity theory. *Ann. Rev. Comput. Sci.*, 3:317–347, 1988.

Chapter 2

Programming models and scheduling

In order to analyze performance of algorithms, a formal model is needed to take the costs into account. The success of the PRAM model is mainly due to the fact that it does not attempt to represent any parallel architecture but can be mapped onto various ones. Moreover, the simulation on a realistic machine can be made efficient (up to a constant related to the granularity), provided many processors of the PRAM are mapped onto a single processor of a host machine. This success is brought to evidence by the fact that most of the tricks used to optimize practical performances when programming on a given architecture are relevant to algorithmic techniques that are theoretically justified on the PRAM model.

Given an algorithm (let us say a macro data-flow graph – DFG – as presented in chapter 1) and a particular multiprocessor architecture, the problem then is reduced to:

- find a good (the best) schedule of the DFG;
- implement the resulting algorithm in a programming language.

Only now, the performance of the program, i.e. the completion time of an execution, may be determined. Assuming fixed the initial algorithm, the machine and the input, this performance depends directly on the scheduling strategy. Tuning the program ammounts to improving the schedule it implements.

This chapter presents the main techniques used to schedule data-dependencies graph (DFG) on a given architecture. As presented in chapter 1, a DFG is the abstract representation of the execution of a particular program on a specific input data x. A fine grain description (elementary instruction, elementary data dependency) is unrealistic for executions requiring hours of computation time.

We will thus assume that arithmetic nodes of the DFG correspond to sequence of instructions: each arithmetic node is then weighted by the number of elementary instructions it performs. Arithmetic depth $T(x)$ and work $W_a(x)$ are evaluated taking into account nodes weights. $T(x)$ is a lower bound of the minimal time required by any schedule ignoring communications times.

$W(x)$ is the exact number of operations required by a sequential execution of the algorithm. Since the best schedule may replicate some arithmetic nodes in order to minimize completion time, note that $W(x)$ is also a lower bound on the number of operations performed by any schedule.

Similarly, transition nodes may correspond to a complex data structure (not a single word); each transition node is weighted by the size of the data it corresponds to. Communication delay $C_d(x)$ and work $W_c(x)$ are also evaluated accordingly. Ignoring arithmetic time, $C_d(x)$ is an upper bound on the minimal communication time required by the best schedule for an infinite number of processors. $W_c(x)$ is an upper bound on the number of remote access (communications) performed by any schedule.

As straightened in the previous chapter, the initial parallel algorithm is assumed efficient, i.e. $W_a(n) = \Theta(W_s(n)$ where $W_s(n)$ is the time of the best known (uniform) sequential algorithm, n being the size of the input. Moreover, in order to make performance evaluation with x in input, we assume that there exists a constant K such that:

$$\forall x, |x| > n_0 : \quad W_a(x) < KW_s(x) \tag{2.1}$$

Note that, for a given input x, DFG$_x$ may be known only after completion: instructions or transitions nodes and edges are dynamically built. In the language ATH introduced in chapter

1, those nodes are created either by execution of a `fork` instruction or access to a shared data. Similarly, the cost of any instruction node (resp. size of data related to any transition) is known only after completion of the instruction (resp. communication). In such a general context, DFG$_x$ has to be scheduled using an on-line algorithm. Related to a functional programming model, most of computer algebra algorithms present such a dynamic behavior; we thus focus on on-line scheduling algorithms.

Organization of the chapter is as follows. In the first section, specific characteristics of asynchronous distributed architectures are recalled. Costs of basic operations are modeled by the *LogP* model introduced in [15]. Basic mechanisms allow parallel and distributed programming: communications, threads, remote memory access and synchronizations tools. In the second section, the scheduling of a PRAM algorithm on such a machine is discussed. Approaches may be distinguished in two classes. The first one [54, 28] is based on the simulation of a PRAM machine on a given architecture: the execution of the parallel algorithm is managed via the simulation. Global synchronization and emulation of the shared memory, which are at the basis of the PRAM model, are key points. The second one [26, 51, 38, 50, 5, 19] is based on the direct scheduling of the DFG. The execution of the algorithm is handled by a scheduling algorithm. Both approaches are motivated by the availability of provably good approximation algorithms to solve the underlying theoretical problems (permutation routing [48, 42, 55, 40] or DAG off-line and on-line scheduling [29, 49, 13, 36, 47, 14, 8, 6, 30]).
The last section focuses on on-line scheduling algorithms which are of main interest in computer algebra. We recall upper and lower bounds on the competitive-ratio without taking into account scheduling and communication overheads. As a corollary, we exhibit a list-scheduling algorithm which achieves optimal simulation of any efficient PRAM algorithm, taking into account those overheads. Finally, we overview some programming languages or libraries based on those approaches, focusing on the one suited to computer algebra algorithms. We describe an effective implementation of the theoretical language ATH introduced in chapter 1, ATHAPASCAN, which achieves provably performances.

2.1 Asynchronous distributed architectures

2.1.1 Realistic models of distributed architectures

There is an apparent convergence in the field of distributed architectures which are similar to a network of workstations. A parallel machine consists in a set of independent processors, each with considerable local memory, linked by an interconnection network. Fundamental differences with the PRAM model are the following (compare 2.1 to 1.1 in 1):

- **asynchrony**: each processor works independently with its own local memory; there are no global synchronization.
- **contention**: the network is a resource with bounded access.

Like the local PRAM introduced in chapter 1, two levels of access may then be distinguished: local and remote access (parallel machines are often called NUMA for non-uniform memory access[1].

[1] Note that this non-uniformity appears also at the processor level between cache and RAM access.

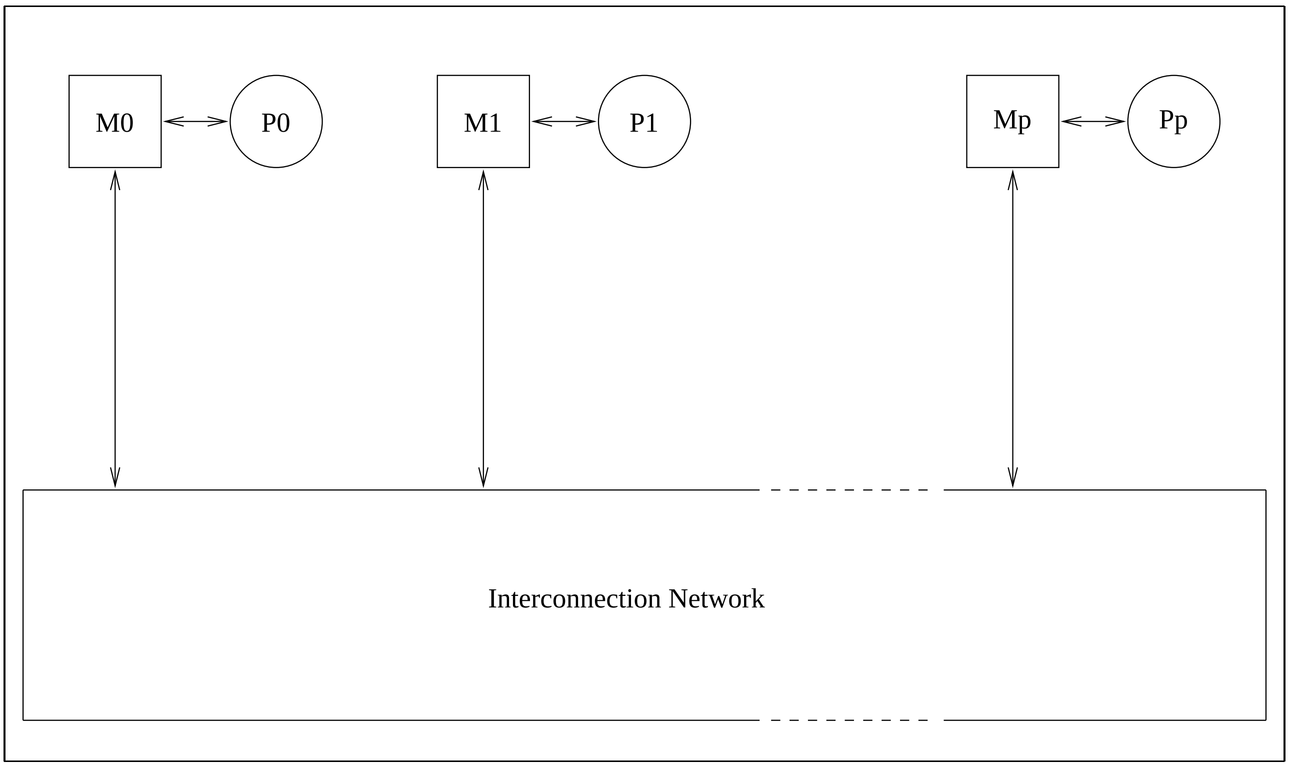

Figure 2.1: General structure of a distributed architecture. *Differences with the PRAM presented in chapter 1 are the absence of a global sequencer and contention for access to the network.*

Costs of remote access are mainly characterized by two factors:

- **bandwidth**: the rate at which each processor can access memory;
- **latency**: the time between making a remote access request and receiving the reply. Latency accounts for overheads involved in resource allocation (solving contention on network), duration of communication (related to physical distance).

The network bandwidth that is available on recent parallel computers ($>$ 1 GB/s on SGI Power Challenge, Cray T3E, SUN HPC) and even on local networks (typically 1 Gb/s using Mirynet connection or DEC Memory Channel) is becoming large enough compared to the bandwidth to local memory; thus it appears less and less as a bottleneck. However, latency is a more serious problem since it is bounded by physical limits.

Several variations of the PRAM model have been proposed in order to take into account those practical constraints [15]: memory contention [40, 54, 42, 45], asynchrony [27], memory hierarchy [3, 34], latency and bandwidth [47, 1]. Considering that point-to-point communication is a basic primitive, the model *LogP* proposed in [16] characterizes a distributed architecture by the following parameters (fig. 2.2):

L : *latency*: an upper bound on the delay incurred in communicating an unit size data (i.e. a small number of words) from its source to its destination; an extension to longer messages has also been developed [2].

o : *overhead*: the time a processor is engaged in the transmission or reception of a message;

g : *gap*: minimum time interval between consecutive message transmissions or receptions. The reciprocal of g corresponds to the available communication bandwidth per processor; it is denoted σ in [47].

P : the number of *processors*.

This model has been successfully used on different architectures to predict the execution time of some parallel algorithms [16, 20]. As a consequence, classical balanced tree schemes used on the PRAM to perform iterated sum or broadcast appear as non optimal [41].

As a conclusion, the portability of a parallel program cannot be achieved if the characteristics of the target architecture are not taken into account. Notingly, the communication parameters, that are partly modeled by *LogP*, have significant influence on the performances.

2.1.2 Basic programming tools

Reliable message-passing communication is the lowest-level feature required for programming a distributed architecture. It allows both to exchange data between processors (the basic functionality of the PRAM shared-memory) and to express synchronization (the functionality ensured by the sequencer of the PRAM).

Since 10 years, several message basic interfaces have been built on top of the low level ones provided on any specific architectures in order to allow portable programming. Most famous ones are PVM [24] and MPI [53]. MPI has been standardized [18] and is nowadays available

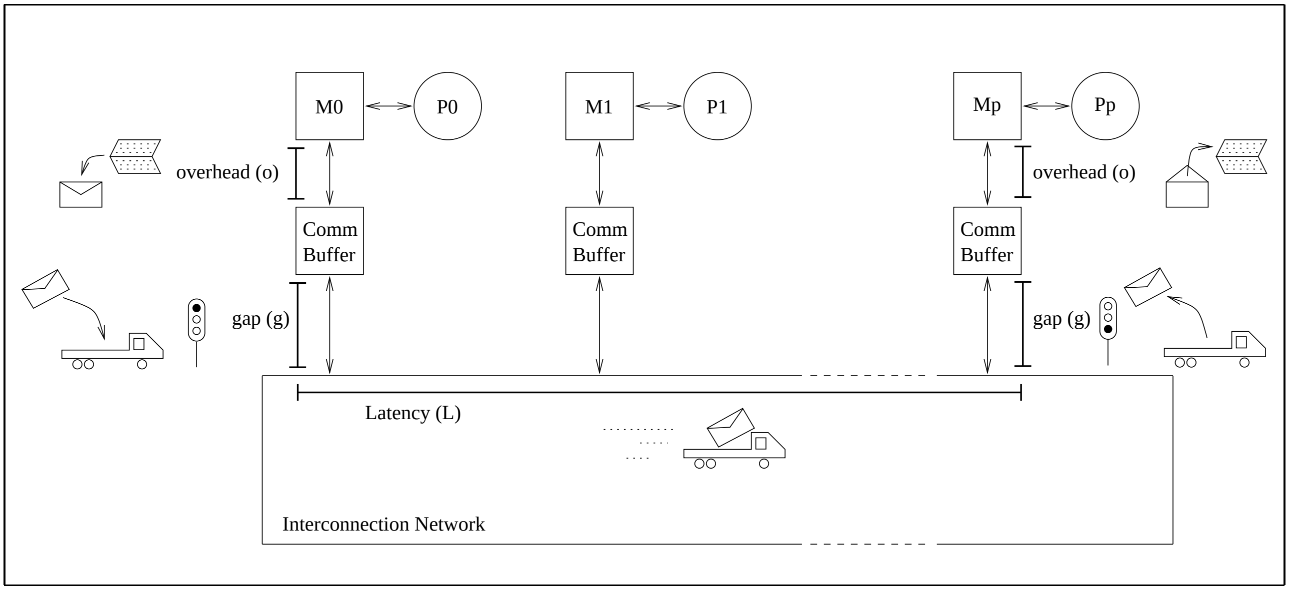

Figure 2.2: Communication cost parameters in the LogP model.

on any distributed architecture or network of workstations. Basic features of MPI are point-to-point and (blocking) collective communications, communication contexts (*communicators*), user-defined data-types. Other extensions concern remote memory access, parallel input and output (MPI-F), active messages and dynamic process control.

In order to hide the communication latency by arithmetic computations, two tools may be used: asynchronous communications and threads. Threads are lightweight processes which require a small overhead for context switching. They are handled directly in the source program: a standard interface, POSIX, has been defined [10]. Threads have firstly been defined for concurrent programming and efficient use of SMPs (Shared Memory Processor) on a single node. Since threads access concurrently the same memory space, synchronization tools are provided for atomicity, such as locks and semaphores (sometimes monitors).

Threads are well suited to hide latency on a distributed architecture: when a thread waits for the result of a communication, it may be preempted and a ready one scheduled. Thus, several portable programming interfaces have been built to couple a message-passing library (usually not thread-safe) and a thread library (available on a single node), providing an easy way to the user for lightweight remote procedure calls or active messages [21, 46, 9].

2.1.3 Shared virtual memory

On many distributed architectures, remote memory access are possible: they provide a virtual shared memory analogous to the one of the PRAM. On such machines, specific hardware allows to load transparently a local or remote data in the cache of a processor. In order to hide the latency of remote access, prefetching and multi-threading is used.

The simulations of the PRAM shared memory on a distributed architecture use hash functions (randomly chosen from a universal class) to map shared memory cells onto the ones of the architecture (i.e. memory modules) [48, 42]. The delay of a simulation is the time required for a single access. It is related to the evaluation of the hash function, the memory contention (when several access to a same module occur), and the routing time if the network is not complete. In [48], a simulation with delay $\Theta(\log p)$ of an EREW PRAM on a butterfly network is given. In [40], randomized simulations of EREW and CRCW PRAMs on a distributed architecture with a complete interconnection network (contention is not taken into account) are presented with delay $O(\log\log p \log^\star p)$. Note that, concerning the CRCW PRAM, this simulation is at a factor $\log^\star p$ from optimal.

In order to obtain optimal simulations, such delays are to be hidden by arithmetic computations. The key idea is parallel slackness [42, 55, 40]: it consists in simulating a PRAM with n processors on a distributed architecture with fewer processors $p < n$. The simulation is optimal (*time-processor optimal*) if the delay for an access is proportional to n/p. For instance, the previous mentioned simulation [40] leads to time-processor optimal simulation of an EREW PRAM with $n = p \log\log p \log^\star p$ processors on a distributed architecture with less than p processors. Note that parallel slackness is also involved when using asynchronous communications and threads to hide latency.

On the contrary of communications, remote access to shared memory do not basically provide a way of synchronizing the computations. In the PRAM, such a synchronization mechanism is provided by the global sequencer. On distributed architectures, intrinsically asynchronous, synchronization tools classically used are communications, locks and semaphores.

2.2 How to schedule a DFG

Being given an algorithm, the problem considered here is to schedule the DFG related to the execution on input data on a distributed architecture. The goal is to obtain an optimal schedule related to the DFG.

2.2.1 Scheduling cost of a DFG

Computing such an optimal schedule is a difficult problem. Even if communication costs are ignored and the DFG fixed (i.e. no dynamic task creation) with tasks of known duration, computing an optimal schedule is NP-complete and deciding whether the length of the optimal schedule is a given integer l is co-NP-complete [23]. However, on machines with p identical processors, there are several polynomial algorithms with bounded competitive ratio, the most famous being list-scheduling [29]. Moreover, even on non-uniform machines, approximation algorithms are known [52, 30].

Computing a schedule implies an overhead in the execution time; this scheduling overhead is governed by the time required to compute the schedule itself (i.e. the cost of the scheduling algorithm) and to realize this schedule (i.e. the mapping of tasks, preemption, migration). The scheduling overhead is included in the execution time $T_p(x)$ of the algorithm with input x on the target machine.

Definition 5 *Being given a scheduling algorithm s, the execution time of an algorithm with input x on a machine with p identical processors using the schedule delivered by s is denoted $T_p^{(s)}(x)$.*

The minimum execution time over all scheduling algorithms s *is denoted* $T_p^\star(x)$

When there is no confusion about s, $T_p^{(s)}(x)$ is denoted by $T_p(x)$.

The cost of computing a schedule is directly related to the size of the DFG, i.e. the number of tasks and dependencies it contains. Note that those costs are different from the arithmetic and communication works considered in the previous chapter which take into account the number of operations performed in each task and the number of communications related to each data dependency (transition).

Definition 6 *Let DFG(x) be the macro data-flow graph corresponding to the execution of a parallel algorithm on an unbounded number of processors. We define the following measures:*

- $N_a(x)$ *is the number of task nodes in DFG*$_a(x)$*;*
- $N_t(x)$ *is the number of transition nodes in DFG*$_t(x)$*;*
- $N_d(x)$ *is the maximal degree of a task node in* $DFG(x)$*; the degree is the number of input and output edges on a task node (to or from a transition node).*

The scheduling cost S_x *of DFG(x) is:*

$$S(x) = (N_d(x), N_a(x) + N_t(x))$$

Note that other measures may be considered in the analysis of a scheduling algorithm. For instance, other parameters considered in [7] are the maximum number of edges between any pair of nodes and the width of $DFG_a(x)$, i.e. the maximum number of tasks that may be executed concurrently.

The finer DFGx, the larger its scheduling cost and thus the more expensive will the computation of its schedule. Similarly to granularity, the regularity ρ is defined as the ratio of the arithmetic work to the size of the DFG.

Definition 7 *The regularity* $\rho(x)$ *is defined by:*

$$\rho(x) = \frac{W_a(x)}{N_a(x) + N_t(x)}.$$

A PRAM algorithm (or equivalently its related DFGs) if said of polynomial regularity *iff:*

$$\rho(x) = |x|^{1+\epsilon} \quad \textit{with} \ \ \epsilon > 0.$$

Notation. In the following, we will consider the execution of a given algorithm on a given p processors machine with an arbitrary input x of size n. Thus, all notations are implicitly related to x and n. For instance, N_a will denote $N_a(x)$, the number of task nodes in the macro data-flow graph related to the execution on an unbounded number of processors with x in input.

2.2.2 Off-line and on-line scheduling

The DFG corresponding to the execution may be partially determined at compile time by data flow analysis of the code of the algorithm, or may be discovered during the execution (depending on the value of computed data) and completely known only after the end of the execution. Depending on this knowledge of the DFG, the scheduling can be then computed off-line or on-line.

Static allocation of tasks to processors.

When the DFG corresponding to the execution can be analyzed at compile-time, it is possible to find a good schedule by hand, eventually with help of static scheduling tools. The result of the scheduling is to assign each task of the DFG to one processor (or more if replication is required). On a given processor, tasks are sequentially ordered[2] in order to respect precedences; data dependencies between them are emulated by access to shared data in the local memory. When tasks are placed on different processors, data-dependencies (i.e. access to data and precedence relations) may be emulated in two different ways:

- By communication. The data corresponding to a write-read dependency has then to be explicitly sent from the writing task to the reading one. This operation corresponds to a physical global copy of the data; locally unreferenced data have to be deleted (local garbage-collection).
  An important point is that the completion of receiving instructions implicitly implements the precedence relation (synchronization).

- By shared-memory access. Communications that implement remote access are then implicit. However, the precedence relation between non local tasks has to be described using global synchronization tools.

As a result, before execution, each processor gets its own program. Usually, this program is the same for all the processors but is parameterized by the pid of the executing processor in order to implement different behaviors. PYRROS uses this approach and a specific scheduling algorithm which performs a clustering of tasks [26, 25].

Dynamic allocation of tasks to processors.

A problem that arises frequently in computer algebra is that elementary tasks are often of unknown cost. For instance, costs of arithmetic operations (on rationals, polynomials or matrices) are usually unknown at compile time since their are related to characteristics of the values computed at execution time (size of the data, degree of a polynomial, sparsity of a matrix). Depending on such values, parallelism (i.e. creation of a task) may be generated during the execution. In such a case, an on-line scheduling algorithm is used.

Most of on-line scheduling algorithms are based on the following greedy scheme called *list-scheduling* [4, 11]:

- When a processor creates a new task (`fork` instruction of the PRAM language ATH), it stores it in a list of tasks.
  Note that, there may exist *ready* tasks, i.e. whose precedence relations are satisfied, and non-ready tasks, i.e. whose one of the precedent tasks is not completed.

- When a processor becomes *idle* (i.e it has no ready task to execute), it gets a ready task in the list.

[2] Multi-threading may be used to describe a partial execution order.

Algorithms vary depending on the way the list is managed and processors put and get tasks in it.

The program that implements the algorithm expresses a *functional parallelism*: tasks generally correspond to procedure or function calls. Non-ready tasks or data are called *future*. An important point concerns the management of data, parameters of the task: they can be systematically copied in a stack corresponding to the function call or passed by a reference to a data in the shared memory. Precedence relations between tasks may correspond either to data dependencies or to task precedences.

Scheduling operations

The previous section does not specify which instructions a scheduling can perform, except classical computations and the possibility of executing a basic task – an elementary node in the DFG – on a processor. Migration instructions allow to suspend a task during its execution in order to map it, eventually later, on another processor [52, 4]:

- *migration restricted to restart:* when a task is moved to another processor, its execution restarts from its beginning;

- *migration:* when a task is migrated to another processor, its execution restarts from its last instruction performed.

A scheduling algorithm with *no-preemption* makes no use of those operations: it has no control on a task once it has assigned it to a processor, just getting information when the task is finished. Migration restricted to restart, denoted in [52] as *no-preemption with restarts,* is useful on machines whose processors are not identical.

2.2.3 Which scheduling algorithms in computer algebra ?

An important point is that on-line and off-line scheduling algorithms have theoretical foundations [29, 22, 35, 11, 30]. There exist provably good approximation algorithms for both with bounded competitive ratio. For both, specific algorithms are developed to increase performances for certain classes of graphs (for instance trees or SP11 graphs – fork-join –).

Of course, performances of off-line algorithms are better when the DFG is known and the machine fixed. However, since on-line algorithms make no hypothesis on the execution (or few for the determination of tasks precedences), they can be used for any class of applications and thus are of general interest.

Thus, both techniques are used in computer algebra. For instance, block-scattering matrix mapping, which can be considered as an hand-made off-line algorithm, leads to near-optimal performances for linear algebra problems like dense matrix multiplication or inversion (cf chapter 1) over a small finite field (e.g. GF(2)) on a distributed architecture with identical processors.

However, due to their generality and their close relation with functional parallelism [31, 51], on-line scheduling are of specific interest for a parallel computer algebra system. In the following we thus focus of those algorithms.

2.3 On-line scheduling algorithms

2.3.1 Foundations of on-line scheduling

Theoretical foundation of on-line scheduling algorithms is due to Graham [29]. The following theorem appears has an arbitrary grain version of Brent's principle presented in chapter 1. We recall its proof which is the basis of most of further results.

Theorem 9 *[29] If scheduling overhead (i.e. the cost of computing the schedule and managing the list of tasks) and communication costs are not considered, any list-scheduling algorithm has competitive ratio* $\left(2 - \frac{1}{p}\right)$*, i.e.*

$$T_p \leq \left(2 - \frac{1}{p}\right) T_p^\star$$

.

A list-scheduling algorithm is such that, at any time, at least one processor is executing a task. Then, if at a given time a processor is idle then there exists at least one processor which executes a task. Let t_{j_1} be one of the tasks completed at date T_p and let d_{j_1} be the date when execution of t_{j_1} has been started. Two cases arise:

1. either no processor was idle before d_{j_1}.
2. either there was at least one processor idle at a certain date before d_{j_1}. Let θ be the latest date before d_{j_1} when a processor was idle. At θ, t_{j_1} was not ready (else it would have been started on an idle processor). Thus, there exists a task t_{j_2} such that t_{j_2} was being executed at θ and $t_{j_2} \prec t_{j_1}$. Let d_{j_2} be the date when execution of t_{j_2} has been started.

Recursively applying this scheme until case 1 occurs, we build a sequence of tasks $t_{j_k} \prec \ldots \prec t_{j_2} \prec t_{j_1}$ such that, at any time where a processor is idle, there exist $1 \leq i \leq k$ such that t_i is being executed on one processor.

Similarly to chapter 1, let T be the minimal arithmetic time on an unbounded number of processors and W_a be the total number of operations. The total idle time is defined by $\#I = pT_p - W_a$. For $1 \leq i \leq k$, let l_i be the duration of task t_{j_i}. We have: $\#I \leq (p-1)\sum_{i=1}^{k} l_i$ which leads to:

$$pT_p \leq W_a + (p-1)\sum_{i=1}^{k} l_i.$$

Besides, since tasks t_i, $1 \leq i \leq k$ are on a critical path: $\sum_{j=1}^{k} l_i \leq T$. This leads to:

$$T_p \leq \frac{W_a}{p} + \left(1 - \frac{1}{p}\right) T \tag{2.2}$$

We also have $T \leq T_p^\star$. Moreover, since W_a operations are to be executed in any schedule, $W_a \leq pT_p^\star$. Replacing in 2.2, we obtain: $T_p \leq \left(2 - \frac{1}{p}\right) T_p^\star$. □

As a corollary, we obtain the following constructive version of the simulation of a PRAM with an unbounded number of processors on one with p. Note that tasks is the DFG are of arbitrary durations; the only restriction which is respected in the DFG representation is that once a task is ready, it can be executed sequentially with no interruption due to synchronization. been considered in the proposed

Theorem 10 *Let $\mathcal{A}$ be an ATH PRAM program that run in (arithmetic) parallel time T and work W_a on a given PRAM with an unbounded number of processors. Then $\mathcal{A}$ can be executed by an on-line list scheduling to run in (arithmetic) parallel time T_p:*

$$Max\left\{\left\lceil\frac{W_a}{p}\right\rceil, T\right\} \leq T_p \leq \left\lceil\frac{W_a}{p} + \left(1 - \frac{1}{p}\right)T\right\rceil \tag{2.3}$$

The proof is direct from 2.2. □

Theorem 9 is stated in a restricted version [4]. In fact the bounds 2.2 holds even if the precedence relation $\prec$ considered by the list scheduling algorithm is weaker than the one $\prec'$ considered for defining the optimal schedule. The proof is direct since we will also have $t_{j_k} \prec' \ldots \prec' t_{j_2} \prec' t_{j_1}$. Clearly, the same remark holds if duration of tasks is increased.
This implies that neither adding precedence constraints such as synchronization barriers to obtain a well structured DFG nor inserting artificially null operations in order to have all tasks of the same length help any on-line algorithm.

Remark. This theorem generalizes Brent's principle (theorem 1 in chapter 1) to arbitrary DFGs, i.e. any ATH program where tasks are generated dynamically with arbitrary shared-data dependencies and are of unknown durations.

2.3.2 Lower bounds for competitive ratio

A natural question is then to determine if it is possible to have a better competitive ratio than $\left(2 - \frac{1}{p}\right)$, either on the same model or by considering larger classes of scheduling algorithms.

This problem has been studied in [52], in which the following proposition is proved.

Theorem 11 *[52] On the p-PRAM, the competitive ratio is lower bounded by $\left(2 - \frac{1}{p}\right)$ for any scheduling algorithm of the following classes:*

1. *Deterministic with no preemption,*
2. *Deterministic with migration;*

and is lower bounded by $\left(2 - \frac{1}{\sqrt{p}}\right)$ for any randomized scheduling with no preemption.

We only sketch the proof for the first case. The complete proof for this theorem is given in [52].

The adversary builds the following DFG instance G due to Graham [29]. G contains $1+p(p-1)$ independent tasks. One task α_1 is of length p, while other tasks β_k, $1 \leq k \leq p(p-1)$ are of length 1.

The optimal schedule is of length p. It executes the task α_1 on a given processor, and the $p(p-1)$ unit tasks β_k on the $p-1$ remaining processors.

The length of any schedule of G is equal to $p+t$, where t is the time when the task α_1 starts its execution. Since the tasks durations are unknown for the scheduling algorithm, the adversary strategy will thus consist in making t as large as possible.
The tasks that are processed first are then the $p(p-1)$ unit time tasks β_k, that are executed in $p-1$

time units with no idle time. Then, at time $t = p - 1$, the task α_1 starts its execution. The length of the obtained schedule is then $2p - 1$, which provides the desired lower bound. $\Box$.

As a consequence, neither preemption nor randomization can improve consequently performances compared to list-scheduling.

In order to increase the competitive ratio, it is then required to use additional informations on the DFG such as its shape or duration of its tasks.

For instance, we consider the case where all tasks are independent and sorted according to their durations; note that only the ordering is known but not durations. In this case, the on-line *LPT* list-scheduling algorithm that assigns the task of maximal duration when a processor becomes idle has competitive-ratio [29][12]:

$$\text{Min}\left\{\left(\frac{4}{3} - \frac{1}{3p}\right), \left(1 + \frac{1}{N_a}\frac{p-1}{p}\right)\right\} \tag{2.4}$$

Note that the adversary considered in the proof of theorem 11) Note that if no information is given on the durations tasks, then the fact that they are independent is of no help to decrease the competitive ratio $\left(2 - \frac{1}{p}\right)$ (cf the adversary considered in the proof of theorem 11).

Remark. List scheduling algorithms are involved as a basic level in on-line approximation algorithms used for other kind of machines such as [52, 30]:

- uniform machines: processors speeds are constant and differ each one from a constant unknown factor;

- non-uniform machines: there are no relation between processors speeds; the duration of a task varies depending on the processor which executes it.

In this case, at least migration restricted to restart is required in order to guarantee a competitive ratio [52].

2.3.3 Communications and scheduling overheads

Previous theorems do not take into account neither the cost of tasks allocation (i.e. scheduling overhead) neither communications required for access in shared memory.

Several authors have considered the theoretical influence of those overheads on list scheduling algorithms in order to provide provably optimal on-line scheduling algorithms. In [13], Cole and Vishkin give an algorithm to schedule n independent tasks optimally on a PRAM with $\frac{n}{\log n}$ processors; this algorithm is used to implement the first optimal algorithm for list-ranking [37, 39]. In [6], Blelloch, Gibbons and Matias study the scheduling of nested fine grain computations, implemented in the language NESL [5]. Blumofe and Leiserson give an optimal list-scheduling algorithm for strict multi-threaded computations[3] [8, 7], based on randomized work-stealing; this algorithm is in the kernel of the Cilk language [38]. Any of those scheduling algorithms restricts to a shape of DFG and do not take into account contention problems.

[3] There is always a dependency between a thread and one of its ancestor and access to shared data are not considered.

In this section, we give a near optimal scheduling algorithm for any DFG shape but with restrictions on the arithmetic and communication costs. We prove that, if input size is large enough, efficient and coarse-granularity PRAM algorithms[4] are executed optimally by a brute force centralized list-scheduling algorithm on a distributed architecture.

We assume that the target machine is a distributed architecture with p identical processors. In order to take into account communication costs and contention, we refer to the LogP model (cf section 2.1.1). The duration between the sending and the reception of a small message (i.e. one word) is bounded by $\sigma = 2g + 2o + L$.

Furthermore, we assume that a shared memory is simulated on the architecture with the help of hashing functions (see section 2.1.3); the delay occurring for any access in the shared memory is bounded by h. Note that d is related to the number of processors if no slackness is used.

Like in chapter 1, let C_d and W_c denote respectively the communication delay and work involved by the algorithm.

Theorem 12 *Let $\mathcal{A}$ be an ATH PRAM program that has parallel arithmetic cost (T, W_a), communication cost (C_d, W_c) and scheduling cost (N_a, N_d). Then $\mathcal{A}$ can be executed to run in parallel time T_p (including scheduling and communication overheads):*

$$T_p \leq \frac{W_a + hW_c}{p-1} + \left(1 - \frac{1}{p-1}\right)(T + C_d) + 4\sigma N_d N_a \qquad (2.5)$$

The proof is based on an adaptation of the scheme used in theorem 9.
We consider here an implementation of a list scheduling on $p-1$ processors, indexed $p_1, \ldots p_{p-1}$. The last processor, p_0, handles the list of tasks and assigns tasks to other processors.

For the sake of simplicity, we restrict the proof to the case where any shared variable is written only once and then read only once; after read access completed, the space related to the shared data is garbaged. This corresponds to the case of an EREW program with single-assignment variables.

When a processor p_i completes the execution of a task, it sends a message to p_0 and waits for receiving from p_0 a new ready task to perform.
When a processor p_i creates a new task (`fork` instruction) it sends asynchronously to p_0 a message of size bounded by N_d that define all data dependencies of the new task (i.e. the shared data that it will read before its execution or write after its completion).

Processor p_0 manages a list of ready tasks Q and a list of idle processors I. For this, it uses two arrays: the one, A, stores the task nodes created and not completed, the other, B, the descriptors of the shared variables allocated. Any descriptors in B points to the task that requires the corresponding shared data in reading. Pointers from B to A are updated at task creation and task completion. When a task in A is pointed to by no more elements in B, it is put in Q. The cost of this arithmetic computation on p_0 is proportional to N_d but independent from p and σ: we neglect it compared to $(p-1)\sigma N_a N_d$.
When p_0 receives a message of task completion, it first updates A and B, putting eventually, new ready tasks in Q. It puts the processor in I. Then, while there are ready tasks in Q and idle processors in I, it gets a task from Q and a processor from I, removes them from the lists and asynchronously sends a message assigning the task to the processor; length of the message is at

[4] i.e. with polynomial speed-up, constant inefficiency and $W_c(n) = W_a^{\epsilon}(n)$ with $\epsilon < 1$ (cf chapter 1).

most N_d. For the whole execution, the computation time on p_0 needed for the management of those lists is proportional to N_a and independent from p, N_d and σ: we also neglect it compared to $(p-1)\sigma N_a N_d$.

Note that due to contention, a processor which is idle may wait at most $(p-1)\sigma N_d$ after p_0 has assigned a new task to it and before it receives it. Conversely, when a processor completes a task, processor P_0 receives the corresponding message at most $(p-1)\sigma N_d$ tops after.

Moreover, let l_i be the length of the task t_i, $1 \leq i \leq N_a$ and let c_i be the number of unit word shared data read by t_i before and written during its execution. A processor is said idle if it i For p_0, the processor executing t_i is considered as active (i.e. *not idle*) when it is not in the list I, i.e. from the moment p_0 has send a task to it and until p_0 receives the task completion message from it: this duration is bounded by $l_i + hc_i + 2N_d(p-1)\sigma$.
Let $\#I$ be the total idle time seen from p_0 on processors $p_1, \ldots, p_{p-1}$. Let T_p be the length of the schedule; we have:

$$(p-1)T_p \leq \#I + \sum_{i=1}^{N_a} l_i + hc_i + 2N_d(p-1)\sigma \qquad (2.6)$$

We now follow the scheme of theorem 9. Let t_{j_1} be the last task completion message received by p_0 at date T_p and let d_{j_1} be the date when p_0 has assigned t_{j_1}. Two cases arise:

1. either no processor was idle for p_0 before d_{j_1}.

2. or there was at least one processor idle for p_0 at a certain date before d_{j_1}. Let θ be the latest date before d_{j_1} when a processor was idle. At θ, t_{j_1} was not ready (else it would have been started on an idle processor). Thus, there exists a task t_{j_2} such that t_{j_2} have been assigned by p_0 before θ and whose completion message has been received by p_0 after θ and such that $t_{j_2} \prec t_{j_1}$. Let d_{j_2} be the date when p_0 has assigned t_{j_2}.

Recursively applying this scheme until case 1 occurs, we built a sequence of tasks $t_{j_k} \prec \ldots \prec t_{j_2} \prec t_{j_1}$ such that, at any time where a processor is idle, there exists $1 \leq i \leq k$ such that p_0 has assigned t_{j_i} to a processor and not yet received the corresponding completion message. We thus have: $\#I \leq (p-2)\sum_{i=1}^{k}(l_{j_i} + c_{j_i} + 2N_d(p-1)\sigma)$. Besides, since tasks t_{j_i}, $1 \leq i \leq k$ are on a critical path: $\sum_{j=1}^{k} l_{j_i} \leq T$ and $\sum_{j=1}^{k} c_{j_i} \leq hC_d$ which leads to:

$$\#I \leq (p-2)(T + hC_d) \qquad (2.7)$$

where T denotes the minimal arithmetic time on an unbounded number of processors.

Let $W_a = \sum_{i=1}^{N_a} l_i$ be the arithmetic work and $W_c = \sum_{i=1}^{N_a} c_i$ be the communication work. Replacing 2.7 in 2.6 leads to:

$$(p-1)T_p \leq (p-2)(T + hC_d) + W_a + hW_c + 4N_a N_d(p-1)\sigma$$

which concludes the proof. □

As a corollary we consider a coarse-granularity efficient PRAM algorithm with polynomial regularity and bounded degree. For the corresponding DFG, this implies that for an input x of size large enough:

- polynomial speed-up: hW_c and pT (note that p is fixed): are neglected compared to W_a;

- polynomial granularity: $hC_d \leq W_c$ and thus hC_d (note that h is bounded and $C_d \leq W_c$) are also neglected;

- polynomial granularity: $4N_aN_d\sigma$ (note that σ and N_d are bounded) is also neglected.

This leads to the following result.

Theorem 13 *Let $\mathcal{A}$ be an efficient ATH PRAM program that has polynomial granularity and polynomial regularity and which have bounded degree.*
Then, for any $\epsilon > 0$, execution time of $\mathcal{A}$ on a distributed architecture with p processors is asymptotically bounded by:

$$T_p(x) < (1+\epsilon)\frac{W_p}{p-1}.$$

This time includes communication and scheduling overheads.

Note that we do not make use of slackess but instead use granularity to decrease communication overheads. An interesting question would be to use slackness in order to obtain time-processor optimal simulation, whatever the delay of d of access in shared memory is.

An improvement would be to decrease the factor $(p-1)^{-1}$ to p^{-1}: this could be possible if a distributed list-scheduling strategy was used. A classical example is *randomized work-stealing*: when a processor becomes idle, it selects uniformly at random a processor to steal a task. When a processor creates a task, it keeps it locally. Such a strategy is theoretically studied in [7]. Asymptotic bounds are given in the framework of strict multi-threaded computations. Other variants exports tasks when exceeding a certain numbers of task creations.
Such list scheduling strategies are very popular in parallel functional languages such as Multilisp [32] or Prolog [17].

In the last section, we turn to an effective implementation of the ATH language which allows the building of the DFG and thus the effective use of the above provably optimal on-line scheduling algorithm.

2.3.4 Athapascan: a simulation of the ATH PRAM language

ATHAPASCAN [43] is a parallel procedural language, inspired by Jade [50], that allows the construction of the DFG of an application during the execution. It thus makes possible the use of provably optimal on-line scheduling algorithms. We give in this section an overview of the main features of the language.

Similar to the ATH language introduce in chapter 1, ATHAPASCAN supports CUMULATIVE-CRCW PRAM algorithms. The building of the DFG is implicit; the `fork` operation (called `new_task` in Athapascan) may take in argument an optional scheduling strategy, default being a distributed list-scheduling algorithm. Taking benefit of knowledge on the graph, this allows to choose a adapted scheduling algorithm such as block-scattering for dense matrix computations or DSC for DAG with known durations [25].

2.3.5 The ATHAPASCAN programming model

The ATHAPASCAN language is a strict and para-functionnal one. It is implemented as a C++ library; it uses inheritance and templates to provide a friendly and easy-to-use interface.

In ATHAPASCAN, parallelism is expressed by asynchronous procedure calls, which correspond to the building of *tasks*. A task describes the execution of a specific procedure (which is defined by formal parameters and a block of instructions) with effective parameters. Two parameter-passing modes are possible, the by value mode copies the effective parameter into the local memory of the task and the by reference mode shares the data among different tasks.

References to shared data are typed according to their access modes. Four modes are defined to access shared data: *read* (`a1_shared_r`), *write* (`a1_shared_w`), *read/write* (`a1_shared_r_w`) and *accumulation* (`a1_shared_cw`). The three first modes are standard and are used in other parallel languages [38, 50]. Accumulation is realized from the initial value of the object by incrementation; this incrementation is defined by a binary function `f` (default is the C++ operator `+=`) which is assumed to be *associative* and *commutative*.
Thus, ATHAPASCAN allows implementation of CUMULATIVE-CRCW PRAM algorithms.

The semantics of ATHAPASCAN [5] are such that each reading of a shared datum gets the value of the last update (writing) in the sequential order of task creations (depth-first ordering). In the current implementation of ATHAPASCAN, these semantics are implemented in the following way: a task becomes *executable* when all the effective parameters that it requires in read (or read/write) mode have been updated by predecessor tasks (relative to the sequential order of task creations).

2.3.6 Execution model of ATHAPASCAN

The control of the execution is based on the building of a macro-DFG which is represented by a direct acyclic hyper-graph, which is distributed among the processors. Vertices correspond to tasks and edges to data dependencies related to shared objects: hyper-edges are used to describe concurrent writings and concurrent readings of shared objects. This graph can be labeled with information attributes (*arithmetic cost* for tasks and *data size* for shared object dependencies). This graph is used to implement both the semantics and the scheduling of tasks. Different scheduling algorithms (denoted as *scheduler*) are available and user-specific ones may be added. The role of the scheduler is restricted to informing the system where and when tasks have to be executed, taking into account information available from the graph. This functionality makes possible the implementation of different classical provably good scheduling algorithms (list scheduling, ETF [11], DSC [26], work-stealing [38] for example).

The following rules define the development of an execution:

- The first executable task is the `a1_main()` function.
- During the execution of a task:
  - when a task is created (call to the `a1_new_task` directive), the new task is inserted into the graph;

[5] ATHAPASCAN [43] allows other accesses to shared objects: postponed (suffix `p`) accesses allow the expression of a larger degree of parallelism and arrays of shared objects.

  - when a task terminates, shared data that it accessed in write or read/write mode are updated. The task is then removed from the graph and the scheduler is informed of new ready tasks (i.e. all shared objects accessed in read or read/write mode are available).

- The scheduler analyzes the graph to make task mapping and starting decisions. The system performs the scheduling decision. When all shared data required by a task in read/read-write mode have been received at the affected node, the task is started.

2.3.7 An example of Athapascan program

The figure 2.3 presents an ATHAPASCAN source code for the triangular resolution of $AX = B$; the algorithm is presented in ATH in chapter 1 (fig. 1.6).

```
struct Update : public a1_task_elem {
  Update( int size ) {
    set_cost(size*size*size);
  }
  // Performs X += -1/A*Y
  void operator() ( a1_shared_cw<matrix<float> > X,
                    a1_shared_r<matrix<float> > A,
                    a1_shared_r<matrix<float> > Y) {
    X.cumul( - A.read().inverse() * Y.read() );
  }
}

struct FinalDivision : public a1_task_elem {
  FinalDivision( int size ) {
    set_cost(size*size*size);
  }
  // Performs X = 1/A*X
  void operator() ( a1_shared_rw<matrix<float> > X,
                    a1_shared_r<matrix<float> > A) {
    X.write( A.read().inverse() * X.read() );
  }
}

struct TriangularSolve : public a1_task {
  TriangularSolve( int nb_elem ) {
    set_cost(nb_elem*nb_elem/2);
  }
  // Performs triangular resolution A*X=B
  // A is coded such that A[n*i+j] ::= A[i][j]
  void operator() (int n,
                   a1_array_of_shared_rp<matrix<float> > A,
                   a1_array_of_shared_cw<<matrix<float> > X,
                   a1_array_of_shared_rp<matrix<float> > B) {
    for(int i=0; i<n; i++) {
      X[i].cumul( B[i].read() );
      a1_new_task( FinalDivision(), X[i], A[n*i+i] );
      for(int j=i+1; j<n; j++)
        a1_new_task(Update(), X[j], A[n*i+j], B[j]);
    }
  }
}
```

Figure 2.3: Triangular resolution of $AX = B$

2.4 Conclusion

In this chapter, the on-line scheduling of a parallel PRAM program on a distributed architecture with a bounded number of processors has been analyzed. List-scheduling, frequently arising in parallel language implementations, have theoretical foundations. An optimal simulation of a PRAM program with polynomial speed-up, polynomial regularity and coarse-granularity is given; cost of communications are considered under the model LogP and a shared-memory is emulated using hash functions.

Due to its experimental good performances [57, 56], most of languages implementing dynamic parallelism use heuristics based on list-scheduling. They essentially differ on the shape of the DFG, depending on the programming model they implement. Thus, performance of list scheduling may vary depending on this model. For instance, if synchronization are authorized in the language (waiting for some future value for instance), the scheduling has to use migration; if not, no guarantee can be given on the competitive ratio.

We focus in this conclusion on languages that use a provably efficient on-line scheduling algorithm. HPF 2 introduced groups of independent tasks of unknown durations via function calls. A BSP [54] program execution consists in a sequence of super-steps, each setted of independent tasks. All shared memory access performed at a step are effective at the next one. Dynamic load balancing is possible [54] but requires task migration in the considered implementation [28].

Functional languages use list-scheduling since a long time. For a survey on parallelism in functional languages, see [33], we just mention here some characteristic languages. Sisal[44] is a data-flow based language which defines a fine grain DFG; however, programming macro-tasks in order to obtain a coarse-granularity algorithm is not directly possible. NESL [5] provides a nested parallel model: graphs corresponds to recursive n-ary set of independent tasks with no data-dependencies but synchronization at the join point. Access are emulated on a virtual shared memory. Cilk [7, 38] is inspired from Multilisp and implements on the C language a model of strict functional computation. Tasks are mapped on functions; all data are accessed in the stack. Function can be migrated at a synchronization point, explicitly defined in the program. Migrations is reduced to a copy of the stack. ATHAPASCAN [19, 43] is inspired from Jade [50]; it implements in a C++ library a programming model similar to the language ATH presented in the previous chapter. Data-dependencies are defined by access to a shared data. Tasks corresponds to procedure calls; parameters can be passed by value or by reference to a shared-data. This last mode defines the precedence. When a task is ready, it can be executed till completion with no synchronization.

In computer algebra, list scheduling occurs frequently. The next chapter is devoted to a description of the different approaches considered in parallel computer algebra.,

Bibliography

[1] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of PRAM's. *Theoretical Computer Science*, 71:3–28, 1990.

[2] A. Alexander, M. Ionescu, K. Schauser, and C. Scheiman. Incorporating long messages into the LogP model. In *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures*. ACM Press, 1995.

[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 1993.

[4] J. Blazewicz, K. Exker, G. Schmidt, and J. Węglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, Germany, 1993.

[5] G. E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[6] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa-Barbara, California, 1995. ACM Press.

[7] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Massachussets Institute of Technology, Boston, 1995.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Symposium on Foundations of Computer Science*, pages 356–368, Santa-Fe, New Mexico, 1994.

[9] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime: Efficiency for irregular problems. In *Proceedings of EuroPar'97*. Springer-Verlag, Aug. 1997.

[10] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional Computing Series, 1997.

[11] P. Chretienne, E. J. Coffman, J. K. Lenstra, and Z. Liu. *Scheduling Theory and its Applications*. John Wiley and Sons, England, 1995.

[12] E. Coffman and S. R. A Generalized Bound on LPT Sequencing. *RAIRO Informatique*, 10:17–25, 1976.

[13] R. Cole and U. Vishkin. Approximate Parallel Scheduling. Part I : The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time. *SIAM Journal on Computing*, 17(1), 1988.

[14] J. Colin and P. Chretienne. C.P.M. Scheduling with small communication delays and task dupli cation. *Operations Research*, 39:680–684, 1991.

[15] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schausser, and T. v. E. Ramesh Subramonian. LogP: A Practical Model of Parallel Computation. *Communications ACM*, 39(11):78–85, 1996.

[16] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schausser, and T. v. E. Ramesh Subramonian. LogP: Towards a realistic model of parallel computation. In *proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[17] J. C. de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, september 1994.

[18] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A Message Passing Standard for MPP and Workstations. *Communications ACM*, 39(7):84–90, 1996.

[19] B. Dumitrescu, M. Doreille, J.-L. Roch, and D. Trystram. Influence of scheduling on actual high-performance computing applications: sparse cholesky factorization as a case study. In *Proceedings of PPAM'97 – 2nd International Conference on Parallel Processing and Applied Mathematics*, Zakopane, Poland, 1997.

[20] A. Dusseau, D. Culler, K. E. Schauser, and R. Mart in. Fast parallel sorting under LogP: experiences with CM- 5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8), 1996.

[21] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *IEEEE Journal of Parallel and Distributed Computing*, 1997.

[22] M. Garey, R. Graham, and D. Johnson. Performance guarantees for scheduling algorithms. *Operation Research*, 26(1):3–21, Jan. 1978.

[23] M. Garey and D. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[24] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, W. Manchek, and V. Sunderam. *PVM: A Users'Guide and Tutorial for Networking Parallel Computing*. MIT Press, Cambridge, Mass., 1994. Available electronically; see ftp://www.netlib.org/pvm3/book/pvm-book.ps.

[25] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling DAG's on multiprocessors. *Journal of Parallel and Distributed Computing*, Dec. 1992.

[26] A. Gerasoulis and T. Yang. PYRROS : Static Task Scheduling and Code Generation for Message-Passing Architectures. Technical report, Rutgers University, USA, 1993.

[27] P. Gibbons. A more practical PRAM model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[28] M. Goudreau, J. Hill, K. Lang, and B. McColl. A Proposal for the BSP Worldwide Standard Library (*preliminary version*). Technical report, http://www.bsp-worldwide.org/, Oxford University, GB, 1997.

[29] R. Graham. Bounds for Certain Multiprocessor Anomalies. *Bell System Tech J.*, 45:1563–1581, 1966.

[30] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to Minimize Average Completion Time: Off-line and On-line Approximation Algorithms. Technical Report 516/1996, Technishe Universität, Berlin, 1996.

[31] R. Halstead. Parallel symbolic computing. *IEEE Computer*, 19 (8):35–43, 1986.

[32] R. Halstead. Parallel computing using multilisp. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 21–49. Kluwer Academic Publishers, 1988.

[33] K. Hammond. Parallel functionnal programming: an introduction. In H. Hong, editor, *First International Symposium on Parallel Symbolic Computation (PASCÓ94)*, Lecture Notes Series in Computing, pages 181–194, 1994.

[34] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation . *Journal on Parallel and Distributed Computing*, 16(3), 1992.

[35] D. S. Hochbaum and Shmoys. Using dual approxiamtion algorithms for scheduling problems: Theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[36] J.-J. Hwang, Y.-C. Chow, F. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[37] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, Massachussets, 1992.

[38] C. Joerg. *The Cilk system for parallel multithreaded computing*. PhD thesis, Massachussets Institute of Technology, january 1996.

[39] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 869–932. Elsevier, 1990.

[40] R. M. Karp, M. Luby, and F. M. auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. *Algorithmica*, 16:517–542, 1996.

[41] R. M. Karp, A. Sahay, E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proceedings of the 5th Symposium on Parallel Algorithms and Architectures*. ACM Press, 1993.

[42] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[43] J.-L. R. Mathias Doreille, François Galilée. Athapascan-1b: Présentation . Technical Report http://navajo.imag.fr/ath1/, Projet APACHE, Grenoble, France, 1996.

[44] J. McGraw. SISAL: Streams and Iterations in a Sigle-Assignment Language – Reference Manual. Technical Report Manual M-146, Lawrence Livermore National Lab., 1985.

[45] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21, 1994.

[46] R. Namyst and J.-F. Méhaut. Pm^2 parallel multithreaded machine: a multithreaded environment on top of pvm. In *Proceedings of EuroPVM'95*, pages 179–184. HERMES (ISBN 2-86601-497-9), 1995.

[47] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[48] A. G. Ranade. How to emulate shared memory. In *Proceedings 28th Annual Symposium on Foundations of Computer Science*, pages 185–192. IEEE, 1987.

[49] V. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[50] M. Rinard. *The design, implementation and evaluation of Jade : a portable, implicitly parallel programming language*. PhD thesis, Stanford University, september 1994.

[51] W. Schreiner. A Para-Functional Programming Interface for a Parallel Computer Algebra Package. *Journal of Symbolic Computation*, 21:593–614, 1996.

[52] D. B. Shmoys, J. Wein, and P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24(6):1313–1331, 1995.

[53] M. Snir, S. W. Otto, S. Hess-Lederman, D. Walker, and J. J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, Mass., 1996. Available electronically; see http://www.netlib.org/utk/papers/mpi-book.html.

[54] L. G. Valiant. A Bridging Model For Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[55] L. G. Valiant. General purpose parallel architectures. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 944–971. Elsevier, 1990.

[56] A. S. Wagner and S. T. Chanson. Performance Models for the Processor Farm Paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):475–489, 1997.

[57] M. Willebeek-Le-Mair and P. Reeves. Strategies for dynamic load-balancing on higly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.

Chapter 3

Parallel computer algebra systems.

3.1 Introduction

In the following we overview and try to compare some existing softwares for parallel computer algebra computations. Our aim is to point out the main research directions that could be applied in general purpose situations.[1]

We focus more on *parallel systems*, *parallel environments*, *parallel libraries* or *parallel extensions of the existing* to execute algebraic algorithms than on closely related topics such as *parallel deduction*, *parallel logic* or *functional parallel programming*. In the same way, we will not refer to the numerous implementations of specific algorithms that can be found in the literature. Well known topics in computer algebra have been addressed *e.g.* linear algebra, polynomial computations, Gröbner bases computations ... But, even if important from an algorithmic point of view, these particular implementations usually do not propose any new concept from the system point of view.

Since the 80's and the first conferences on the subject [13, 14] many experiments have been done in parallel computer algebra. Two concurrent evolutions, the progress of the parallel computers and networks, and the coming out of some common paradigms – we specially think to the use of *threads* – have given rise to numerous reports on such experiments. We will list and briefly describe the main ones[2] from §3.4 to §3.14[3]. It may not be vain to note that even if each environment is usually tested and validated by its authors on few specific applications, most of them can be used in much general contexts.

3.1.1 Simultaneous computations in computer algebra

In parallelizing computer algebra, one is faced to three main different tasks:

- Computer algebra aspects: one has to continue the efforts in sequential performance.

- Expressing parallelism: programming details, especially in parallel, can be very time-consuming. One has to free the user from too technical details by providing a simple way to express parallelism.

- Issues relating to parallel implementation: one has to study problems such as load-balancing, scheduling, granularity and locality.

The first task concerns computer algebra and the two other ones concern parallelism, the main problem is thus to determine how the two aspects have to interact. Depending on the relative importance of these aspects for the end-user, three different approaches may be distinguished to conceive a parallel computer algebra system or to provide parallel resources to the user:

A. Parallel aspects are transparent to the end-user who use a sequential system as usually. This can be accomplished by replacing calls to sequential routines to calls to parallel ones using a parallel library. Sequential operators may be overloaded.
Advantages: the user is free from any parallel aspect. Codes do not need any modification.

[1] An overview by W. Küchlin appeared in [65] and provides many relevant complementary remarks and references.
[2] The author will appreciate any remark about forgettings to improve subsequent versions of this document.
[3] The ordering is not relevant.

Drawbacks: the user relies on the available parallel library and cannot augment it.
This strategy can be viable and very interesting when key routines of the computer algebra kernel are parallelized. This is one of the target goals with the vectorization of the basic arithmetic in [71, 72] (*cf* §3.7), or with its parallelization [17] (*cf* §3.13.6).

We may note that the same result could be obtained using automatic parallelization of sequential codes, and generate parallelism at compilation. Such studies remains theoretical [37] (*cf* §3.13.4), especially in computer algebra, we are not aware of practical aspects of this approach.

B. Using a sequential computer algebra system or library, the language is augmented to provide means to express parallelism. The sequential kernel is linked to or rely on a library that handle the parallel tasks and the related data communications. This is currently the most common strategy. This uses either standard or *ad-hoc* computer algebra library (From MAPLE, REDUCE ... to C++ libraries from the scratch). Either *message passing* or a *fork / join model* is used to express parallelism.
Advantages: efficiency should be obtained for any parallel algorithms if the parallel software is efficient. Any level of task granularity is thinkable.
Drawbacks: the user is not free from parallel aspects.

As said above most of the references we have got enter this class. They will be different in the parallel programming model used and in the way the parallel tasks are mapped onto the physical processors. We refer to §3.1.2 and to subsequent sections for more technical aspects and descriptions.

C. In a distributed environment, several simultaneous sessions of a computer algebra system may be launched and considered as servers of a unique application.
Advantages: parallel algebraic computations are made possible in an heterogeneous environment (heterogeneous computational units).
Drawbacks: parallelism is generally used at a coarse-grain level of granularity.

This approach was somehow the one followed in [18] (*cf* §3.4), even if DSC mainly belong to the previous class of systems. The same remark is valid for the software in [31] (*cf* §3.14.3). For distributed computer algebra the reader may refer to [28, 29] and references therein. With the evolution of the machines and of the communication networks, the boundary between distributed and parallel computations may be very indefinite. Furthermore, the two fields have strongly related concerns. One of these is to develop means of communicating mathematical data. We refer to §3.15 for a brief discussion on this subject.

3.1.2 Parallel computer algebra

The parallel library in the first approach above relies either on the second approach or on the third one. The distributed approach is not fully usable when communication is really intensive or when efficiency is the main concern. Up to now, for these two main reasons, we will focus on the second one, we mean on softwares to implement medium/fine grain parallel computer algebra algorithms.

Again, three main directions may be followed [4, 43]:

B1. A *standard computer algebra system* is used as sequential kernel. Several such kernels are connected to exchange data.
Advantages: easy to use for someone who yet knows the system. Quite easy to build. Exploit proven codes.
Drawbacks: often implies a *message passing* programming model or coarse grained applications. Indeed, the time required to transfer data between processes determines the problem grain sizes that can be used. The costs of data format conversions in this approach may be prohibitive.

Among the references in the text we may refer for instance to [20, 95, 19, 18] for experiments with MAPLE or to [76, 77] with REDUCE.

B2. A *sequential computer algebra library* whose source is available is augmented for the expression of parallelism and/or for data communications.
Advantages: can exploit any grain of granularity and any type of memory. Can use proven code as a basis.
Drawbacks: the parallel model or language must be an extension of the sequential one.

Typically this is the chosen strategy in [111, 65, 92] where SAC-2 is extended, in [55, 7, 107] where SACLIB is used or in [9] with ALDOR[4].

The main difference with the previous approach is usually the communication cost and consequently, as remarked above, the target granularity.

B3. A *new parallel computer algebra system* is built using existing or new parallel facilities (language,library . . .).
Advantages: this is potentially more portable and efficient.
Drawbacks: requires more implementation efforts.

Such a point of view is followed in [44, 46, 43] where a parallel software and a computer algebra library are simultaneously designed.

The difference with the two previous approaches is thus in the fact that the conception of the parallel programming model can done with more care.

At present, these three approaches may sometimes[5] be viewed as favoring computer algebra (B1) or favoring parallel aspects (B3), (B2) being a compromise; as favoring studies on algorithms (B1) (*e.g.* fast experimental studies with huge data and problem dimensions) or favoring studies on parallel concepts (*e.g.* which criteria are relevant for dynamic load-balancing at a fine-grain level of granularity).

The paper is organized as follows. In §3.2 we briefly have a look at the main evolutions in the field. Since the first practical experimentations on networks of workstations or on parallel machines around 1984, the main evolutions has obviously concerned the machines and as a consequence, the programming model. This study is based on the aspects that are specifically relevant in computer algebra in §3.2.1; on a fast description of three main types of programming models in §3.2.2;

[4] In fact, in most of these cases, since the sequential language is not too restricting, the difference between this approach and the next one may be difficult to make.

[5] Even if the references cited usually address both computer algebra and parallel aspects.

on some remarks about tasks scheduling and dynamic load-balancing in §3.2.3. Some possible criteria for a comparison will be pointed out in §3.3 before the overview itself of existing softwares from §3.4 to §3.14. Before concluding we make some remarks about links with the developments of protocols for the communication of mathematical data.

3.2 Parallel computer algebra: what changed since the 80's?

Ten years ago a change of parallel architecture and operating environment was requiring many work – from the conception of the parallel algorithm to its implementation – to be redone. What about the situation today? Since the – very early – experimental studies of [108], what can we conclude about quite many existing works and experiments with parallel computer algebra?

Beyond the underlying evolution of the machines, two main facts are remarkable during this period[6]:

***De facto* message-passing standard** libraries for connecting parallel tasks on either homogeneous or heterogeneous networks of computers have appeared. These are PVM [47] and MPI [40] (*cf* §3.14). The associated programming model is based on message passing.

Operating systems and threads are now widely available on most of the parallel machines. This provides an alternative programming model also very commonly used.

Thus in 12 years, machines became easier to program and codes became portable. Note that from this standardization of the programming models[7], the differences between the hardwares (shared / distributed memories, workstation networks) has became blurred from the end-user point of view.

Most of the old problems (84 – 94) was to design environments on particular machines and to show – especially for computer algebra – that it was possible to obtain optimal speed-ups for particular algorithms on specific architectures. Open questions now mainly concern the way to *easily* get a *satisfying efficiency* in general situations, on possibly many processors (*scalability*) and for various architectures (*portability*).

In the following me make some basic remarks to see how such questions are commonly addressed in computer algebra.

There are two main differences between sequential and parallel computers: in the latter there are potentially large overheads in communication (or access to a shared memory) and in task management. *Related questions are thus to provide means to express communications and task management, and then to provide means to route the data and map/schedule the tasks.*

3.2.1 Specifically in computer algebra?

Clearly, computer algebra data and data structures are specifically of various types. This implies sophisticated means to exchange them between computational units. But we put off the discussion

[6]See the conclusion concerning current trends

[7]Other standards have emerged during the same period, especially *High Performance Fortran* for data parallel computations [39]. This model seems to be too restrictive for general algebraic computations and have not be used for this purpose to our knowledge. See also the *Bulk Synchronous Parallel* model in §3.2.2.

of this subject till §3.15 and will mainly focus on task expression and management.

Another distinctive feature of computer algebra algorithms is that they can be *highly dynamic* (precise behavior known at execution). From a parallel point of view this can be understood using the concepts of *locality* and of *irregularity*. Very intuitively, an algorithm is local if the computational complexity dominates the communication complexity [79]. An algorithm is irregular if it is costly to map/schedule the tasks it generates [46].

The dynamic behavior of the algorithms implies[8] dynamic tools to obtain satisfying performances at execution in a portable manner. Mapping and scheduling are thus major issues in the field.

We will see that some dynamic load-balancers are now designed and used in computer algebra. Again, due to the evolution of the parallel machine capabilities (operating systems), such tools should now be widely recognized, as they are in other topics that share the same characteristics [46, 34, 35].

3.2.2 Parallel programming models

One main question in parallelism is to provide a programming model that could be easily translated into the machine language and efficiently executed on a wide class of machines [101][9].

Three main parallel models are used[10].

Data parallelism is data driven and generated by data splittings. A typical language for this model is HPF [39]. A parallel program differ from a sequential one mainly in the parallel iterators (*e.g.* parallel loops). Synchronizations, communications and tasks are generated at compilation. With a unique flow of control, the model is particularly suited to regular data structures.

Parallel iterators can be found in computer algebra. We refer for instance to `parmap` in [19], to the parallel evaluation in [95] or to parallel loops in [51]. Anyway, in these cases, these parallel constructs are build upon execution layers of the two other types below.

Message passing was the only model available on the first distributed memory machines. This partly explains the nowadays infatuation for the libraries PVM [47] and MPI [40, 36], or at a lower level for the BSP model [101] and its implementations [52].

This model, introduced in [53], is based on the notion of *communicating processes*. A process is the execution of a program in different *states* related to *actions*. With parallel computations actions correspond to `create` and `destroy` for processes, to `send` and `receive` for messages.

Processes are *heavy processes* (*e.g.* Unix ones) in PVM and MPI[11], communications are either from a process to another (point to point) or from a group of processes to another group (global synchronizations and communications). A virtual global memory and a programming style – based on super-steps – that depends on few machine parameters (flops,

[8] At present.
[9] This is so different from the sequential context where languages such C, LISP or FORTRAN exist.
[10] Or at least vaguely considered concerning *Data Parallelism* in computer algebra.
[11] A lot of studies aim at including threads in these standards.

bandwidth and synchronization cost) seems to provide a more portable (with respect to the efficiency) model BSP [101, 52].

Message passing has been often used in the past and is still used essentially using PVM and MPI, see [76] or [107] among other references.

Functional programming with fork / join provides a general MIMD model. A function (or a procedure) is forked *i.e.* a new task is created by an instruction `fork` (synchronous or asynchronous). Results are obtained when the task resume by joining using an instruction `join` together with the calling function. Practically this can be implemented using threads (shared memory) or distributed threads[12] (distributed memory). Threads create other threads, this can be done recursively with no theoretical limit to the number of threads. The model can also be implemented as in LINDA [48, 2] where the evaluation of a virtually shared name space, permits to spawn procedures. There is no connection between this number and the number of physical processors. When launched, the tasks can exchange data using co-routines [26] (*cf* §3.4); or using *multi-functions* [5] (*cf* §3.11); or accessing to a shared memory either physically (*cf* §3.5 and §3.8)) or virtually (*cf* §3.10 and §3.12).

This model is often associated to a *divide and conquer* programming style. It must be built on a software layer or using an operating system, that will map and schedule the executions of the tasks as we will see in next section.

Early work in the functional programming field and in computer algebra use *data-flow analysis* [70, 38] for LISP or for REDUCE [37] (*cf* §3.13.4). Parallelism is achieved by evaluating the arguments of a function in parallel. The parallelism is thus investigated at the level of a function investigation, this is very similar to the approach based on *fork / join*. One main difference is in the way the data-flow graph is computed: either statically *e.g.* at compilation, or dynamically *e.g.* during execution[13].

Anyway, the differences in the way to express parallelism and in the work that has to be done at compilation in the latter approach, show that many practical studies remain to be done to make a good choice. One important work in functional programming has also led to the concept of *future* [59, 49, 50]. Instructions `fork` and threads are often similar to *futures*.

Data parallelism language constructions can be very useful for regular computations in computer algebra (*e.g.* for replicated computations, for a `map` operation on an array or a parallel loop). However this model seems to be too restrictive for *irregular* algorithms.

Message-passing models are quite often preferred nowadays since *standards* softwares are available on a very wide class of machines (machines dedicated to parallelism, homogeneous and heterogeneous networks). Unfortunately the model, often available only with coarse-grained granularity, can be difficult to use when dynamic load-balancing, at a medium/fine grained granularity, is an important issue.

We will definitely prefer the *fork / join* model at least in this latter case.

[12] By abuse of language, we will frequently omit “distributed” in “distributed threads”.
[13] A mixed approach is thinkable. See for instance the dynamical data-flow analysis in §3.12.

3.2.3 Dynamic load-balancing

In [19] (for instance) we read: "*Optimal scheduling of a set of tasks with a predetermined number of homogeneous, unloaded processors is one thing; getting plausible performance during a computation chock full of possibility of heterogeneous concurrent tasks on system with other users is another*". Such a comment seems to be obvious to an aware reader and could be found in many other papers in the field of parallelism. We repeat it for two main reasons.

On the one hand, it appears that the remark is too often forgotten. For example, one may be easily tempted to work with few processors on a fixed problem, then to extend the particular conclusion to a more general context. Fortunately, it seems that this is less common than before in the literature. On the other hand many questions related to this remark remains open.

If parallel tasks defined at an abstract level by the programmer was mapped "by-hand" when using the first distributed memory machines, this is no more true in most cases. Since the functional programming model is well established in computer algebra, this model with *fork / join* instructions is always used for dynamic load-balancing in parallel. When a task is created, it is generally placed in a queue and wait to be mapped and scheduled on a target computational unit. Further, by analyzing the task queue and in particular the dependence of the input/output one may deduce a data-flow graph. Specific techniques may then be used to map and schedules the tasks from the global information available from the graph. We refer to [16] (*cf* §3.12).

We are not aware[14] [15] of other types of approaches (with practical experiments) in computer algebra.

Once the tasks are created many strategies can be used to map and schedule them, and various informations – including ones provided by the users – can be useful to optimize this process. The main problem is to permit implementations that are *execution-portable* (efficiency on various machines) and *scalable* [98] (remain efficient when the number of processors is increased). We may identify five main issues to attain this goal:

Mapping / Scheduling. By abuse we will call *load-balancer*, a tool which maps the tasks onto computational units, give them a date to execute *i.e.* schedules them and possibly make them migrate, sleep or stop. A load-balancer is generally formed of two parts: the *information* one to spy the current load of the machine and the *control* part to actually do the mapping and the scheduling (taking into account the informations about the load). (Active/passive, centralized/distributed, …). We do not detail anymore this subject, the reader will refer to [15] for an overview. We will just precise, during each description of existing softwares, how the two parts have been conceived and how do they work.

Granularity. Strongly related to the cost of communications, this is a main issue. We have seen that the programmer specifies parallelism at an abstract level of granularity. To prevent a to large number of simultaneous tasks and data transmitted through the network (that could overload the system) or too small tasks, it is necessary to limit this number either *a priori* or *a posteriori*. Using a *divide and conquer* style, it is possible to *a priori* limit the number, by choosing to run the task on the same processor than their calling tasks as soon as enough

[14] Static approaches are very common in parallelism but seems to be too restrictive for computer algebra purposes.

[15] It is important to note that, as seen in previous section, the data-flow analysis proposed in [38, 37] is a main possible alternative to the model based on *fork / join*. Good performances could certainly be obtained this way.

parallel tasks have been created [74, 19, 85]. *A posteriori*, as proposed in [65], it is possible to coalesce virtual tasks into bigger real ones according to the informations about the load.

Locality. Can be crucial when the state used by a task is large and cannot be replicated. Static or dynamic strategies should be used to place tasks on units that own most of the relevant data or to limit data replications. (*cf* §3.12).

Cost informations. In addition to the informations about the machine load, cost informations about the tasks may be provided by the user. Such informations may be relevant for the efficiency of the load-balancer [86, 46] (*cf* §3.4,3.6,3.12).

Poly-algorithms. For several algorithms solving a given problem, to know the best one for a given machine, will depend on machine parameters (*e.g.* the number of processors) and on problem parameters (*e.g.* the dimension). The use of *poly-algorithms*, we mean algorithm that permit to differ the choice of the appropriate method (using machine parameters) till the compilation or even the execution, is a natural way to solve the problem [98, 86] (*cf* §3.6).

We see that many relevant have to be taken into account to build an efficient tool to get good performances. At present, there are quite few experiences in this field in computer algebra. A lot of work remains to be done to test, tune and validate the strategies in meaningful situations.

3.3 Which criteria for a classification?

We summarize the main aspects presented in previous sections and a little bit more detailed in subsequent ones. From §3.2.1 we know that dynamic load-balancing is at present the first issue of domain.

In table 3.3 we consider the main[16] softwares available : DSC (§3.4), PARSAC-2 (§3.5), PAR. POLY. OP. (§3.6), SUGARBUSH (§3.8), STURM (§3.10), GIVARO (§3.11), GIVARO with ATHAPASCAN-1 (§3.12), ||MAPLE|| (§3.13.2) and MUPAD (§3.13.3).

The first rows describe the GLOBAL DATA MODEL: the way the memory of a processor can be accessed by another processor.

Then the TASK MODEL rows precise if *heavy processes* (*e.g.* Unix ones), *Remote Function call* non multi-threaded, or multi-threads is used. The *synchronization* explains how these process are explicitly synchronized.

The ARCHITECTURE rows indicate if the software is designed for distributed or shared memory machines[17], and shows if the system can run on heterogeneous architectures.

The LOAD-BALANCING rows show if a dedicated tool has been conceived to handle the tasks and give some precisions on the used strategies.

3.4 DSC

DSC is a general purpose tools, applied in particular to algebraic computing, that manages tasks distributed over a network of workstations (Unix) [32, 30] for large computations. DSC has first

[16] And more recent.

[17] It is much easier to port a software from a distributed memory to a shared one, than to do the converse.

Table 3.1: Classification of softwares for parallel computer algebra.

	DSC	PS-2	PPO	SBSH	STU	GIV-1	GIV-2	||M||	μP
GLOBAL DATA MODEL									
- message passing		•	•		•				•
- async. comm.						•			
- virtually shared				•			•		(•)
TASK MODEL									
- heavy processes	•		•						
- RFC				•				•	
- multi-threading		•			•	•	•		•
- preemptive		•			•				•
- async. creation	•	•			•		•		•
-SYNCHRONIZATION									
join variable		(•)			•		•		
coroutine	•								
multi-procedure						•			
ARCHITECTURE									
- dist. memory	•	•	•	•	•	•	•	•	(•)
- shared. memory		•	•	•	•			•	•
- heterogeneous	•		•	•			•		
LOAD-BALANCING									
- distributed						•	•		
- passive strategy					•	•			
- load indicator	•	•		•			•		
- user info.	•	•	•	(•)		•	•		

been used by programming in C or LISP, an interface to MAPLE [22] has then been proposed in [18].

The model of computation is *client /server* and hides both the interprocess communications and processor allocation to the user. The communications are based on standard Unix capabilities (TCP / IP and UDP). The library essentially permits to submit a task to the environment and to wait for its completion. The subtasks themselves may recursively spawn further subtasks. An additional mechanism – say *co-routines* – whereby a subtask remains loaded in memory space on a return, until it is subsequently awaken, appears to be very useful in DSC for real applications [30].

Table 3.2: Classification – DSC.

Languages	Fork/Join	Memory	Load-balancing	Grain
C / LISP / TCP-UDP	Unix processes	Distributed	Yes / Task queue	Coarse / Medium

The system has been primarily designed for large computer algebra applications on heterogeneous networks of workstations. A sophisticated scheduler is proposed [30]. On the one hand, the scheduler receives the cpu and memory loads of the available machines. On the other hand the user specifies a rough amount of needed usage. Then, on a request, the system makes the selection of which processor is to handle the work, or decides to queue the request for later distribution. Such a finely tuned feature seems to have been necessary to run the test on inputs as large as the ones reported.

The design has been extensively tested on symbolic applications such as sparse linear system solution [30, 67], primality testing and factorization [100]. The scheduler has been tested with a common network of up to 30 machines.

3.5 PARSAC-2

PARSAC-2 [62] is a programming environment belonging to the class of systems based on threads of control. The support for *fork / join* parallelism initially for shared memory machines in [63, 64] (S-THREADS) has been extended to distributed ones in [10] (DTS). It relies on SAC-2 [25] translated to C from ALDES [68] for its sequential computer algebra library. For a comprehensive introduction we refer to [65].

The programming model is based on *fork / join* functionalities, the suggested way to program is to use a *divide and conquer* style.

The system relies on S-THREADS for parallel programming on each node of the target machine. For portability, the design supports a *virtual parallel programming model*: an algorithm is parallelized with respect to its logical parallel structure. The corresponding logical threads of control are mapped at run-time. Another layer, DTS, is built upon PVM to handle fork / join across a network.

For efficiency, a dynamic load-balancer is available. The current approach is centralized but could be distributed. The chosen strategy runs `Maxjobs` on each processing unit and keeps the others in a global queue. This maximum number depends on a constant (currently an heuristic) and on the number of pending jobs with respect to the number of processors. This is designed in order to avoid both the risk of unbalanced load and of overloading of the units.

Table 3.3: Classification – PARSAC-2.

Languages	Fork/Join	Memory	Load-balancing	Grain
SACLIB / C/ PVM	Threads	Shared / Distributed	Yes / Task queue	Medium / Fine

For good speed-ups on a network of workstations, the reader may refer to [10] (polynomial resultants). Further parallel concepts (search parallelism and thread groups) are used in [3] for Gröbner bases computations. Load-balancing is completed by some user-specified key parameters (*e.g.* number of concurrent reductions of polynomials). This clearly shows the need for further studies about relevant parameters of automatic load balancers in computer algebra.

3.6 Parallel Polynomial Operations, MP / MPP

Research at Kent focuses since several years on parallel symbolic computations on polynomials. The reader will refer to [103, 104, 105] for reports on experiments and timing date and to [106] for a recent survey of the corresponding parallel algorithms. The experiences has been conducted mainly on symmetric multi-processors and now turn towards general environments [107] as we will see in §3.14.

The work is done using C language or a ported SACLIB package [11] together with a module for data communication between tasks. Several libraries have been implemented, especially for factorization of polynomials. This does not really form a parallel computer algebra system but provides key ideas for a future realization of such a system. Parallel algorithms are implemented by forking subprograms and waiting for the results.

Table 3.4: Classification – Parallel Polynomial Operations.

Languages	Fork/Join	Memory	Load-balancing	Grain
C, SACLIB	System processes	Shared	By hand	Medium / Fine

An interesting point concerns dynamic load-balancing. Depending on the number of available processors and on the size of the problem to solve, "by hand" dynamic load-balancing is ensured by the programs. We mean that a poly-algorithm is provided and that the subprogram which is actually executed is chosen during the execution. This is a rough mechanism but implements an interesting concept.

3.7 Parallel REDUCE

Vectorization has been proven to be efficient for big numbers and polynomial arithmetic in [71, 72, 73] (this include a vectorized garbage collection). These articles reports intensive experimentations on CRAY machines with coarse-grain parallelism especially for Gröbner bases computations. Fine grain parallelism was judged to be useless on many processors. The main reason being that the heuristics used in sequential seem to be somehow inconsistent with many concurrent tasks [72, 78][18].

Following the evolution of both the parallel machines and of the parallel tools, a parallel version of REDUCE with PVM is under development [76, 77]. The basic protocol uses send/receive for LISP forms, the basic model of parallelism is master/slave. Applications are coarse-grain computer algebra ones, algorithms in linear algebra are currently under investigation.

[18] Since then, many papers have been published on parallel Gröbner basis computation. The reader will refer to them for a discussion about the ability to take advantage of many parallel tasks or not.

3.8 MAPLE/LINDA

A general-purpose parallel algebraic system is proposed in [20] as an association of MAPLE and LINDA [48, 2]. A main feature of LINDA is that system and architecture-dependent details for parallelism are hidden from the final programmer. LINDA provides a global name space where all the processes can read and write. The user programs as if using a shared memory machine. The C/LINDA used here extends the C with parallel procedures working on *tuples* (ordered sequences of objects or names). A *global* collection of data is maintained by the system: it is viewed as shared among all processes. The *eval* procedure on tuples spawns separate process for each element of the tuple. Each process evaluates its element which can be a procedure call. LINDA tuple space leads naturally to the notion of tasks into a homogeneous pool (or queue, stack, ...). MAPLE/LINDA appears to be a batch system with several copies of MAPLE running simultaneously and communicating by LINDA operations. A *master*/*slave* approach is tested in [20] (parallel iterations, sparse modular gcd) and speed-ups are given with three independent MAPLE processes (shared memory Sequent Balance).

This parallel MAPLE has also been implemented for networks of workstations [21] under the name SUGARBUSH. In [19] (for instance) empirical data are reported with up to 27 processors for parallel integer multiplications. Empirical data are obtained using load-balancing heuristics (see also [66]). Even if user-specified, ways are provided by the system to tune the granularity by cutoff. These experiments show the consequences of a lack of dynamic scheduling when using many processors. The authors also ask some important questions about the inclusion of dynamic scheduling heuristics in parallel computer algebra systems. For instance, one problem is clearly to tune thresholds to decide when a given task should be splitted (for subsequent concurrent tasks) or not?

Table 3.5: Classification – MAPLE/LINDA.

Languages	Fork/Join	Memory	Load-balancing	Grain
C / MAPLE	System processes	Shared / Distributed	By hand / Task queue	Medium

3.9 PACLIB with PD

The PACLIB kernel [55] is the parallel variant of the runtime system for SACLIB. The computer algebra facilities are thus the ones of this latter library [11]. For the parallel aspects, PACLIB relies on μSYSTEM task package (C functions for concurrency) [12]. A more sophisticated version of PACLIB is using the high-level programming interface PD proposed in [89, 91]. The reader will refer to [57] and to [91] for detailed presentations of the corresponding programming models.

Using PD, at the lowest level, the programming model is based on the *fork /join* operations of tasks, or *streams* (semantically lists except for the type interface). At this level, the programmer has to care for the insertion of synchronizations functions and the modification of the function interfaces. This is avoided at the highest level by annotations. For instance `f(x)@` will be transformed to `start(f,x)`: a task executing f with entry x will be spawned. From one of these levels, a compiler generates PACLIB C code with all explicit task creations and synchronizations.

Table 3.6: Classification – PACLIB with PD.

Languages	Fork/Join	Memory	Load-balancing	Grain
C and PD	Threads	Shared	Task queue	Medium / Fine

This provides a friendly interface for the end user. The problem of task mapping is not really addressed. It is handled by the running time system which is based on virtual processors [90] (*cf* also §3.5) and light-weight processes. One global queue contains all the tasks that are active but not yet running. The scheduler, provided by the μSYSTEM, selects tasks for execution from the head of the queue. A global list is available for memory management.

The development of applications is illustrated either at the C level in [54], or at the highest level using PD in [91]. High speed-ups are reported on basic examples.

The PACLIB kernel is implemented on a shared memory machine, a Sequent Symmetry. Anyway, the functional model is also well suited to distributed memory. This latter model is also considered by the same research group, see §3.10.

3.10 STURM

The above PACLIB kernel is essentially the substratum for the STURM kernel designed either for shared memory machines [56] or for networks of such machines [7]. The task management is thus inherited from PACLIB whilst the memory management is redesigned. The kernel is implemented in C and a C++ shell is provided around the C interface.

The programming model is still based on *fork /join* and the tasks are scheduled via a global queue (with user-specified time-slicing for preemptive scheduling). Concerning the memory, the system simulates an infinite heap from which blocks can be allocated. The heap is organized is a set of clusters. On request for a free block, a hierarchical search is done from the local cluster to the whole machine (all the processors may be interrupted for garbage collection).

Table 3.7: Classification – STURM.

Languages	Fork/Join	Memory	Load-balancing	Grain
C / C++	Threads	Shared / Distributed	Task queue	Medium / Fine

The kernel is running on shared memory machines but we are not aware of experimental data on distributed memory machines or on networks.

3.11 From PAC to GIVARO

The GIVARO library for computer algebra and dynamic load-balancing in distributed environments, comes from the older library PAC.

The PAC library [83, 84] was written in C and was using a message passing programming model. Experiments and speed-ups on various algorithms can be found in [102, 82, 81, 97] using up to 32 processors and vector units. Experiments have also been done on 128 transputers as reported in [94]. Then PAC has been redesigned in C++: PAC++ [44, 24] was formed of a sequential computer algebra kernel based of GNU GMP and on a distributed thread system. A programming model based on *fork / join* capabilities (themselves provided by the runtime layer ATHAPASCAN-0a [23, 8]) and on a *divide and conquer* style, has thus followed the older message passing one. We refer to [44, 46, 24] for details and experiments. Neither of these two libraries was providing dynamic load balancing of tasks. However, experiments with them have been necessary in order to fix the best suited programming model in subsequent developments.

This has led to GIVARO [43, 45] which extends the model and includes automatic load-balancing. Still using *fork / join* in a *divide and conquer* style, a program is written at an abstract level of granularity, the tasks are mapped onto the processors at run-time. Two types of synchronous remote calls to functions are available: to *split* computations into independent subtasks the programmer can use a call to n functions; to *merge* results or to *cooperate during computation* GIVARO offers the concept of *multi-function* [5] through a synchronization function. The underlying runtime support is ATHAPASCAN-0_{MP} [80] which extends ATHAPASCAN-0a by the implementation of multi-functions.

The expression of parallel programs is distinct from the way they are executed by the current *scheduler*: the scheduler is invoked at each call to a function. Using *cost informations* given by the users, the scheduler can also takes some algorithmic choices, such as choosing the splitting factor at an n-arity call. At runtime, each task can be stopped then continued on a different site of execution (this was designed for the manipulation of algebraic numbers [44, 45]). The mechanism is based on user defined continuation functions.

Table 3.8: Classification – GIVARO.

Languages	Fork/Join	Memory	Load-balancing	Grain
C++	Threads	Shared / Distributed	Yes / Task queue	Medium / Fine

GIVARO has been tested mainly on an IBM SP2 [44, 24, 43]. Some different strategies of load-balancing are available via default schedulers. However, the modular design of the library allows the user to define its own new schedulers without any modification of the parallel programs.

3.12 GIVARO with ATHAPASCAN

The sequential library of GIVARO [43] (which includes basic data types and their bufferization, and basic arithmetics in C++) can be used in sequential or with other runtime supports.

Table 3.9: Classification – GIVARO with ATHAPASCAN-1.

Languages	Fork/Join	Memory	Load-balancing	Grain
C++	Threads / Virt. sh. mem.	Sh. / Distr.	Yes / Task queue	Medium / Fine

Another parallel interface for GIVARO is under development using ATHAPASCAN-1 as runtime layer. The programming model is still based on *fork / join*, global synchronizations are done via a virtually shared global memory [16] (read, write and accumulation). A program is written at a virtual level of granularity, a dependence graph is built from an analysis of the input parameters of the tasks and is mapped at execution (the process could be done statically in certain cases).

The software is built from two layers: one for parallelism expression and one for task scheduling and mapping. Sophisticated task schedulers are available and take into account both *cost informations* about tasks and *locality of input data*. The scheduling library can be augmented with user defined strategies.

The main difference between this approach and the one in the previous section, is in the model of global synchronization between threads. On the one hand *multi-procedures* and on the other hand *accesses to a virtual shared memory* provide this capability.

Very first experiments with this approach can be found in [33].

3.13 Other parallel extensions of sequential softwares

3.13.1 ALDES and SAC-2

The library SAC-2 [25] written in ALDES [68] has been extended with means for parallel computing [92, 93]. The main goal is to provide to the programmer, a way to specify concurrency in a very simple way – with no great change in the sequential codes. This is realized by the use of *futures* [59, 49] to extend the sequential semantic. The associated concurrent tasks are launched via *asynchronous procedure calls*. Tasks are managed by a *scheduler*. They are sent – on request – to *algorithm servers*.

Good speed-ups are reported on very regular examples such as Chinese remaindering for computing polynomials resultants [93].

An implementation of SAC-2 has been done on Cray machines, corresponding experiments may be found in [58]. Other parallel developments with SAC-2 relies on the translation of ALDES into Fortran for the parallel aspects [87]. Here, the computation is *data driven* (cells for cylindrical algebraic computation). Several identical programs are distributed among the processors of a shared memory machine. Each processor then take a task to perform in a centralized queue as soon as the previous job is finished. Good speed-ups are reported on few processors.

On network of workstations ALTS [99] is a library which provide basic communication routines for an ALDES programmer. A unique program is distributed over the network. Data are shared by the processors via a virtual global memory which permits communications between tasks. Tasks and global events (interruption) somehow allow to relax the constraints imposed by the unique flow of control, this allows speculative parallelism. Some experiments are reported but the quite coarse-grained tasks (implemented using Unix processes even if the abstract level is lower) limits the power of the software if dynamic load-balancing is a relevant issue.

Using threads and relying on the scheduling strategies of the all-cache parallel machine KSR1, some remarks and experiments are given in [61] about the parallel implementation of MAS [60].

3.13.2 MAPLE

In addition to the use of DSC seen at §3.4, other systems have been proposed to interface MAPLE in parallel.

A *manager/worker* approach is considered in [95] and applied for instance to Gröbner bases computation [96]. Using the parallel language STRAND [41] (guarded Horn clause type), ||MAPLE|| kernel is developed and tested on both shared and distributed memory machines. A simple set of interesting interface routines between STRAND and MAPLE is proposed, it mainly provides the *splitting* of inherently parallel tasks and the *combination* of results. The *manager/worker* scheme is possible and recommended in case of load-balancing requirements (this latter problem is not considered). This approach clearly allows to easy take advantage of the whole MAPLE LIBRARY to run complex codes. Good speed-ups are reported on 16 processors on some basic problems.

Other investigations using MAPLE are presented in [6]. The approach is different from the one above since a modified version of the MAPLE kernel is used. Classical message-passing concepts are implemented but high speed-ups are reported on many processors (81 nodes with two Cpus).

3.13.3 MUPAD

A parallel version of MUPAD [42] should be released in the future. Some well known concepts for shared memory machines (*e.g.* job queues, parallel loops ...) are planned to be provided [75, 51].

3.13.4 REDUCE

Some attempts have been made to extend the capabilities of REDUCE for vector and parallel machines. We have seen REDUCE on Cray machines at §3.7.

Also using REDUCE another direction has been taken in [37]. To use coarse grain parallelism, this paper proposes a way to automatically generate parallelism at compilation. Parallelism is detected at the level of a function invocation, a data flow analysis is proposed to provide the necessary semantic information. This study remains theoretical (concerning the parallel execution), no parallel experiment is given.

3.13.5 ALDOR

The library Π^{it} is currently under development [9] at the ETH Zurich. The goal of the project is to incorporate parallel constructions in the general purpose language Aldor [110, 109] in order to write *portable* parallel programs using various types of systems and machines.

The library is formed of a few set of packages on which high level parallel programming models are based. These packages concern basic communications, asynchronous calls to functions and a job scheduling interface. Parallel programming models are: *fork / join*, parallel map using iterators on splitted data structures and to reconstruct results from sub-computations. Scheduling strategies are under development and tuned on different architectures.

3.13.6 CALYPSO and ALGBENCH

The parallel computer algebra library CALYPSO presented in[17] is a set of C++ parallel classes and programs for infinite precision arithmetic. Using MPI these algorithms have been tested on various parallel architectures (programs are parameterized by the number of available processors). This library can be linked with other ones, this is done for instance with the ALGBENCH system [69], viewing CALYPSO as its arithmetic parallel engine.

3.14 Integrations of MPI or PVM

The efforts related to MPI [40] and PVM [47] have produced specifications intended for the portable development of message-passing applications. Implementations exist on various architectures, the use of MPI or PVM broadens the applicability of the software.

Based on these remarks, the three tool-boxes below provide the integration of either MPI or PVM with several existing softwares. This should be the first step towards more general-purpose applications because clearly, this is not enough. The lack of dynamic management of the processes and of dynamic load-balancing should be handled by a complementary layer.

3.14.1 STAR / MPI

We refer to [27] for the integration of MPI with existing softwares. The author presents the implementation of a classical basic master / slave interface between MPI and GNU Common LISP or GAP [88].

3.14.2 Tools for parallel mathematical computations

A set of tools for data communications between concurrent tasks is presented in [107]. One software permits to run MAXIMA as a PVM task through a Common LISP interface to PVM. Another library interfaces SACLIB [11] and related computer algebra systems.

3.14.3 FOXBOX

The software FOXBOX for manipulating black box representations in symbolic calculus, provides in particular a parallel interface [31]. A basic client / server protocol manages parallel black box objects (C++). Currently, applications are realized using MPI but the software could take advantage of other parallel systems. This parallel library is part of a wider software that also allows the use of general purpose computer algebra systems.

3.15 Protocols for mathematical communications

When homogeneous or heterogeneous computational units are connected and can physically send and receive messages, one need a common protocol to efficiently exchange data.

On the one hand, in the parallel computation community, especially in computer algebra, from the sections above we see that for many years, each research group has defined its own basic means of communicating mathematical informations and protocols.

On the other hand, in the computer algebra community, many efforts have been done to provide connectivity between different softwares in a distributed setting. This includes the development of means for communicating mathematical informations between applications, we may refer to [1] and to references therein.

Since parallel and distributed computations are main applications of these latter studies on protocols, and since communicating is a main concern in parallelism we may hope that these two directions will learn from each other. In particular, as we have seen, parallel programming relies on various models to express parallelism (*message passing*, *fork / join*, BSP, ...): one question is to have common means of communicating and interfaces that are independent of the chosen model *i.e.* that can be used with any model.

3.16 Conclusion

We have seen that a lot of work has been done to exploit parallelism in computer algebra. If we go back to the three main possible approaches A, B and C discussed in the introduction, we may conclude that the three should be considered as complementary means. And the same holds, for the more detailed directions B1, B2 and B3. Indeed for instance, we think that the studies done somehow separately with standard systems and with new ones has came to maturity, and could be integrated in a unique framework. To use a standard computer algebra system does not exempt from taking into account accepted facts in parallelism. To build a new parallel computer algebra system does not exempt from knowing about the best computer algebra algorithms.

Computer algebra system nowadays are more open toward outside than during the past, parallelism will certainly benefit a lot from this evolution. Will this evolution, together with progresses in providing efficient and standard communication means, accelerate the emergence of standard ideas? We believe that *data-flow graphs*, *functional programming* together with threads and *dynamic load-balancing* with adaptative granularity/locality and *cost informations*, can be such ideas.

Bibliography

[1] J. Abbott, A. van Leeuwen., and A. Strotmann, *Objectives of* OPENMATH, Tech. Report TR 12, RIACA, Amsterdam, jul 1996.

[2] S. Ahuja, N. Carriero, and D. Gelernter, *Linda and friends*, IEEE Computer (1986), 26–34.

[3] B. Amrhein, O. Gloor, and W. Küchlin, *A case study of multi-threaded Gröbner basis completion*, International Symposium on Symbolic and Algebraic Computation, Zurich, Suisse, ACM Press, July 1996, pp. 95–102.

[4] B. Amrhein and W. Küchlin, *Parallel computer algebra on the desk-top*, 1995, Electronic proceedings, First IMACS Conference on Applications of Computer Algebra, Albuquerque, NM USA.

[5] J.P. Banâtre, M. Banâtre, and F. Ployette, *The concept of multi-fonction: a general structuring tool for distributed operating systems*, 6th Int. Conf. on Distributed Computing Systems, Cambridge, UK, 1986.

[6] L. Bernardin, MAPLE *on a massively parallel, distributed memory machine*, Second International Symposium on Parallel Symbolic Computation (PASCO'97), Maui, Hawaii, USA, Jul 1997.

[7] P.G. Bertoli, H. Hong, A. Neubacher, W. Schreiner, and V. Stahl, *The C++ interface to the* STURM *distributed multi-processor kernel*, Tech. Report TR 94-32, RISC Institute, Linz, Austria, 1994.

[8] J. Briat, I. Ginzburg, and M. Pasin, ATHAPSACAN-*0b user's manual (Version 2.4.7)*, Tech. Report APACHE, IMAG Grenoble France, 1997, `http://www-apache.imag.fr/`.

[9] M. Bronstein and T. Gautier, Personal communication, 1997.

[10] T. Bubeck, M. Hiller, W. Küchlin, and W. Rosenstiel, *Distributed symbolic computation with DTS*, Proc. of IRREGULAR'95, Lyon, France, LNCS 980, Springer-Verlag, Sep. 1995.

[11] B. Buchberger, G.E. Collins, and *al.*, SACLIB *1.1 user's guide*, Tech. Report TR 93-19, RISC Institute, Linz, Austria, 1993.

[12] P. Buhr and R. Stroobosscher, *The* μSYSTEM*: providing light-weight concurrency on shared-memory multiprocessor computers running unix*, Software – Practice and Experience **20** (1990), no. 9, 929–964.

[13] CAP'88, CAP, Premier Colloque International sur le Calcul Formel et le Parallélisme Grenoble France, J. Della-Dora, J. Fitch Ed., North-Holland, 1988.

[14] CAP'90, CAP, Second International Workshop on Computer Algebra and Parallelism, Ithaca USA, R.E. Zippel Ed., LNCS 584 Springer-Verlag, 1990.

[15] T.L. Casavant and J.G. Khul, *A taxonomy of scheduling in general-purpose distributed computing systems*, IEEE Transactions on Software Engineering (1988), no. 14, 141–154.

[16] G. Cavalheiro, M. Doreille, F. Galilée, and J.L. Roch, ATHAPSACAN-*1 user's manual*, Tech. Report APACHE, IMAG Grenoble France, 1997, `http://www-apache.imag.fr/`.

[17] G. Cesari, CALYPSO*: a computer algebra library for parallel symbolic computation*, Second International Symposium on Parallel Symbolic Computation (PASCO'97), Maui, Hawaii, USA, Jul 1997.

[18] K.C. Chan, A. Díaz, and E. Kaltofen, *A distributed approach to problem solving in Maple*, Maple Summer Workshop and Symposium, Birkhäuser Verlag, Boston, 1994, pp. 13–21.

[19] B. Char, J. Johnson, D. Saunders, and A.P. Wack, *Some experiments with parallel Bignum arithmetic*, First International Symposium on Parallel Symbolic Computation (PASCO'94), Lecture Notes Series in Computing – Vol. 5, World Scientific, 1994, pp. 94–103.

[20] B.W. Char, *Progress report on a system for general purpose parallel symbolic algebraic computations*, International Symposium on Symbolic and Algebraic Computation, Tokyo, Japan, ACM Press, pp 96–103, 1990.

[21] ______, *A users's guide to sugarbush – parallel Maple through Linda*, Tech. Report MCS-94-01, Drexel University, Department of Mathematics and Computer Science, 1994.

[22] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt, MAPLE V *language reference manual*, Springer-Verlag, 1991.

[23] M. Christaller, *Athapsacan-0a sur PVM.3*, Tech. Report APACHE 11, IMAG Grenoble France, 1994.

[24] Christaller, M. and Castaneda Retiz, M.R. and Gautier, T., *Control parallelism on top of pvm: the* ATHAPASCAN *environment*, Second European PVM User's Group, Ecole Nationale Supérieure, Lyon, France, Hermès, sept. 1995.

[25] G.E. Collins and R. Loos, *Specification and index of sac-2 algorithms*, Tech. Report WSI-90-4, Universität Tübingen, Germany, 1990.

[26] M.E. Conway, *Design of separable transition diagram compiler*, Comm. of ACM **6** (1963), no. 7, 396–408.

[27] G. Cooperman, STAR / MPI*: binding a parallel library to interactive symbolic algebra systems*, International Symposium on Symbolic and Algebraic Computation, Montreal, Canada, ACM Press, July 1995, pp. 126–132.

[28] S. Dalmas, M. Gaëtano, and A. Sausse, *Distributed computer algebra: the Central Control approach*, First International Symposium on Parallel Symbolic Computation (PASCO'94), Lecture Notes Series in Computing – Vol. 5, World Scientific, 1994, pp. 104–113.

[29] Dalmas, S. and Gaëtano, M., *The Central Control approach: a flexible tool to experiment with distributed symbolic computation*, 1995, Electronic proceedings, First IMACS Conference on Applications of Computer Algebra, Albuquerque, NM USA.

[30] A. Díaz, M. Hitz, E. Kaltofen, A. Lobo, and T. Valente, *Process scheduling in* DSC *and the large sparse linear systems challenge*, Journal of Symbolic Computation **19** (1995), no. 1–3, 269–282.

[31] A. Díaz and E. Kaltofen, FoxBox*: a system for manipulating symbolic objects in black box representation*, Manuscript, Dpt. Math., North Carolina State University, NC USA, 1997.

[32] A. Díaz, E. Kaltofen, K. Schmitz, and T. Valente, *DSC: A system for Distributed Symbolic Computation*, International Symposium on Symbolic and Algebraic Computation, Bonn, Germany, ACM Press, 1991, pp. 323–332.

[33] J.G. Dumas, *Calcul parallèle avec des nombres algébriques réels*, june 1997, Rapport de DEA, INPG, Grenoble, France.

[34] G. Authié et al., *Algorithmes parallèles, analyse et conception*, Hermès, 1994.

[35] ———, *Parallélisme et applications irrégulières*, Hermès, 1995.

[36] G. Fagg and J. Dongarra, *An integration of the* PVM *and* MPI *systems*, Calculateurs Parallèles **2** (1996).

[37] J. Fitch, *Can Reduce be run in parallel?*, International Symposium on Symbolic and Algebraic Computation, Portland, Oregon, ACM-Press, Jul 1989, pp. 155–162.

[38] J.P. Fitch and J.B Marti, *On the complexity of finding short vectors in integer lattices*, EUROCAL 83, London, UK, Springer, LNCS 162, 1983, pp. 236–244.

[39] HPF Forum, *High performance fortran language specification, version 1.1*, Tech. report, Rice University, Houston Texas, USA, 1994.

[40] MPI Forum, *A Message-Passing Interface standard*, Internation Journal of Super Computer Application **8** (1994), no. 3-4.

[41] I. Foster and S. Taylor, *Strand - New concepts in parallel programming*, Prentice-Hall, 1989.

[42] B. Fuchssteiner and *al.*, MuPAD *User's Manual*, 1996, Wiley Ltd.

[43] T. Gautier, *Calcul formel et parallélisme : conception du système* Givaro *et applications au calcul dans les extensions algébriques*, Ph.D. thesis, INPG, Grenoble, France, 1996.

[44] T. Gautier and J.L. Roch, *PAC++ System and Parallel Algebraic Numbers Computation*, First International Symposium on Parallel Symbolic Computation (PASCO'94), Lecture Notes Series in Computing – Vol. 5, World Scientific, 1994, pp. 145–153.

[45] T. Gautier and J.L. Roch, *Fast parallel Algebraic Numbers Computations*, Second International Symposium on Parallel Symbolic Computation (PASCO'97), Maui, Hawaii, USA, Jul 1997.

[46] T. Gautier, J.L. Roch, and G. Villard, *Regular versus irregular problems and algorithms*, Proc. of IRREGULAR'95, Lyon, France, LNCS 980, Springer-Verlag, Sep. 1995.

[47] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *Parallel Virtual Machine*, MIT Press, 1994.

[48] D. Gelernter, *Generative communication in Linda*, Tech. Report RR-294, Yaleu/DCS, Nov 1983.

[49] R.H. Halstead, *Multilisp: a language for concurrent symbolic computation*, ACM Trans. Progr. Lang. Sys. **7** (1985), no. 4, 501–538.

[50] ———, *Parallel symbolic computing*, IEEE Computer **19** (1986), no. 8, 35–43.

[51] C. Heckler, T. Metzner, and P. Zimmermann, *Progress report on parallelism in* MUPAD, 1997, MUPAD Report, Dpt. Math. Comp. Sc., Univ. Paderborn, Germany.

[52] J.M.D. Hill and *al.*, BSPLIB*: the* BSP *Programming Library*, Tech. report, Oxford University Computing Laboratory, UK, may 1997.

[53] C. Hoare, *Communicating sequential processes*, Communication of the ACM **8** (1978), 666–677.

[54] H. Hong and H.W. Loidl, *Parallel computation of modular multivariate polynomial resultants on shared memory machines*, CONPAR 94 - VAPP VI, Linz, Austria, LNCS 854, September 1994, pp. 325–336.

[55] H. Hong, A. Neubacher, and W. Schreiner, *The design of the* SACLIB / PACLIB *kernels*, DISCO'93, Gmunden, Austria (A. Miola, ed.), Springer, LNCS 722, 1993.

[56] H. Hong, A. Neubacher, W. Schreiner, and V. Stahl, *The C++ interface to the* STURM *multi-processor kernel*, Tech. Report TR 94-31, RISC Institute, Linz, Austria, 1994.

[57] H. Hong, W. Schreiner, A. Neubacher, and *al.*, PACLIB *user's manual*, Tech. Report TR 92-32, RISC Institute, Linz, Austria, 1992.

[58] R. Johnson, J, *Some issues in designing algebraic algorithms for the CRAY X-MP*, Tech. Report 88-02, Univ. of Delaware. Center for Mathematical Computation, 1988.

[59] P. Knueven, P.G. Hibbard, and B.W. Leverett, *A language system for multiprocessor environment*, Fourth International Conference on the Design and Implementation of Algorithmic Languages, June 1976, pp. 264–274.

[60] H. Kredel, *MAS Modula-2 algebra system*, DISCO'90, Capri, Italy (A. Miola, ed.), Springer, LNCS 429, Apr. 1993, pp. 270–271.

[61] ______, *Computer algebra on a KSR1 parallel computer*, Tech. report, Rechenzentruum Universität Mannheim, Germany, 1994.

[62] W. Küchlin, *PARSAC-2: a parallel SAC-2 based on threads*, Eights Int. Symposium on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Tokyo, Japan., LNCS 508, Springer-Verlag, 1990.

[63] ______, *The S-Threads environment for parallel symbolic computation*, Second International Workshop on Computer Algebra and Parallelism, Ithaca, USA, LNCS 584, Springer-Verlag, 1990.

[64] ______, *On the multi-threaded computation of integral polynomial greatest common divisors*, International Symposium on Symbolic and Algebraic Computation, Bonn, Germany, ACM Press, 1991, pp. 333–342.

[65] W. Küchlin, PARSAC-2*: parallel computer algebra on the desk-top*, Computer algebra in science and engineering (J. Grabmeier J. Fleisher and F. Hehl, eds.), World Scientific, 1995, pp. 24–43.

[66] H.R. Lee, B.D. Saunders, and R. Shtokhamer, *On Scheduling Algebraic Algorithms for Parallel Execution*, Tech. report, Delaware Department of Computer and Information Sciences, CIS TR-9015, 1990.

[67] A. Lobo, *Matrix-free linear system solving and applications to symbolic computation*, Ph.D. thesis, Dept. Comp. Sc., Rensselaer Polytech. Instit., Troy, New York, Dec. 1995.

[68] R.G.K Loos, *The algorithm description language Aldes (report)*, ACM Sigsam Bull. **1** (1976), no. 10, 15–39.

[69] R. Maeder, *The design of the* SACLIB / PACLIB *kernels*, DISCO'92, Bath, UK (J. Fitch, ed.), Springer, LNCS 721, 1993.

[70] J.B Marti, *Compilation techniques for a control flow concurrent Lisp system*, Proceedings of the 1980 Lisp Conference, 1980, pp. 203–207.

[71] H. Melenk and W. Neun, *Reduce User's Guide for CRAY 1/X-MP Series Running COS*, Tech. Report 87-4, Konrad Zuse-Zentrum für Informationstecnik Berlin, 1987.

[72] ———, *Parallel polynomial operators in the large Buchberger algorithm*, CAP , premier colloque international sur le calcul formel et le parallélisme Grenoble, J. Della-Dora, J. Fitch éd., North-Holland, 1988.

[73] ———, *Very large Gröbner basis calculations*, Second International Workshop on Computer Algebra and Parallelism, Ithaca, USA, LNCS 584, Springer-Verlag, 1990.

[74] E. Mohr, D. Kranz, and R. Halstead Jr., *Lazy task creation: a technique for increasing granularity of parallel programs*, IEEE Trans. on Par. Distr. Sys. **2** (1991), no. 3, 264–280.

[75] H. Naundorf, *Parallelism in* MUPAD, 1995, Electronic proceedings, First IMACS Conference on Applications of Computer Algebra, Albuquerque, NM USA.

[76] W. Neun, *Using PVM based software for parallel computation in computer algebra*, 1994, The Rhine Workshop on Computer Algebra, Karlsruhe, Germany.

[77] ———, REDUCE *on a massively parallel system*, 1995, Electronic proceedings, First IMACS Conference on Applications of Computer Algebra, Albuquerque, NM USA.

[78] C.G. Ponder, *Evaluation of performance enhancements in algebraic manipulation systems*, Ph.D. thesis, Berkeley Univ. of Californy, 1988.

[79] A. Ranade, *A framework for analyzing locality and portability issues in parallel computing*, Parallel Architectures and their Efficient Use, LNCS 678, 1993, pp. 185–194.

[80] M. Rivière, Concepts structurants pour la mise en oeuvre d'applications irrégulières : application au support exécutif parallèle ATHAPASCAN, INPG, Grenoble, France (Thèse en préparation).

[81] F. Roch, *Calcul formel et parallélisme : forme normale d'Hermite, méthodes de calcul et parallélisation*, Ph.D. thesis, INPG, Grenoble, France, 1990.

[82] J.L. Roch, *Calcul formel et parallélisme : l'architecture du système* PAC *et son arithmétique rationnelle*, Ph.D. thesis, INPG, Grenoble, France, 1989.

[83] ———, *The* PAC *system : general presentation*, Second International Workshop on Computer Algebra and Parallelism, Ithaca, USA, LNCS 584, Springer-Verlag, 1990.

[84] J.L. Roch, F. Siebert, P. Sénéchaud, and G. Villard, *Computer Algebra on a MIMD machine*, SIGSAM Bulletin **23** (1989), no. 11, 16–32.

[85] J.L. Roch, A. Vermeerbergen, and G. Villard, *Cost prediction for load-balancing: application to algebraic computations*, CONPAR 92, Lyon, France, LNCS 634, September 1992.

[86] ______, *A new load-prediction scheme based on algorithmic cost functions*, CONPAR 94, Linz Austria, LNCS 854, Sep. 1994.

[87] B.D. Saunders, H.R. Lee, and S.K. Abdali, *A parallel implementation of the cylindrical algebraic decomposition algorithm*, International Symposium on Symbolic and Algebraic Computation, Portland, Oregon, ACM-Press, Jul 1989, pp. 298–307.

[88] M. Schönert and *al.*, GAP – Groups, Algorithms and Programming, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, Fifth Edition.

[89] W. Schreiner, *Parallel functional programming for computer algebra*, Ph.D. thesis, Johannes Kepler University, Linz, Austria, Sept. 1994.

[90] ______, *Virtual tasks for the* PACLIB *kernel*, CONPAR 94 - VAPP VI, Linz, Austria, LNCS 854, September 1994, pp. 533–544.

[91] ______, *A para-functional programming interface for a parallel computer algebra package*, Journal of Symbolic Computation **21** (1996), no. 4-6, 593–614.

[92] S. Seitz, *Parallel algorithm development*, CAP , premier colloque international sur le calcul formel et le parallélisme Grenoble, J. Della-Dora, J. Fitch éd., North-Holland, 1988.

[93] ______, *Algebraic computing on a local net*, Second International Workshop on Computer Algebra and Parallelism, Ithaca, USA, LNCS 584, Springer-Verlag, 1990.

[94] F. Siebert and G. Villard, *PAC : First experiments on a 128 transputers Meganode*, International Symposium on Symbolic and Algebraic Computation, Bonn Germany, ACM Press, 1991.

[95] K. Siegl, *Parallelizing algorithms for symbolic computation using* $||$*Maple*$||$, Tech. Report 93-08, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.

[96] ______, *A parallel factorization tree Gröbner basis algorithm*, First International Symposium on Parallel Symbolic Computation (PASCO'94), Lecture Notes Series in Computing – Vol. 5, World Scientific, 1994, pp. 356–362.

[97] P. Sénéchaud, *Calcul formel et parallélisme : bases de Gröbner booléennes, méthodes de calcul, applications et parallélisation*, Ph.D. thesis, INPG, Grenoble, France, 1990.

[98] M. Snir, *Scalable parallel computers and scalable parallel codes: from theory to practice*, Parallel Architectures and their Efficient Use, LNCS 678, 1993, pp. 176–184.

[99] T.C. Torgersen, *Distributing symbolic computations on a network of workstations*, Computational Algebra, volume 151 of LN Pure and Applied Math., Marcel Dekker Inc., 1994, pp. 233–246.

[100] T. Valente, *A distributed approach to proving large number primes*, Ph.D. thesis, Dept. Comp. Sc., Rensselaer Polytech. Instit., Troy, New York, 1992.

[101] L. Valiant, *A bridging model for parallel computation*, Communication ACM **33** (1990), 103–111.

[102] G. Villard, *Calcul formel et parallélisme : résolution de systèmes linéaires*, Ph.D. thesis, INPG, Grenoble, France, 1988.

[103] P.S. Wang, *Parallel univariate polynomial factorization on shared-memory multiprocessors*, International Symposium on Symbolic and Algebraic Computation, Tokyo, Japan, ACM Press, 1990.

[104] ______, *Parallel univariate p-adic lifting on shared-memory multiprocessors*, International Symposium on Symbolic and Algebraic Computation, Berkeley California USA, ACM Press, July 1992, pp. 168–176.

[105] ______, *Parallel polynomial operations: a progress report*, First International Symposium on Parallel Symbolic Computation (PASCO'94), Lecture Notes Series in Computing – Vol. 5, World Scientific, 1994, pp. 394–404.

[106] ______, *Parallel polynomial operations on SMPs: an overview*, Journal of Symbolic Computation **21** (1996), no. 4-6, 397–410.

[107] ______, *Tools for parallel / distributed mathematical computation*, Second International Symposium on Parallel Symbolic Computation (PASCO'97), Maui, Hawaii, USA, Jul 1997.

[108] S. Watt, *Bounded parallelism in Computer Algebra*, Ph.D. thesis, University of Waterloo, Ontario, 1985.

[109] S.M. Watt and *al.*, The A♯ user's guide, NAG Ltd.

[110] ______, *A first report on the A♯ compiler*, International Symposium on Symbolic and Algebraic Computation, Oxford, UK, ACM Press, July 1994, pp. 25–31.

[111] D Weeks, *Adaptation of SAC-2 algorithms for an SIMD machine*, CAP , premier colloque international sur le calcul formel et le parallélisme Grenoble, J. Della-Dora, J. Fitch éd., North-Holland, 1988, pp. 167–178.

Chapter 4

Parallel linear algebra.

To be provided later.

Contents