Measuring Concurrency of Regular Distributed Computations

Cyrille Bareau[1], Benoît Caillaud[2], Claude Jard[1]
René Thoraval[3]
E-mail: {name}@irisa.fr

[1] IRISA, Campus de Beaulieu, 35042 Rennes cedex, France
[2] LFCS, JCMB, King's Buildings, The University of Edinburgh, Edinburgh, EH9 3JZ, UK
[3] Université de Nantes, Section Informatique, 2 rue de la Houssinière 44072 Nantes cedex 03, France

Abstract. In this paper we present a concurrency measure that is especially adapted to distributed programs that exhibit regular run-time behaviours, including many programs that are obtained by automatic parallelization of sequential code. This measure is based on the antichain lattice of the partial order that models the distributed execution under consideration. We show the conditions under which the measure is computable on an infinite execution that is the repetition of a finite pattern. There, the measure can be computed by considering only a bounded number of patterns, the bound being at most the number of processors.

1 Introduction

The trend towards the use of distributed memory parallel machines is very evident. However, their programming environments have to be significantly improved, especially in the field we are mainly interested in: semi-automated distribution of sequential code for scientific computing. Indeed, programmers need sophisticated performance evaluation tools. However, there is no well-accepted "complexity" criterion for distributed programs, for the behaviours of asynchronous message-passing programs are not yet sufficiently understood. It is also very difficult to design tools that can give relevant performance information from static analysis of distributed code.

A research axis is to study runs of a distributed program instead of the program itself. Especially, it needs to define concurrency measures, i.e. measures that can help reveal the synchronization structure of a computation, as opposed to the traditional ones (message count, for instance) which only give quantitative information about the computation. For this it is now usual to take a distributed execution as a partially ordered set of events that are causally related by process sequentiality and interprocess communication [14]. As far as we know, the first concurrency measure that takes account of causality was proposed by Charron-Bost [5], followed by [9, 11, 16].

It is shown in [11] that either these measures are too inaccurate or their computational complexity makes them impracticable. It is also important to observe that these measures only deal with finite runs. Although executions of reactive programs are usually infinite. Further, executions of distributed programs for scientific applications, even finite, are usually very long. Nevertheless, [5] and [11] give some encouraging results. The measure in [5] is shown to behave well with respect to a particular kind of concatenation operator on computations. On the other hand, the work in [11] shows that there is some hope of obtaining practical concurrency measures for particular classes of executions (for instance, [11] considers executions that can be modeled as so-called interval orders).

We address several closely related problems. First, is it possible to define a measure that gives significant values even in the case of infinite executions? And if such a measure exists, does there exist a class of infinite computations for which the measure can be computed? Naturally, if such computations do exist, they must exhibit some kind of regularity that the measure must take into account. Moreover, as an infinite execution can be seen as a limit of a sequence of finite ones, the computation of the measure should not depend on the size of the order that models an execution: it must only depend on the size of a bounded subset of this order.

In this paper we give a first positive answer to these problems. We define a concurrency measure and a class of executions that exhibit a particular kind of regularity: an execution in this class can be modeled as either an infinite or a finite repetition of a finite elementary order we call a *basic pattern.* If this pattern is well connected, we establish for any regular execution (even infinite) that our measure is bounded and can be computed merely from a bounded number of repetitions of this basic pattern. Such an execution is said to be *well-synchronized.* Finally, we show that this property is of interest for semi-automated distribution of sequential programs.

Paper Organization. We first describe the formal framework used throughout the paper and present a model of an execution of a distributed program as a labeled poset.

We then define our concurrency measure μ on a distributed execution. It is expressed in terms of the antichain lattice of the associated labeled poset. Measure μ associates a value with each event in the execution. An event with a small value denotes a strong synchronization, that is, an execution bottleneck.

We then formally define regular executions and well-synchronization. For a regular well-synchronized execution, we show that μ is bounded, reflects regularity and can be computed on at most $2N-1$ repetitions of the basic pattern (where N is the number of processes). This enables the definition of a measure μ_∞ on the events of the basic pattern. In the case of very long or infinite executions, the computation of this measure suffices to determine μ on almost the whole execution. We also show that the antichain lattice of the infinite repetition of the pattern is regular enough so that μ_∞ can be computed on at most N repetitions of this pattern.

We then compare μ with other measures from the literature.

Finally, we show that our measure is especially relevant for automatically distributed programs.

An extended version of this paper, with more detailed proofs, is available as a technical report [3].

2 Framework

2.1 Definitions and Notations

For an introduction to poset theory see for instance [6].

A set E associated with a partial order relation $\preceq$ is called a *partially ordered set* (*poset* for short) and is denoted by $\mathcal{E} = \langle E, \preceq \rangle$. Let $x, y \in E$: we say that x and y are *comparable* in $\mathcal{E}$ when either $x \preceq y$ or $y \preceq x$, otherwise x and y are said to be *incomparable*, denoted by $x \| y$ (as usual, $x \preceq y \wedge x \neq y$ is denoted by $x \prec y$). A *chain* (resp. an *antichain*) in $\mathcal{E}$ is a subset A of E such that every pair of distinct elements of A are comparable (resp. incomparable). Letting $A \subseteq E$, $\max(A) = \{e \in A \mid \forall f \in A, e \not\prec f\}$ is the set of *maximal* elements in A. The *width* of $\mathcal{E}$ is the maximum number of elements in an antichain in $\mathcal{E}$. The *covering relation* of $\preceq$ is denoted by $-\!\!\prec$, i.e. $e -\!\!\prec f$ (f covers e) $\Leftrightarrow$ ($e \prec f$ and $\nexists g,\ e \prec g \prec f$). For each element e of E, we define $\downarrow e = \{f \in E \mid f \preceq e\}$ (the set of predecessors of e), and for each subset F of E, $\downarrow F = \bigcup_{f \in F} (\downarrow f)$.

The *Hasse diagram* of a poset $\mathcal{E}$ is the directed graph whose vertices are the elements of E and the arcs are the elements of $-\!\!\prec$ (usually, the direction of the arcs is not represented by arrows but must be read bottom-up).

We define a labeled poset Θ as a tuple $\langle E, \preceq, L, \pi \rangle$ consisting of a non-empty poset $\langle E, \preceq \rangle$ of finite width and with no infinitely decreasing chain, a non-empty set L of labels and a labeling function $\pi : E \longrightarrow L$.

The set of antichains of Θ is denoted by $\mathcal{A}(\Theta)$. This set is known to be a distributive lattice when equipped with the partial order $\sqsubseteq$ defined as follows: $\forall A, B \in \mathcal{A}(\Theta),\ A \sqsubseteq B \iff \downarrow A \subseteq \downarrow B$. Moreover it is easy to show that:

Lemma 1. *Let Θ be a labeled poset. Then $\forall A, B \in \mathcal{A}(\Theta)$,*

$$B \text{ covers } A \iff A \sqsubseteq B \text{ and } |\downarrow B \setminus \downarrow A| = 1$$

Then we can define $\mathcal{G}(\Theta) = \langle \mathcal{A}(\Theta), \Gamma(\Theta) \rangle$ where $\Gamma(\Theta) \subseteq \mathcal{A}(\Theta) \times E \times \mathcal{A}(\Theta)$ is the set of edges (A, e, B) such that B covers A and $(\downarrow B) \setminus (\downarrow A) = \{e\}$. We call this graph the labeled Hasse diagram of $\langle \mathcal{A}(\Theta), \sqsubseteq \rangle$.

2.2 Discrete Model of a Distributed Computation

Let us consider a computation of a distributed program, that is, a parallel run of a family $(P_i)_{i \in \{1,..,N\}}$ of N sequential processes that communicate by asynchronously exchanging messages. Let P denote the set $\{1, \ldots, N\}$.

We define a discrete model of this computation as a labeled poset $\Theta = \langle E, \preceq, P, \pi \rangle$ that we will call a *distributed order* in the sequel. The elements of E are significant events that occur during the computation. The partial order $\preceq$ indicates how these events are causally related (causality is based on process sequentiality and interprocess communication). The labeling function $\pi : E \longrightarrow P$ associates with each event the identifier of the process it occurs on.

Process Sequentiality. $\pi^{-1}(\{i\})$ denotes the set of events that occur on any given process P_i. Since P_i runs sequentially, any two events of $\pi^{-1}(\{i\})$ are causally related, that is, $\pi^{-1}(\{i\})$ is modeled as a chain. Thus, the family $(\pi^{-1}(\{i\}))_{i \in P}$ is a N-chain decomposition of the labeled poset Θ, the width of which is therefore no more than N.

Concurrency. As we intend to measure concurrency throughout a computation, we are interested to know how far processes can simultaneously proceed at any given point in it. In the context of our discrete model Θ of a computation, that means we are interested in all the sets of events that are causally unconnected, i.e. the antichains of Θ.

Thus, the distributive lattice $\mathcal{A}(\Theta)$ of antichains of Θ well describes the dynamics of concurrency throughout the computation. Moreover, as the behaviours $(\pi^{-1}(\{i\}))_{i \in P}$ of the processes constitute a N-chain decomposition of Θ, this lattice can be given a graphical representation (see Fig. 1), for its Hasse diagram can be embedded into a N-dimensional grid with one dimension per process.

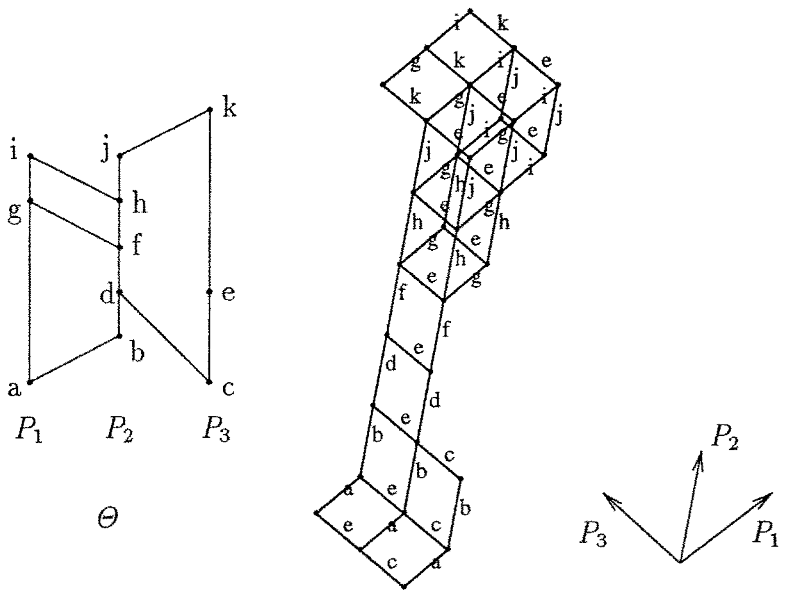

Fig. 1. Hasse diagram of a distributed execution Θ on 3 processors and its graph $\mathcal{G}(\Theta)$

The left part of Fig. 1 shows the Hasse diagram of a computation Θ of a distributed program consisting of three processes P_1, P_2 and P_3. Its middle part illustrates $\mathcal{A}(\Theta)$ by showing the labeled directed graph $\mathcal{G}(\Theta)$ (each direction in the graph corresponds to a processor, as shown in the right part). In this graph, a path from the bottom to the top represents a linear extension of Θ.

3 A Concurrency Measure on the Antichain Lattice

3.1 Definition

The intuition underlying our measure is that the degree of concurrency of an event e is related to "what can happen simultaneously with e". The first idea is to count the number of processors ready to work, i.e. those that are not blocked waiting for e. However this criterion is not accurate enough: for example, it does not enable us to distinguish between events e and h although e is clearly more concurrent than h (e is completely independent on the run of processors P_1 and P_2, whereas h blocks the execution of events i, j, k, therefore the other processors). This idea can be refined by computing the number of antichains containing e, i.e. the number of configurations where the processor that performs e works in parallel with other processors (this is a refinement of a global indicator proposed by Charron-Bost [5]).

Definition 2. Let Θ be a distributed order $\langle E, \preceq, P, \pi\rangle$. For each event e of E:

1. $C_\Theta(e) = \{f \in E \mid f \parallel e\}$ is the set of events that are concurrent with e.
2. $\mathcal{A}_\Theta(e) = \{A \in \mathcal{A}(\Theta) \mid e \in A\}$ is the set of antichains that contain e.

The concurrency measure $\mu_\Theta : E \longrightarrow \mathbb{N} \cup \{\omega\}$ is defined as follows:

$$\forall e \in E, \quad \mu_\Theta(e) = |\mathcal{A}_\Theta(e)|$$

A large value of $\mu_\Theta(e)$ (we write $\mu(e)$ when no confusion is possible) means that many things may happen between the first and the last place where e can occur, that is, e has a great "latency" before actually occurring, and therefore is "very concurrent". In contrast, a little value means that this latency is very short: e is in fact a point of strong synchronization, a "bottleneck".

In the execution of Fig. 1, events g and h for instance have respectively h, e, j, k and g, e as concurrent events; $\mu(h) = |\mathcal{A}_\Theta(h)| = |\{\{h\}, \{h, e\}, \{h, g\}, \{h, e, g\}\}| = 4$ and $\mu(g) = |\{\{g\}, \{g, e\}, \{g, h\}, \{g, e, h\}, \{g, j\}, \{g, e, j\}, \{g, k\}\}| = 7$. The following table gives the values for all events of the execution:

a	b	c	d	e	f	g	h	i	j	k
3	3	3	2	12	2	7	4	5	5	3

In Figure 1, we can see on the lattice that, for instance, events d and f strongly synchronize the execution: nothing happens on processor P_1 simultaneously, and only e can occur on P_3. In contrast, e does not depend on the computation on processors P_1 and P_2 and therefore is very concurrent because there are several possible configurations where it can be executed.

3.2 Computation

In the previous section, we define the measure of an event e in terms of the set of antichains that contain it. To compute this measure, we need to count the edges labeled by e in the labeled Hasse diagram $\mathcal{G}(\Theta)$ of the lattice of the execution:

Proposition 3. *Let Θ be a distributed order and $e \in E$. Then*

$$\mu_\Theta(e) = |\{B \in \mathcal{A}(\Theta) \mid \exists A \in \mathcal{A}(\Theta), (A, e, B) \in \Gamma(\Theta)\}|$$

Proof. From Lemma 1, clearly $\forall A, B \in \mathcal{A}(\Theta), \forall e \in E$, $(A, e, B) \in \Gamma(\Theta)$ iff $A = \max((\downarrow B) \setminus \{e\})$. Then $\mathcal{A}_\Theta(e) = \{B \in \mathcal{A}(\Theta) \mid \exists A \in \mathcal{A}(\Theta), (A, e, B) \in \Gamma(\Theta)\}$. □

This is computationally equivalent to the problem of counting the antichains of an order, which is known to be #P-complete.

This leads us to look for executions for which the number of antichains to be counted does not depend on the length of the execution.

4 Regular and Well-synchronized Executions

The aim of this section is to study a particular class of regular executions: executions which are finite or infinite repetitions of an elementary one (for instance a loop-body). We will show that for a subclass of these executions, μ is computable even in the infinite case by only taking into account at most N (the number of processes) repetitions of a basic "pattern".

4.1 Regular Executions

Definition 4 Regular distributed order. Let $p \in \mathbb{N}^+ \cup \{\omega\}$ and $\Theta = \langle E, \leq, P, \pi \rangle$ be a finite distributed order. Let $E_p = \bigcup_{i \in [0,p[} \varphi_i(E)$ where $(\varphi_i(E))_{i \in [0,p[}$ is a sequence of mutually disjoint isomorphic copies of E. The distributed order $\Theta_p = \langle E_p, \leq_p, P, \pi_p \rangle$ is defined up to order-isomorphism by:

- $\forall i \in [0, p[,\ \pi_p \circ \varphi_i = \pi$
- $\leq_p$ is the least order relation on $E_p \times E_p$ such that:
  - $\forall i \in [0, p[, \forall e, f \in E,\ \varphi_i(e) \leq_p \varphi_i(f) \Longleftrightarrow e \leq f$
  - $\forall i, j \in [0, p[, \forall e, f \in E,\ (\pi(e) = \pi(f)) \wedge (i < j) \Longrightarrow \varphi_i(e) \leq_p \varphi_j(f)$

We say that a distributed order Φ is regular if there exists $p \in \mathbb{N}^+ \cup \{\omega\}$ and a finite distributed order Θ such that Φ is order isomorphic to Θ_p.

When in the sequel we consider a regular distributed order Θ_p, $\leq_p$ and π_p are denoted by $\leq$ and π for the sake of clarity. To speak about events of Θ_p more conveniently, we identify $\varphi_0(E)$ with E and we use $\lambda : E_p \longrightarrow E_p$ defined as follows: $\forall e \in E, \forall i \in [0, p-1[,\ \lambda(\varphi_i(e)) = \varphi_{i+1}(e)$. This allows us to use the non-negative powers of λ instead of φ_i because $\forall e \in E, \forall i \in [0, p[,\ \varphi_i(e) = \lambda^i(e)$ (see Fig 2.(1) for a very simple example) and, as λ is clearly injective, its negative powers can also be used. Moreover, λ has the following property:

Lemma 5. *Let Θ_p be a regular distributed order. Then $\forall e, f \in E, \forall i, j \in [1, p[$,*
$$\lambda^{i-1}(e) \preceq \lambda^{j-1}(f) \Longleftrightarrow \lambda^i(e) \preceq \lambda^j(f)$$

In other words, λ preserves the order relation $\preceq$: it is an order isomorphism from $\bigcup_{i \in [0,p-1[} \lambda^i(E)$ onto $\bigcup_{i \in [1,p[} \lambda^i(E)$. On the Hasse diagram of Θ_p, λ is represented as a "one pattern upward shift". Therefore the Hasse diagram of Θ_p is invariant by pattern-wise translations.

Proof. Let $e, f \in E$ and $i, j \in [1, p[$ such that $\lambda^{i-1}(e) \preceq \lambda^{j-1}(f)$. Then there exists a finite path $\lambda^{i-1}(e) = e_0 \multimap\!\!\!\prec e_1 \multimap\!\!\!\prec \ldots \multimap\!\!\!\prec e_h = \lambda^{j-1}(f)$. From Def. 4, we clearly have $\forall k \in [0, h-1],\ \lambda(e_k) \multimap\!\!\!\prec \lambda(e_{k+1})$, hence the result. □

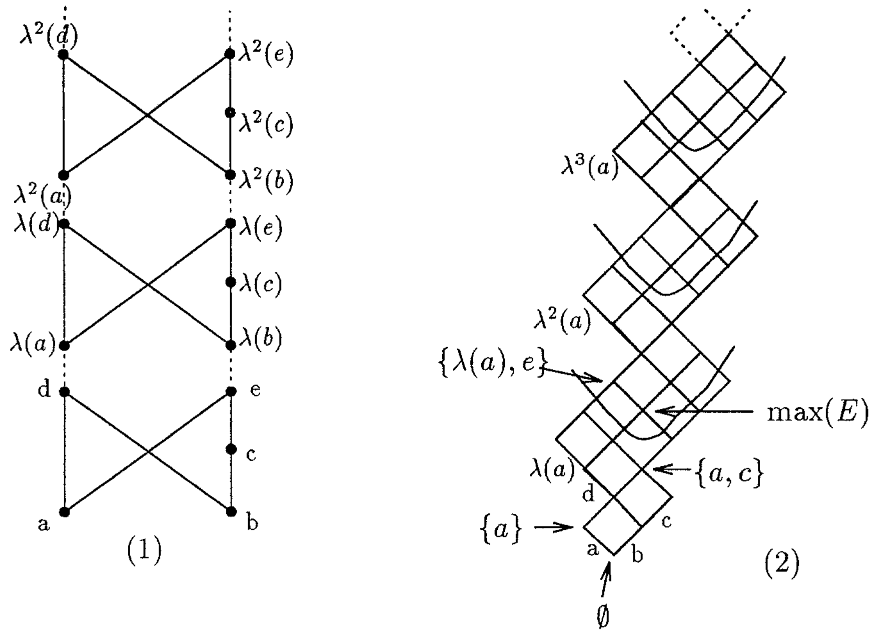

Fig. 2. Notations for regular executions

4.2 Well-synchronized Executions

For a given event e, the number of events that are incomparable with e in an infinite execution can be *a priori* infinite, and so $\mu(e) = \omega$. It is however interesting to see if there exists infinite regular executions for which all events have finite measures.

We will show (Lemma 7) that this property is related to the communication scheme of the basic pattern. We first introduce the notion of *communication graph* of a distributed order.

Definition 6. We call communication graph of a distributed order Θ the quotient of its Hasse diagram by the equivalence relation induced by π^{-1}, i.e. the directed graph $Q(\Theta) = \langle V, C\rangle$ where:

- $V = \pi(E)$
- $C = \{(\alpha,\beta) \in V \times V \mid \exists e \in \pi^{-1}(\{\alpha\}), f \in \pi^{-1}(\{\beta\}),\ e \prec\!\!\!\cdot\, f\}$

A distributed order Θ whose associated communication graph is strongly connected is said to be well-synchronized. The diameter of $Q(\Theta)$ is then denoted by $k_\Theta - 1$.

Note that $k_\Theta \in [1, N]$, N being the number of processes.

Lemma 7. *Let Θ be a finite distributed order. The concurrency measure μ_{Θ_ω} of any event in $\Theta_\omega = \langle E_\omega, \preceq_\omega, P, \pi_\omega\rangle$ is finite if and only if Θ is well-synchronized.*

Proof. $\exists e \in E_\omega, \mu_{\Theta_\omega}(e) = \omega \iff \exists e \in E_\omega, |C_{\Theta_\omega}(e)| = \omega \iff$ (P is finite) $\exists e \in E_\omega, \alpha \in \pi(E_\omega), |C_{\Theta_\omega}(e) \cap \pi^{-1}(\{\alpha\})| = \omega \iff$ (Θ_ω has no infinitely decreasing chain) $\exists e \in E_\omega, \alpha \in \pi(E_\omega), (\forall f \in \pi^{-1}(\{\alpha\}), e \not\preceq f) \iff$ (Lemma 5) $\exists \alpha, \beta \in \pi(E_\omega), \forall f \in \pi^{-1}(\{\alpha\}), e \in \pi^{-1}(\{\beta\}), e \not\preceq f) \iff$ (Def. 4) $\mathcal{Q}(\Theta)$ is not strongly connected. □

4.3 Boundedness and Regularity of the Measure

A property stronger than Lemma 7 can be immediately shown: for a well-synchronized regular execution, the measure of the events is not only finite but also bounded and regular.

Lemma 8. *Let Θ_p be a regular well-synchronized distributed order.*

1. $\forall i \in [0, p[, \forall e \in \lambda^i(E), \quad C_{\Theta_p}(e) \subseteq \bigcup_{j=\max(0, i-k_\Theta+1)}^{\min(p-1, i+k_\Theta-1)} \lambda^j(E)$
2. $\forall i \in [k_\Theta - 1, p - k_\Theta[, \ \forall e \in \lambda^i(E), \ C_{\Theta_p}(\lambda(e)) = \lambda(C_{\Theta_p}(e))$

Proof. 1. Let $i \in [0, p[$, $e \in \lambda^i(E)$, $j \in [i + k_\Theta, p[$, and $f \in \lambda^j(E)$. From Def. 6, since the diameter of $\mathcal{Q}(\Theta)$ is $k_\Theta - 1$: $\exists e' \in \lambda^{i+1}(E), \pi(e') = \pi(e)$ and $\exists f' \in \lambda^{i+k_\Theta-1}(E), \pi(f') = \pi(f)$ such that $e' \preceq f'$. From Def. 4, $e \preceq e'$ and $f' \preceq f$, hence e and f are comparable. Similarly, $\forall j$, $0 \le j \le i - k_\Theta$, $\forall f \in \lambda^j(E)$, $f \preceq e$.
2. Routine application of Lemmas 8(1) and 5. □

The measure clearly is bounded: $\forall e \in E$, $\mu(e) \le |\mathcal{A}(\Theta_{2k_\Theta-1})|$. By Lemma 5, it is also regular: $\forall i, j \in [k_\Theta - 1, p - k_\Theta + 1[$, $\forall e \in E$: $\mu_{\Theta_p}(\lambda^i(e)) = \mu_{\Theta_p}(\lambda^j(e))$.

Moreover, μ can be computed on $\mathcal{A}(\Theta_{2k_\Theta-1})$ (even in the case of infinite executions), thus taking at most $2N - 1$ patterns into account (N being the number of processes). For infinite or very long regular well-synchronized executions whose basic patterns have reasonable sizes, μ can be realistically computed.

We have implemented the computation of μ in our distributed environment [12] based on the Estelle specification language. This environment provides a mechanism of vectorial clocks [15], that are traced "on line". These traces are used as input for our algorithm of construction of the antichain lattice of an order [7, 13]. When given as input a linear extension of $\Theta_{2k_\Theta-1}$, this algorithm has a time complexity of $\mathcal{O}(|\mathcal{A}(\Theta_{2k_\Theta-1})| + |\Gamma(\Theta_{2k_\Theta-1})| + N \times |\Theta_{2k_\Theta-1}|^2)$.

The fact that μ reflects regularity makes useful the definition of a measure μ_∞ on the basic pattern:

Definition 9. Let Θ be a finite, well-synchronized distributed order, we write:

$$\forall e \in E, \quad \mu_\infty(e) = \mu_{\Theta_\omega}(\lambda^{k_\Theta-1}(e))$$

As clearly, $\forall e \in E$, $\mu_\infty(e) = \mu_{\Theta_{2k_\Theta-1}}(\lambda^{k_\Theta-1}(e))$, μ_∞ can also be computed on $\mathcal{A}(\Theta_{2k_\Theta-1})$.

We also have a stronger result on μ_∞: it can be computed on a subgraph of the labeled Hasse diagram $\mathcal{G}(\Theta_{k_\Theta})$ of $\mathcal{A}(\Theta_{k_\Theta})$, that is, by only taking account of at most

N repetitions of the basic pattern [4]. This result does not significantly improve the complexity of the computation of μ_∞ (which becomes in N instead of $2N-1$) but enlights the regular structure of the antichain lattice. We briefly present the way to obtain this result, without any proofs nor algorithms (see [3] for details).

4.4 Computation of μ_∞

Let Θ_ω be a regular well-synchronized distributed order.

Regularity. We show in [3] that the labeled Hasse diagram $\mathcal{G}(\Theta_\omega)$ of its antichain lattice $\mathcal{A}(\Theta_\omega)$ is regular.

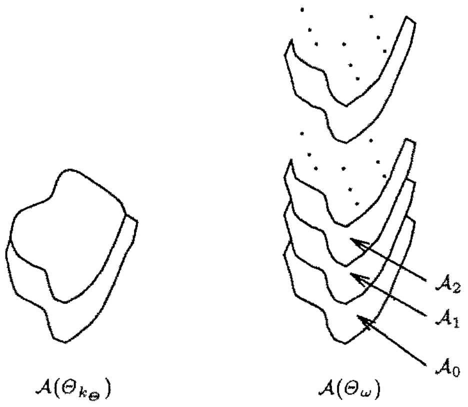
$\mathcal{A}_2$
$\mathcal{A}_1$
$\mathcal{A}_0$
$\mathcal{A}(\Theta_{k_\Theta})$
$\mathcal{A}(\Theta_\omega)$

Fig. 3. Hasse diagram of $\mathcal{A}(\Theta_\omega)$

First, we define a partition $(\mathcal{A}_i)_{i\in\mathbb{N}}$ of $\mathcal{A}(\Theta_\omega)$ where $\mathcal{A}_0$ denotes the set $\{A \in \mathcal{A}(\Theta_\omega) \mid A \not\sqsubseteq \max(E)\}$ and for any $i \in \mathbb{N}^+$, $\mathcal{A}_i = \{A \in \mathcal{A}(\Theta_\omega) \mid \lambda^{i-1}(\max(E)) \sqsubseteq A \wedge \lambda^i(\max(E)) \not\sqsubseteq A\}$ (see Fig. 3). Similarly, we define a partition $(\Gamma_i)_{i\in\mathbb{N}}$ of $\Gamma(\Theta_\omega)$: for any $i \in \mathbb{N}$, Γ_i denotes the set $\{(A,e,B) \in \Gamma(\Theta_\omega) \,|\, A \in \mathcal{A}_i\}$. Finally, we denote by $\mathcal{G}_0$ the subgraph of $\mathcal{G}(\Theta_\omega)$ whose set of vertices is $\mathcal{A}_0 \cup \{B \in \mathcal{A}_1 \mid B \cap \max(E) \neq \emptyset\}$ and whose set of edges is Γ_0.

Lemma 10. *For any $i \in \mathbb{N}$,*

1. *There exists is a one-to-one mapping Λ_i of $\mathcal{A}_0$ onto $\mathcal{A}_i$.*
2. *There exists a one-to-one mapping $\tilde{\Lambda}_i$ of Γ_0 onto Γ_i such that $\forall (A,e,B) \in \Gamma_0$, $\tilde{\Lambda}_i(A,e,B) = (\Lambda_i(A), \lambda^i(e), \Lambda_i(B))$.*

[4] Clearly, μ_∞ cannot be computed on $\mathcal{G}(\Theta_p)$ where $0 < p < k_\Theta$.

Clearly, the set of vertices of $\mathcal{G}_0$ is included in $\cup_{i\in[0,k_\Theta[}\lambda^i(E)$. Hence $\mathcal{G}_0$ is a subgraph of $\mathcal{G}(\Theta_{k_\Theta})$. Then Lemma 10 shows that all the information about $\mathcal{G}(\Theta_\omega)$ is contained into the subgraph $\mathcal{G}_0$ of $\mathcal{G}(\Theta_{k_\Theta})$. Consequently, we can compute μ_∞ by only taking $\mathcal{G}_0$ into account.

Computation. In [3] we show how to compute μ_∞ on $\mathcal{G}_0$:

Proposition 11. $\forall e \in E$,

$$\mu_\infty(e) = |\{(A, f, B) \in \Gamma_0 \mid \exists i \in \mathbb{N}, f = \lambda^i(e)\}|$$

This proposition gives us a new algorithm for the computation of μ_∞. It relies on the computation of the subgraph $\mathcal{G}_0$ of $\mathcal{G}(\Theta_{k_\Theta})$ instead of the computation of $\mathcal{G}(\Theta_{2k_\Theta-1})$ (see subsection 4.3). Its time complexity is the same as for the computation of $\mathcal{G}(\Theta_{k_\Theta})$, that is $\mathcal{O}(|\mathcal{A}(\Theta_{k_\Theta})| + |\Gamma(\Theta_{k_\Theta})| + N \times |\Theta_{k_\Theta}|^2)$ [7, 13] where $k_\Theta \leq N$ (recall N is the number of processes).

5 Comparison with Other Measures

In this section, we present some recently proposed concurrency measures. We show that ours is comparable with each of them, especially if we study the behaviour in case of infinite regular executions.

At first, we present two "global" measures that give a single value to quantify a whole execution. In our context (see section 6), we are not interested in such an approach for it does not enable the detection of bottlenecks (although it can be suitable for other distributed programming problems). However, note that a "local" measure (that associates a value with each event of an execution) can be derived from a global one. The following measures are local ones. They are computed directly on the order of an execution, not on the antichain lattice; the advantage is a better complexity (polynomial in the number of events), but the drawback is that they are less accurate, and not suited to infinite well-synchronized regular executions.

Charron-Bost [5] As far as we know, this is the first attempt to take into account, for a concurrency measure, the causal structure of an execution. In our notation, this measure is:

$$m(\Theta) = \frac{|\mathcal{A}(\Theta)| - |E| - 1}{c(\Theta) - |E| - 1}$$

where $c(\Theta)$ is the number of antichains in a totally concurrent execution (i.e. any two events with distinct labels are incomparable) with as many processes and as many events per process as in Θ, that is to say $c(\Theta) = \prod_{i=1,N}(|\pi^{-1}(\{i\})| + 1)$. In fact, μ is quite similar to m: the idea is to count the number of antichains in an execution. The difference is that μ is a global measure and that it is normalized: it ranges from 0 in the worst case ($|E| + 1$ is the number of antichains of a totally sequential execution) to 1 in the best (a totally concurrent execution). But this normalization is unsuited to the infinite case: $c(\Theta_p) = \prod_{i=1,N}(p|\pi^{-1}(\{i\})| + 1) = p^N \prod_{i=1,N}(|\pi^{-1}(\{i\})| + 1/p)$, hence for a well-synchronized execution:

$$m(\Theta_p) = \frac{(p - k_\Theta + 1)\,|\mathcal{A}_0| + |\mathcal{A}(\Theta_{k_\Theta}) \setminus \mathcal{A}_0| - p|E| - 1}{p^N \prod_{i=1,N}(|\pi^{-1}(\{i\})| + 1/p) - p|E| - 1} \Rightarrow m(\Theta_\omega) = 0$$

Habib et al. [11] They propose a "worst-case-measure" of the concurrency of an execution: the minimal size of the maximal (for inclusion) antichains of its associated poset. It can be seen as the number of processors that can proceed during the worst bottleneck of the execution.
This measure could be made local by computing for each event e the minimal size of the maximal antichains that contain e. As this measure depends on the antichains of a poset, it is obvious from our framework and results that such a derived measure presents the same regularity properties as ours. The difference with our measure is that we consider all the antichains which contain an event, whereas they only consider the maximal antichains.

Fidge [9] Fidge proposes a local measure, that he extends to a global one. The measure β of an event e in an execution Θ is defined as follows:

$$\beta(e) = \frac{|\downarrow e| - 1 - h(e)}{|\downarrow e| - 1 - a}$$

(Fidge proposes two closely related measures, whether $a = 1$ or $a = 1/N$). $h(e)$ is the "height" of e, i.e. the length of the longest chain ending by e. If we consider a well-synchronized regular execution Θ_ω, we can easily prove that β converges:

$$\forall e,\ \lim_{p \to \infty} \beta(\lambda^p(e)) = \frac{|E| - h(E)}{|E|}$$

We obtain a finite and computable measure. But it does not preserve regularity and for infinite executions, the measure is identical for almost all events.

Raynal et al. [16] This measure is a variant of Fidge's one:

$$\alpha(e) = \frac{|\downarrow e| - 1 - h(e)}{v(\downarrow e) - 1 - h(e)}$$

where $v(\downarrow e)$ is defined as the "volume" of the causal past of e, i.e. $v(\downarrow e) = \sum_{i=1}^{N}(h(e_i) + 1)$ with $e_{\pi(e)} = e$ and for all $i \neq \pi(e)$, e_i is the maximum of the predecessors of e in $\pi^{-1}(\{i\})$. For well-synchronized executions, α converges as well:

$$\lim_{p \to \infty} \alpha(\lambda^p(e)) = \frac{|E| - h(E)}{v(E) - h(E)}$$

We have computed these measures on a kernel of the Jacobi algorithm, automatically distributed on five processors. The results are presented in Fig. 4.

The values for μ must be read on the right vertical axis, and the values for the three other measures (α, β, m) on the left axis: from 0 to 1 because these measures are normalized. Therefore, for comparison sake, the exact values are not to be taken into account, but only the variations of the graph. Charron-Bost's measure m is also presented on this graph, but the plots are in fact the values $m(\Theta_p)$, Θ being the pattern observed on this execution.

This diagram clearly shows that our measure remains relevant when time flows, whereas the others converge to a single value.

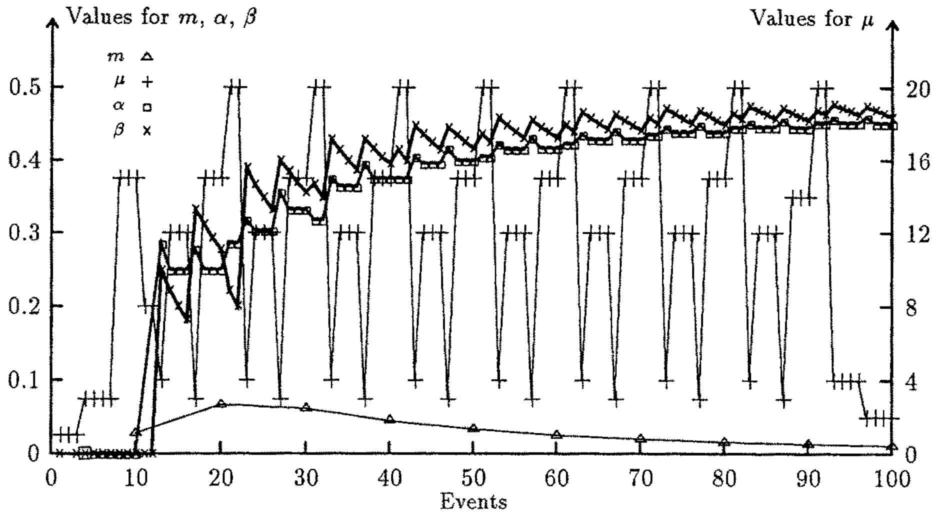

Fig. 4. Measures for the Jacobi algorithm.

6 Use in Automated Distribution of Sequential Programs

6.1 Motivation

Much research is being done on efficient sequential code distribution techniques (see for instance [17]). Our work originates with the practical problem of evaluating automatically distributed programs.

To automatically parallelize a sequential program for a distributed memory parallel computer, compiling directives must be given. For a data-driven distribution technique, the key directive is to specify a data distribution, that is, to indicate how data structures are to be decomposed and mapped onto the network of processes.

A programmer needs tools that help him to select a good distribution of the data structures of a source program. That is, he must be able to evaluate quantitatively and qualitatively the executions of the distributed code that can be generated given a data decomposition. For instance, he needs to determine the fragments of the source code for which a data decomposition is unsuitable. For this, a tool that only measures the average degree of concurrency of a distributed execution is clearly inadequate.

To be efficient enough the tools to be designed should be able to produce relevant outcomes without having to entirely run a generated parallel program. In other words, tools are needed that can collect as much relevant information as possible by efficient static analyses [8] of a source code and of an associated data decomposition.

Semi-automatic distribution is used in application fields (scientific computing) where source codes are generally composed of loops operating on arrays [10]. In fact, available compilers are inefficient when a source program is not regular.

Applying a distribution technique does not affect syntactic regularity. First, generated codes are SPMD (Single Program Multiple Data) [4]: the control structure of each generated process is a copy of that of the source program. Second, data distribution rules are regular as well: arrays, for instance, are decomposed into blocks of contiguous rows or blocks of contiguous columns.

6.2 Detection of Regular Well-synchronized Executions

We deal with programs that are mainly composed of loops operating on arrays. Data distribution is expressed by a distribution function that associates with each array element the processor on which it is located, called its owner ([1]).

From an intuitive point of view, it is clear that the loops of a parallelized program may lead to regular executions, i.e. the finite repetition of the same distributed order. The idea is to consider as the "basic pattern" the distributed order corresponding to the execution of one step of the loop. Then, an execution is actually regular (in the sense of section 4) if *each* step of the loop produces this pattern.

There is repetition of a pattern from one step of the a loop to another if the same events are observed on each processor, as well as the same comparabilities between events. To check this, we have to look at each array reference. For all values of the loop index, this reference must correspond to array elements that are owned by the same processor. Two possibilities arise: the index is not *syntactically* used in the reference, (for instance the external loop of the Jacobi relaxation algorithm), or it is syntactically used but not *semantically* (its value has no incidence on the result of the determination of the owners).

Checking this is not statically possible in the general case. However, in practice, the distribution function and the expressions in array references are often affine. Therefore compile-time checking is possible in some cases.

Considering a regular execution, it is easy to detect if it is well-synchronized or not: it suffices to execute one step of the loop, build the communication graph, and check its connexity.

We have not made an exhaustive study of benchmark programs, but we have found programs whose runtime behaviours are regular and well-synchronized. For example, Jacobi-like programs (walking n times through a matrix, updating each time the values by a function of the neighboring values) satisfy this property whatever the data distribution is, and many linear algebra programs also satisfy it but only for particular data distributions.

7 Conclusion

The contribution we have presented in this paper originates with the practical problem of evaluating the synchronizations of a distributed program running on a network of processors. We are faced with such a problem in the field of automatic parallelization of sequential programs for distributed memory computers (high performance computing).

In this field the generated programs are weakly deterministic [2] and are often control static ones, for which studying one particular execution of a program gives information on the exact quantity of parallelism extracted by the compiler/parallelizer. Another salient feature of the run-time behaviours of these programs is their regularity.

Consecutively, we have been interested in a concurrency measure that would take regularity into account. We have defined a measure that associates a value with each event of an execution. In the case of a regular and well-synchronized execution, this value remains bounded even if the execution is infinite. This is not the case with other measures in the literature, that ultimately associate the same value with all events that occur, although some of them could be extended to take the well-synchronization into account as we have done for that of Charron-Bost. Our measure is therefore relevant whatever the length of the execution is and can be computed from the basic pattern of the execution by taking at most N repetitions of this pattern into account, where N is the number of processors.

To obtain this result, we used partial order theory. A distributed execution is modeled as the causality partial order between events. The degree of synchronization is captured by counting the number of antichains that contain a given event. This theory has proved useful, providing us with an adequate framework to describe the regularity of an execution.

The computation of our measure has been integrated in a parallelization environment developed in our research team [1]. Its exploitation is at the planning stage.

Acknowledgments

This work was partially supported by the project Trace of the French Ministère de l'Enseignement Supérieur et de la Recherche.

References

1. F. André, O. Chéron, and J-L. Pazat. Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 293–308, Elsevier Science Publishers B.V., 1993.
2. C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. Correctness of automated distribution of sequential programs. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, pages 517–528, LNCS 694, Springer Verlag, June 1993.
3. C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. *Measuring Concurrency of Regular Distributed Computations*. Research Report 882, Irisa, Rennes, France, October 1994.
4. D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.
5. B. Charron–Bost. Combinatorics and Geometry of Consistent Cuts : Application to Concurrency Theory. In Bernard and Raynal, editors, *Int. Workshop on Parallel and Distributed Algorithms*, pages 45–56, Springer Verlag, Nice, France, 1989.
6. B.A. Davey and Priestley H.A. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

7. C. Diehl, C. Jard, and J.X. Rampon. Reachability analysis on distributed executions. In JP. Jouannaud MC. Gaudel, editor, *Proc. TAPSOFT,93 LNCS 668*, pages 629–643, Springer–Verlag, Orsay, Paris, April 1993.
8. T. Fahringer, R. Blasko, and H.P. Zima. Automatic Performance Prediction to Support Parallelization of Fortran Programs for Massively Parallel Systems. In *Proc. of the '92 International Conference on Supercomputing*, pages 347–356, ACM press, July 1992.
9. C.J. Fidge. A simple run–time concurrency measure. In *Proceedings of the 3rd Australian Transputer and OCCAM User Group Conference*, pages 92–101, 1990.
10. G.H. Golub and C.F. Van Loan. *Matrix computations.* The Johns Hopkins University Press, second edition, 1990.
11. M. Habib, M. Morvan, and J.X. Rampon. Remarks on some concurrency measures. In *Graph–Theoretic Concepts in Computer Science*, pages 221–238, LNCS 484, june 1990.
12. C. Jard and J.-M. Jézéquel. ECHIDNA, an Estelle-compiler to prototype protocols on distributed computers. *Concurrency Practice and Experience*, 4(5):377–397, 1992.
13. C. Jard, G.V. Jourdan, and J.X. Rampon. *Some On–Lines Computations of the Ideal Lattice of Posets.* Research Report 773, IRISA, December 1993.
14. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
15. F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *Proc. Int. Workshop on Parallel and Distributed Algorithms Bonas, France, Oct. 1988*, North Holland, 1989.
16. M. Raynal, M. Mizuno, and M.-L. Neilsen. A synchronization and concurrency measure for distributed computations. In *12th IEEE Int. Conf. on Distributed Computing Systems*, pages 657–664, Yokokama, June 1992.
17. H. Sips. 4th Int. Workshop on Compilers for Parallel Computers. Sips, H. Editor. Delft, 13-16 december. Delft University of Technology, 1993.