Loop-based Modeling of Parallel Communication Traces

Alain Ketterlin, Matthieu Kuhn, Stéphane Genaud, Philippe Clauss

Loop-based Modeling of Parallel Communication Traces

Alain Ketterlin, Matthieu Kuhn, Stéphane Genaud, Philippe Clauss

RESEARCH REPORT
N° 8562
Juillet 2014
Project-Team Camus

ISSN 0249-6399   ISRN INRIA/RR--8562--FR+ENG

Loop-based Modeling of Parallel Communication Traces

Alain Ketterlin, Matthieu Kuhn, Stéphane Genaud, Philippe Clauss

Project-Team Camus

Research Report n° 8562 — Juillet 2014 — 12 pages

Abstract: This paper describes an algorithm that takes a trace of a distributed program and builds a model of all communications of the program. The model is a set of nested loops representing repeated patterns. Loop bodies collect events representing communication actions performed by the various processes, like sending or receiving messages, and participating in collective operations. The model can be used for compact visualization of full executions, for program understanding and debugging, and also for building statistical analyzes of various quantitative aspects of the program's behavior.
The construction of the communication model is performed in two phases. First, a local model is built for each process, capturing local regularities; this phase is incremental and fast, and can be done on-line, during the execution. The second phase is a reduction process that collects, aligns, and finally merges all local models into a global, system-wide model. This global model is a compact representation of all communications of the original program, capturing patterns across groups of processes. It can be visualized directly and, because it takes the form of a sequence of loop nests, can be used to replay the original program's communication actions.
Because the model is based on communication events only, it completely ignores other quantitative aspects like timestamps or messages sizes. Including such data would in most case break regularities, reducing the usefulness of trace-based modeling. Instead, the paper shows how one can efficiently access quantitative data kept in the original trace(s), by annotating the model and compiling data scanners automatically.

Key-words: Loop-based modeling, parallel traces, MPI communication

Modélisation à base de boucles des traces de communication parallèles

Résumé : Ce rapport de recherche décrit un algorithme qui prend en entrée la trace d'un programme distribué, et construit un modèle de l'ensemble des communications du programme. Le modèle prend la forme d'un ensemble de boucles imbriquées représentant la répétition de motifs de communication. Le corps des boucles regroupe des événements représentant les actions de communication réalisées par les différents processus impliqués, tels que l'envoi et la réception de messages, ou encore la participation à des opérations collectives. Le modèle peut servir à la visualisation compact d'exécutions complètes, à la compréhension de programme et au *debugging*, mais également à la construction d'analyses statistiques de divers aspects quantitatifs du comportement du programme.

La construction du modèle de communication s'effectue en deux phases. Premièrement, un modèle local est construit au sein de chaque processus, capturant les régularités locales ; cette phase est incrémentale et rapide, et peut être réalisée au cours de l'exécution. La seconde phase est un processus de réduction qui rassemble, aligne, et finalement fusionne tous les modèles locaux en un modèle global décrivant la totalité du système. Ce modèle global est une représentation compacte de toutes les communications du programme original, représentant des motifs de communication entre groupes de processus. Il peut être visualisé directement et, puisqu'il prend la forme d'un ensemble de nids de boucles, peut servir à rejouer les opérations de communication du programme initial.

Puisque le modèle construit se base uniquement sur les opérations de communication, il ignore complètement d'autres données quantitatives, telles que les informations chronologiques, ou les tailles de messages. L'inclusion de telles données briserait dans la plupart des cas les régularités topologiques, réduisant l'efficacité de la modélisation par boucles. Nous préférons, dans ce rapport, montrer comment, grâce au modèle construit, il est possible d'accéder efficacement aux données quantitatives si celles-ci sont conservées dans les traces individuelles, en annotant le modèle et en l'utilisant pour compiler automatiquement des programmes d'accès aux données.

Mots-clés : Modélisation par boucles, traces parallèles, communication MPI

Loop-based Modeling of Parallel Communication Traces

Alain Ketterlin*†, Matthieu Kuhn†, Stéphane Genaud† and Philippe Clauss*†
*INRIA (Camus team), Nancy Grand-Est (France)
†ICube research center, Université de Strasbourg (France)
Email: {ketterlin,kuhn,genaud,clauss}@unistra.fr

***Abstract*—This paper describes an algorithm that takes a trace of a distributed program and builds a model of all communications of the program. The model is a set of nested loops representing repeated patterns. Loop bodies collect events representing communication actions performed by the various processes, like sending or receiving messages, and participating in collective operations. The model can be used for compact visualization of full executions, for program understanding and debugging, and also for building statistical analyzes of various quantitative aspects of the program's behavior.**

The construction of the communication model is performed in two phases. First, a local model is built for each process, capturing local regularities; this phase is incremental and fast, and can be done on-line, during the execution. The second phase is a reduction process that collects, aligns, and finally merges all local models into a global, system-wide model. This global model is a compact representation of all communications of the original program, capturing patterns across groups of processes. It can be visualized directly and, because it takes the form of a sequence of loop nests, can be used to replay the original program's communication actions.

Because the model is based on communication events only, it completely ignores other quantitative aspects like timestamps or messages sizes. Including such data would in most case break regularities, reducing the usefulness of trace-based modeling. Instead, the paper shows how one can efficiently access quantitative data kept in the original trace(s), by annotating the model and compiling data scanners automatically.

I. Introduction

The ever-growing scale and complexity of parallel systems, that will commonly have thousands to hundreds of thousands of cores will enable to run more and more massively parallel applications. In order to design, develop and tune such applications, a large variety of tools is required. The code tuning phase requires tools to monitor and analyze the program behavior on an actual execution platform. We can distinguish between two complementary approaches for this task: profiling and tracing. Profiling aims to record aggregated data that reflects specific aspects of the program behavior. It is generally designed to have the lowest possible intrusiveness in order to collect accurate measurements. In contrast, tracing addresses the need to record all communication events to understand how the concurrent processes interact all along the program execution. Tracing is challenging because it involves huge amounts of records.

Post-mortem examination of those records reveals how each process behaved, and shows the causes of a potential loss of performance. In this paper, we focus on how the numerous time-based events contained in communication traces can be translated to some higher-level information which summarizes the communication scheme of an application. This approach has also been tried through profiling. For example [1] use clustering on data collected by processor performance counters to identify groups of processes with similar behavior, or computation phases with similar characteristics. However, communication traces are central to the analysis and understanding of message-passing parallel programs. The need to understand the program at different levels of details is witnessed by the compelling usage of visualization tools like Jumpshot [2]. Unfortunately, the exhaustive log of events displayed by visualization tools often yields complicated and varied communication patterns, which obfuscates the essential logic of the program.

The work presented here has three main motivations. The first is to adopt a strictly logical (or topological) point of view for trace analysis. The goal is to obtain a high-level view of the trace, in terms of communication patterns, rather than a detailed, timing based graphical listing of individual events. This entails some trade-offs, favoring regularity and simplicity against precision and completeness, which we do not ignore but try to solve with complementary techniques. Our primary goal is to help understanding (and maybe debugging) distributed programs by abstracting away low-level details, and rather focus on obtaining an overview of the general architecture of the application, from which further analysis can be planned.

Our second motivation is our will to define a formal model of the communication behavior of a distributed program, that goes beyond the compact archiving of traces, or the ability to replay the program. A long term goal of this work is to use the results of trace analysis for program transformations, for instance to suggest new communication architectures, or new data distributions. We are still far from that, but this motivation explains our choice of explicit loop nests to represent regular communication patterns. This model and general approach has been successful in compiler optimizations [3], and we have the intuition that it can be useful also for large distributed program optimization.

Our third motivation is the conviction that a compact, formally analyzable model of a trace is of great help for further, detailed studies of the behavior of the program. We do not think than focusing on high-level communication patterns is orthogonal to detailed performance analysis. Actually, we are convinced that a global view can help a lot in focusing specific measurements, and drive the analyst in detecting performance bugs and locate optimization opportunities.

Along these lines, this paper makes the following contributions:

- define a formal model based on loop nests to represent parallel communication traces;
- describe a modeling algorithm based on loop transformations that merges intra-node traces into a global, program-wide trace model;
- explain how the formal, high-level model of a trace can help drive quantitative, statistical analysis of trace data.

The next section reviews background material that is at the foundation of our work. Section III goes into the details of our modeling algorithm. Trace models produced by the algorithm can be visualized directly: this is explained and illustrated in Section IV. The models can also be used to help further, time-based analysis of the raw trace: an example is given in Section V. Related work is reviewed before the conclusion.

II. Background

A. Modeling Architecture

The modeling architecture we have used is depicted on Figure 1, with references to the relevant sections of the text. During a distributed program execution, every process produces an individual trace containing a stream of events. The trace is analyzed locally to produce a local model made of loops and atomic events. At the end of the execution, local models are progressively merged to form a global model, which is the final result of the analysis. We make no assumption on whether local models are built on the execution nodes or not, and on whether all local models are collected on a single node before merging or merged during a distributed reduction. Such an architecture is very common in tracing software [4]. In practice we use the TAU tracing infrastructure [5] to obtain individual traces.

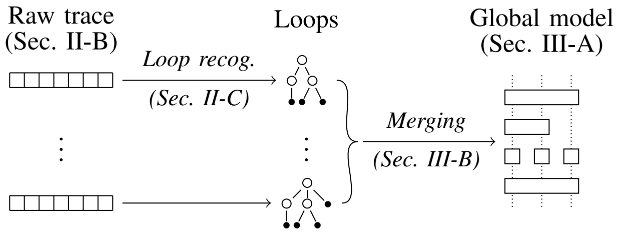


Fig. 1. The modeling architecture

B. Trace Format

Individual traces collect events representing all communication actions performed by the corresponding process. There are four types of events:

- <u>src</u> send <u>dst</u> <u>tag</u> is the basic message sending action: <u>src</u> and <u>dst</u> are the ranks of the source and destination processes involved, and <u>tag</u> corresponds the MPI-tags, designating a specific channel between the two processes. Channels (defined by triplets including two process identifiers and a tag) are assumed to respect a FIFO discipline;
- <u>src</u> recv <u>dst</u> <u>tag</u> represents the sending of a message (the order of source and destination process identifiers may seem unexpected on first read, we have favored consistency with send);
- <u>proc</u> sync <u>name</u> <u>group</u> indicates that process <u>proc</u> takes part in a collective action named <u>name</u> involving all processes in <u>group</u> (a list of process identifiers, essentially corresponding to an MPI communicator). All examples in this paper use a group comprising all processes of the program, but any subset could appear in a sync event.
- <u>proc</u> local <u>desc...</u> denotes an event that is strictly local to process <u>proc</u>: it doesn't involve any kind of communication, and is here only to let one include descriptive events in a trace (<u>desc...</u> can be any sequence of strings and numbers).

A final assumption is that a collection of individual traces actually represents a program execution: all messages sent are received, all processes participating in a collective operation emit the corresponding event, etc.

The notions of tag and process groups correspond to MPI's tags and communicators [6]. However, it is important to realize that traces are abstract representations of actual events, not a faithful representation of MPI program executions. In particular, send and recv events represent the actual processing of a message by the program, independent on the MPI call(s) that triggered them. Specific semantics, like those of asynchronous operations in MPI, have no direct translation in the trace, and must be represented explicitly with the help of local events. For instance, here is an excerpt from the trace of process 0 in an execution of the BT program from the NAS Parallel Benchmark suite (NP) [7]. program with four processes:

```
0 local call MPI_Isend
0 send 1 tag3000
0 local return MPI_Isend
0 local call MPI_Irecv
0 local return MPI_Irecv
0 local call MPI_Wait
0 local return MPI_Wait
0 local call MPI_Wait
1 recv 0 tag3003
0 local return MPI_Wait
```

Here local events represent entry into and exit from MPI calls, but have absolutely no meaning for the algorithms that are presented below: they will be kept throughout the modeling phase, essentially as comments, and their interpretation is left to the client application. Figure 2 shows a fragment of a timeline displaying messages along with MPI calls (end of calls are represented with a closing bracket). This requires specific processing of local events. It is the only place in this

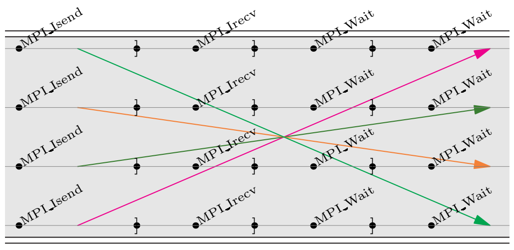


Fig. 2. Displaying MPI calls and messages

paper where they are used with particular semantics.

C. Nested Loop Recognition

The NLR algorithm [8] detects repetitions in linear traces of records. A record is a vector whose elements can be numbers or symbols. NLR proceeds by looking for short, 3-fold repetitions of arbitrary sequences of values, that it replaces immediately with a loop of the form:

```
for i=1 to 3
  <sequence>
done
```

It also searches for an occurrence of a loop followed a copy of its body, in which case it increments the loop upper bound, and removes the redundant sequence. Explicitly building syntactic terms makes it easy for NLR to recognize repetition of structured terms, and build loops nested to an arbitrary depth. An example of NLR output appears in Figure 3: the input trace is that of process 0 of NPB LU class C with 16 processes. Output includes loops (`for...done`) and events (`val`).

```
...
0 sync MPI_Allreduce 0-15
0 send 1 tag2
1 recv 0 tag1
0 send 4 tag4
4 recv 0 tag3
0 sync MPI_Allreduce 0-15
0 sync MPI_Barrier 0-15
for i0 = 1 to 249
  for i1 = 1 to 160
    0 send 1 tag2
    0 send 4 tag4
  done
  for i1 = 1 to 160
    1 recv 0 tag1
    4 recv 0 tag3
  done
  0 send 1 tag2
  1 recv 0 tag1
  0 send 4 tag4
  4 recv 0 tag3
done
...
```

Fig. 3. Individual trace produced by NLR (NPB LU, class C, 16 processes, process 0).

NLR has several interesting properties in the context of trace modeling. First, it is a greedy algorithm, and as such may produce a sub-optimal result, but it is very fast, catching repetitive patterns quickly, and providing high quality results on our benchmarks: some performance indicators appear in Table I. NLR process 80,000 events per second in the worst case, and often much more. The resulting trace takes a few kilobytes in text format (as used in Figure 3) and less than a kilobyte when compressed with `gzip` to remove syntactic redundancies. Second, NLR is fully incremental: it keeps a window of loops and values to which it appends incoming values, and creates or updates loops in his window. Random access to the trace is not required, and memory consumption is negligible. Regarding our profiling task, NLR could reasonably be integrated into the profiling infrastructure, saving I/O time and disk space by performing on-line trace modeling.

There are two more features of NLR that we would like to mention quickly. The first is that NLR not only recognizes repetitions, but is also able to *create* them, by turning numbers inside events and loop bounds into affine functions of the enclosing loop indices. It has thus more expressive power than usual pattern detection algorithms. Unfortunately, we have found no use of this power in our benchmark program traces. The second feature that may become of interest in a middle term is NLR's ability to combine modeling and prediction, letting it emit predictions on forthcoming values at an arbitrary distance. We plan to leverage this ability in future work. NLR is fully described in [8].

III. Merging Individual Models

At the end of execution of a distributed program, all individual traces have been processed with NLR, and what remains is a set of N individual models, made of sequences of loop and events. The merging phase is basically a reduction over this set of models, where the reduction operator somehow combines two intermediate results. This combination relies on a single idea: when two loops based on two distinct sets of processes exchange messages, then, if the amount of messages exchanged has certain properties, the two loops can be replaced by a single loop. Here is the simplest possible example of such a situation, with two process. Each individual trace is modeled by a single loop:

```
// Process 0          // Process 1
for i = 1 to 10       for i = 1 to 10
  0 send 1 t            0 recv 1 t
done                  done
```

Here, these two models can be merged into a single loop, covering both processes, and capturing the message pattern:

```
// Process 0 & 1
for i = 1 to 10
  0 send 1 t
  0 recv 1 t
done
```

What appears on this trivial example is the fact that we will build loops that *cover* multiple processes (the body of the loop contains events that originate in different processes). The rest of this section explains the details of this merging process.

Program [/Class] /#Proc.	Events per trace (min[–max])	Processing time (s) (average)	Processing rate (ev./s) (average)	Trace size as text (bytes) (min-max)	Trace size gzipped (bytes) (min-max)
`bt/C/16`	9671	0.043502	224486	3197–3377	480–497
`bt/C/64`	19319	0.246827	79318	5953–6293	750–793
`cg/C/16`	27971	0.033502	842150	1961–2089	225–238
`cg/C/32`	39979	0.052253	769207	2449–2633	254–264
`lu/C/16`	161682–323338	0.332770	796646	1362–2245	248–313
`lu/C/64`	161682–323338	0.411963	714833	1362–2283	249–316
`mg/C/16`	6240–6744	0.026751	281726	3883–6181	325–384
`mg/C/64`	6324–6828	0.021314	323557	4236–6582	357–478
`sp/C/16`	19269	0.030502	636774	2105–2205	341–362
`sp/C/64`	38517	0.059628	650019	2125–2225	347–373
`sweep3d/8`	12835-19235	0.013000	1306352	937–1162	219–241
`sweep3d/256`	12835–25635	0.020892	1169937	945–1518	221–285

TABLE I
PERFORMANCE OF LOOP RECOGNITION ON INDIVIDUAL TRACES

A. Data Structures

Individual traces are modeled by sequences of loops and events, and bodies of loops are simple sequences of events and other loops. Introducing parallelism inside loops requires more elaborate data structures. In the rest of this section, we will use the term *construct* to designate either a loop or an event. A model of a set of processes, hereafter called a *system* will be a triple (V, T, C) where:

- V is a set of constructs;
- $T \subseteq V \times V$ is a set of (directed) *topological links*: $(v_1, v_2) \in T$ means that there is a process p such that events belonging to p in v_1 directly precede events belonging to p in v_2. Essentially, T represents the chronological order of constructs;
- $C \subseteq V^2$ is a set of (undirected) *communications links*: $(v_1, v_2) \in C$ means that v_1 and v_2 exchange messages.

Individual trace models are simple systems where V is the set of constructs, T includes a pair (v_1, v_2) whenever v_2 follows v_1 in the sequential execution order, and C is usually empty (except when a process sends messages to itself). The body of every loop is itself a system over the events and loops appearing in it.

To characterize communication links, we need the notion of a *flow*: a flow maps *channels* (two process identifiers and a tag) to positive integers representing the number of messages sent or received along this channel. Initially, every construct has a *potential flow*, which gives the number of all messages it exchanges per channel (essentially, “dangling” messages). The potential flow of a construct only counts messages for which no correspondent is know yet. Communication links are also labeled with a flow, describing the total volume of messages actually exchanged along the link. When merging two systems, some potential flow is turned into communication links. When a system covers all processes of a program, constructs have a potential of zero and all messages are counted in communication links.

B. Merging Strategy

Starting with individual trace models, a reduction is performed on the set of systems. We are now going to detail the steps of the reduction operation. We assume in this section that the operation applies to two systems, which are then replaced by the results of merging, even though a single merging operation can operate on any number of systems. But because independent merging steps can be performed in parallel, pairwise merging seems to be the most appropriate strategy. We describe the operation as it applies on two systems covering each an arbitrary number of processes.

Merging two systems $S_1 = (V_1, T_1, C_1)$ and $S_2 = (V_2, T_2, C_2)$ proceeds in several steps. First, a new system $S = (V, T, C)$ is created, with $V = V_1 \cup V_2$, $T = T_1 \cup T_2$,and $C = C_1 \cup C_2$. The next three steps are the following:

1) new communications links between constructs from distinct systems are added to C;
2) existing loops are *regularized*, such that they have a single correspondent per channel;
3) groups of loops are coalesced into a single loop when appropriate.

The rest of this section examines these steps in detail.

1) Communication: When building a new system S from two systems S_1 and S_2, some constructs find correspondents for part of their potential flow. The goal of the communication phase is to turn this potential flow into communication links. Since all channels are FIFO, this is performed by iterating over constructs in topological order, and keeping a list of constructs that have yet unsatisfied communications (called *inflight* constructs). When a construct is processed, its potential flow is intersected with every inflight construct, and links are created accordingly. A simplified algorithm is:

```
inflight = []
for every construct $v$ of $S$ in topological order
    for every construct $f$ in inflight
        if potential($v$) $\cap$ potential($f$) is not empty
            add $(v, f)$ to $C$, and update potentials
    append $v$ to inflight
```

There are various heuristics to keep the inflight list short. A system knows which processes it covers, and constructs can be eliminated from the list as soon as they have no more messages to exchange with other constructs. Also, the topological order used for the outer loop can be chosen so as to minimize the time spent by a construct in the inflight list. All these techniques are fairly straightforward, and will not be detailed further. At the end of this phase, the communication links contain all exchanges between constructs of the system. Topological links are unchanged.

2) Regularization: The goal of the regularization phase is to prepare loops for coalescing. Two loops can be coalesced (and their body merged) if and only if each of them is the exclusive correspondent of the other. That is: for all channels of the flow of the communication link between the two loops, the volume exchanged must be the total amount of communication of the loops along these channels. We have seen an positive example at the beginning of this section. Here is a negative example, where a loop communicates only part of its flow on the `(0,1,t)` channel to its correspondent (which means it has another correspondent, not shown). Communication links are drawn as dotted lines:

```
for i = 1 to 20 .......... for i = 1 to 10
  0 send 1 t     .           0 recv 1 t
done              .        done
                   .       ...
```

We say that a loop is *regular* if and only if, for each of its communication links, the volume on every channel is the total volume of communication sent or received along this channel. We say that a communication link is regular if both endpoints are regular. Note that the use of tags makes this definition slightly more complex than expected. In the following example, the loop on the left is regular:

```
for i = 1 to 10 .......... for i = 1 to 10
  0 send 1 t1    .           0 recv 1 t1
  0 send 1 t2     .        done
done               .       for i = 1 to 10
                             0 recv 1 t2
                           done
```

Here the loop has two correspondents, but it is regular: the link to the first correspondent transmits all its messages along `(0,1,t1)`, while the link to the second correspondent transmits all messages along `(0,1,t2)`.

When a loop is not regular, it means that it has at least one link that consumes fewer messages than the loop produces. In that case, the regularization phase will break the loop into two parts, one that is adjusted to the link, and a remaining part whose flow will be distributed to other correspondents. Assume loop L communicates with construct C with flow f. Note n the number of iterations of L, and b the total flow of its body. Testing whether L is regular amounts to testing whether:

$$\exists h \in f \text{ such that } f[h] < n \cdot b[h]$$

where $f[h]$ denotes the number of messages on channel h in flow f. If such a channel exists, the loop is *split* into two copies, the first one performing

$$k = \min_{h \in f} \left(\left\lfloor \frac{f[h]}{b[h]} \right\rfloor \right)$$

iterations, and the second one performing $n - k$ iterations, unless $n - k = 1$, in which case the loop is *unwrapped* and replaced by a copy of its body. The communication links incident to C are then removed, and the corresponding flow is re-distributed.

A special case of loop splitting occurs when the flow along a communication corresponds to less than one iteration, as in the following example:

```
for i = 1 to 10 .......... 0 recv 1 t
  0 send 1 t     .
  ...             .        ...
  0 send 1 t
  ...
done
```

In that case, the regularization phase tries to "pump" out of the body of the loop just enough flow for the link, preserving the iteration count if possible. This is called *shifting* the loop.

The regularization phase will process every loop, in topological order, splitting and shifting loops when necessary, and updating topological and communication links in the system. Regularization ends when all communication links are regular. Simple events (sends, receives, and collective operations) either have a single correspondent, or none (meaning the corresponding process is not yet covered). Loops may have several correspondents, however the sets of channels of its various communication links are disjoint.

3) Coalescing: The role of regularization is to prepare the way to loop coalescing, where two (or more) loops are turned into a single loop, and their communication links are turned into links between elements of their merged bodies. Basically, a global flow of messages between loops is turned into a per-iteration flow between loop bodies. There are two difficulties to consider.

The first difficulty is arithmetic. Two loops may have a regular communication link between them but have different iteration counts. Here is an example:

```
for i = 1 to 10 .......... for i = 1 to 20
  0 send 1 t                 0 recv 1 t
  ...
  0 send 1 t
  ...
done
```

In such cases, loops have to be "unrolled", or "blocked" before coalescing. With iteration counts n_1 and n_2, the blocking factor is $g = \gcd(n_1, n_2)$. If $g = 1$, both loops are completely unrolled. Otherwise, if g is less than the iteration count for any one loop, the following transformation applies:

```
for i = 1 to n  →   for i = 1 to g
  ...                 for i'=1 to n/g
                        ...
```

The second difficulty is topological. Carelessly coalescing loops may introduce cycles in the graph (V, T). The simplest pathological example is:

```
for i = 1 to 10       for i = 1 to 10
  0 send 1 t            1 send 0 t
done                  done
for i = 1 to 10       for i = 1 to 10
  1 recv 0 t            0 recv 1 t
done                  done
```

Coalescing loops around communication links would lead to a graph with two loops, with each loop being topologically before *and* after the other. Another example where coalescing cannot be performed naively is the following:

```
for i = 1 to 10      for i = 1 to 10
  0 send 1 t1          0 recv 1 t1
  0 send 1 t2        done
done                 for i = 1 to 10
                       0 recv 1 t2
                     done
```

Here we have a loop communicating with two other loops that need to be kept sequentially ordered. There is no reason to favor coalescing along one link compared to the other. In this case, we have decided not to perform an arbitrary coalescing, and keep all loops as is.

Even though troublesome cases like the ones just shown are unlikely in real code, the coalescing phase needs to take a principled approach. First, coalescing is only performed on sets of loops that communicate. Second, the effect of coalescing on the topology of the resulting graph may prevent coalescing, to avoid nonsensical systems. Here is the algorithm we use: starting with a system (V, T, C) where all links in C are regular:

- compute $V' = \{V'_1, \dots, V'_N\}$ the set of connected components of (V, C), i.e., groups of loops that communicate;
- build the graph $G' = (V', T')$, with $(v'_1, v'_2) \in T'$ if and only if $\exists v_1 \in v'_1, v_2 \in v'_2$ such that $(v_1, v_2) \in T$, i.e., the condensation of graph (V, T);
- compute the strongly connected components (SCCs) of G', i.e., sets of groups of loops that constitute topological cycles;
- coalesce loops in trivial SCCs that have no self loop.

A trivial SCC contains a group of loops that does not topologically conflict with any other SCC. If it does not conflict with itself, the set of loops it represents can be coalesced after all iteration counts have been made equal (as explained above). Coalescing a set of loops removes these loops and their communication links, and inserts a new loop whose body is the recursive merging of the bodies of the original loops.

An important property of this merging process is that loops are coalesced only if they communicate. Merging does *not* simply align constructs between systems and "factor" out loops. Instead it builds loops that span several processes only when these processes actually communicate, and leaves concurrent loops separate. This will be apparent in some of the example programs used in the next section.

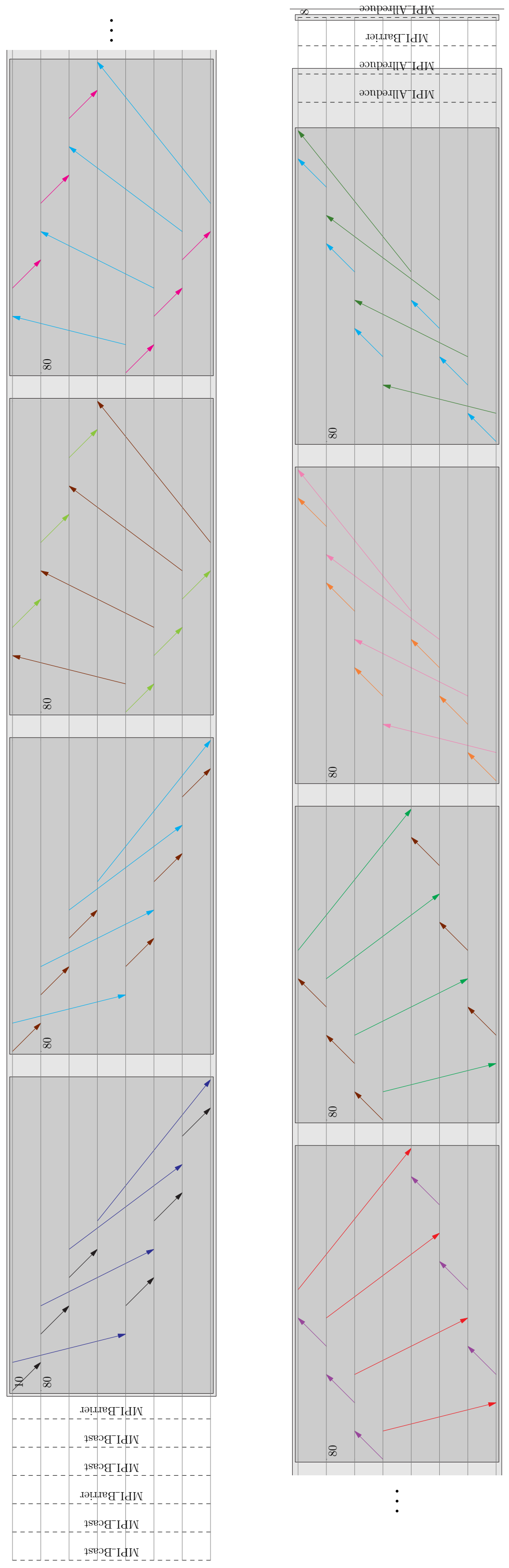


Fig. 4. Sweep3d, over 8 processes (full execution).

IV. Visualization

Visualizing a distributed system's communications is fairly easy, using timelines for processes, and arrows for messages. There are however two major inconveniences. First, traces are usually long, and it is difficult to have a high-level view of the program behavior. Second, message exchanges need careful positioning to make patterns appear. We think that building a loop-based model of the trace the way we just described and directly displaying this model helps on both aspects.

We have adopted a direct representation of our data-structures, explicitly displaying loops and their process coverage. Graphs have a line per process as usual, but these lines may change their vertical position to match loop coverage. A Lamport clock is used to assign horizontal positions to constructs: all positions are relative to the start of the current system, including when the system represents a loop body.

Figure 5 shows four examples of model visualization, in increasing order of complexity[1]. The first model (BT on 16 processes) is the simplest of all, since a very large part of its execution is made of 200 iterations of a loop. Capturing this repeating pattern simply shortens the display (by a factor 200 here). The second model (LU) is similar, except that the trace shows a two-level pattern, with an outer loop whose body contains two inner loops and several individual messages. In this case, it would be very difficult to discern this pattern in an complete, flat trace. Another example of multi-level patterns is on Figure 4, displaying the model of a `sweep3d` running with 8 processes, where the outer loop contains 8 loops and two collective operations.

The next example on Figure 5 show interesting characteristics of our model structures, namely the link between the coalescing of loops along communications channels. The third graph (CG) has an outer loop with a fairly large body, including seven loops. Interestingly, four of these loops have a restricted process coverage, designating a part of the execution where four groups of processes have concurrent behavior (in terms of communication). Our point is that this spatial partitioning is explicit in the model, each of these loops being represented by its own construct.

Finally, the last example (SP) adds an additional level of abstraction, by moving away from the usual timeline frame, where events from one process are to be found on a single horizontal line. What happens here is that non-communicating concurrent loops appear (as in the previous example), but these loops cover non-consecutive ranges of processes. Rather than spreading the loop across non-participating processes, thereby blurring the dynamic grouping, we have chosen to keep loops as a tight boxes and instead make process timelines change their vertical position to meet the loops they participate in. Because these moving lines are hard to follow, the process number is also displayed on the left side of the loop when some processes have an usual vertical position.

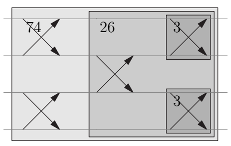
[1]We have done our best to find a trade-off between coverage and readability in these graphs, by removing mildly interesting leading and trailing phases. The reader may wish to look at the graphs on screen. We apologize for any inconvenience. Full execution traces are available as PDF files from the authors.

V. Generating Trace Scanners

The major benefits of having a formal model based on loops is that, first, this model is analyzable with techniques usually applied to source programs, and second that the model itself is executable, for example to replay the sequence of events for one or more processes of the program. This section tries to illustrate both aspects.

A. Counting Events

The loop-based model can be used to delimit regions of interest in the trace, by highlighting phases of repetitive communication patterns. It can also be used to compute simple characteristics of selected phases. An immediate example is that of communication matrices. Once a region has been delimited, it is immediate to count the number of messages exchanged by consulting the model, without any need to access the original traces. Figure 6 shows a simple example, on a loop taken from a model of the NPB CG program run on 4 processors. This loop covers 54464 events. The model lets one directly compute the number of messages exchanged between any two processes, producing the matrix displayed on Figure 6. This can be done by a simple post-order traversal of the loop. The virtue of loop-base models is double in this case: first it helps locate coherent phases of the execution, and second, it directly provides simple characteristics of these phases.



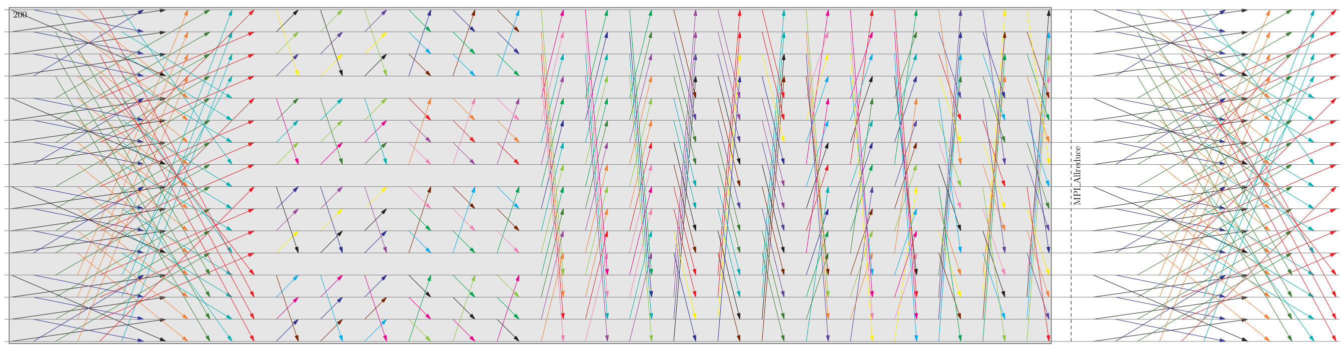
	0	1	2	3
0		5846		
1	5846		1924	
2		1924		5846
3			5846	

Fig. 6. A loop (in CG/C on 4 processors) and its communication matrix.

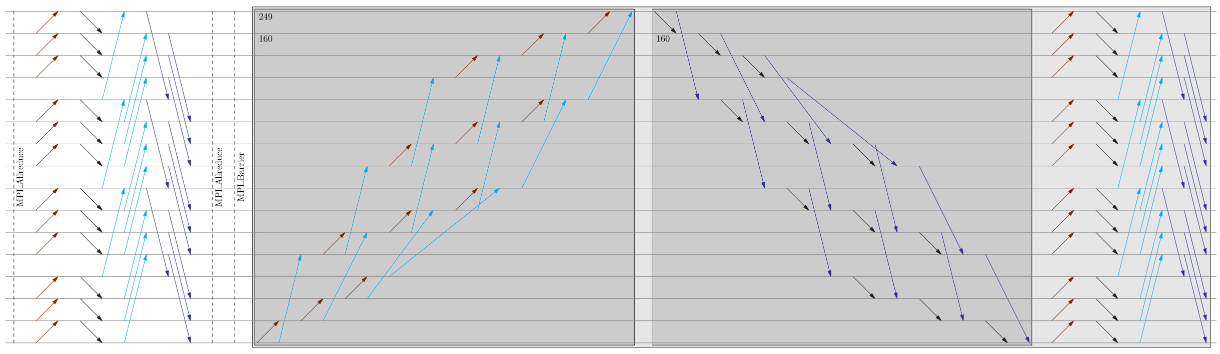
B. Generating Data Extractors

The modeling algorithm presented in the previous sections is purely topological, in that it uses only the chronology of intra-node events (too build local loop nests) and the volume of communication between process (to merge loops into a global model). In practice, traces always contain other kinds of data, usually of quantitative nature: typically, messages have sizes, events have timestamps, etc. These data are much less regular: timestamps, for example, show variations that are sufficiently large for them to be considered irregular, even in favorable execution conditions. Deciding whether they should be included in the analysis is difficult. From the point of view of modeling, the dilemma is the following: either sacrificing regularity and manipulate raw data, or sacrificing precision in the hope of finding regular patterns. Regularity being the mainspring of our work, the first option is unacceptable. Precision being essential when dealing with quantitative data, the second option is hardly tenable.

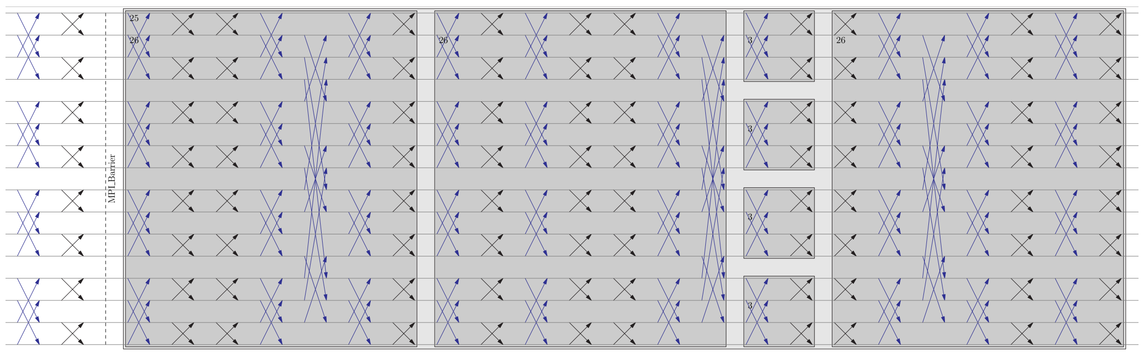
We have chosen to experiment another way to solve the dilemma, with the goal of leveraging regularity and keeping



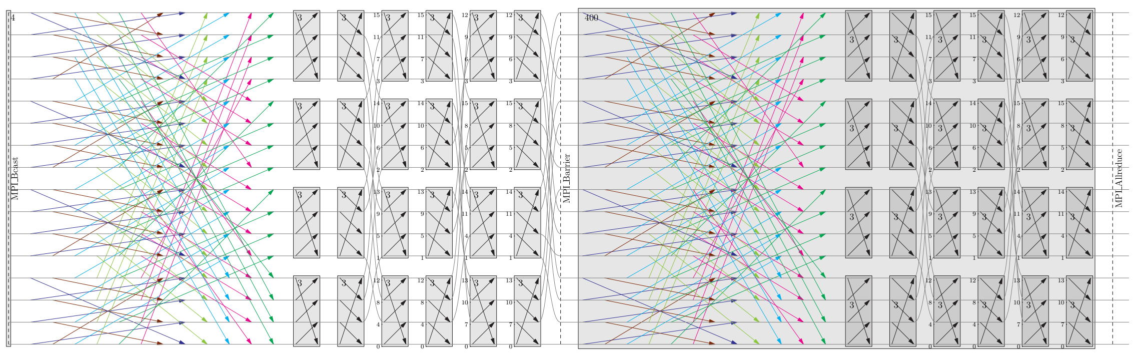
(a) BT



(b) LU



(c) CG



(d) SP

Fig. 5. A gallery of NPB models (class C, 16 processes): gray boxes represent loops (with iteration count in the upper left corner, and gray level indicating loop-depth), message colors represent tags, and vertical dashed lines represent collective operations. All graphs show more than 95% of events in the trace.

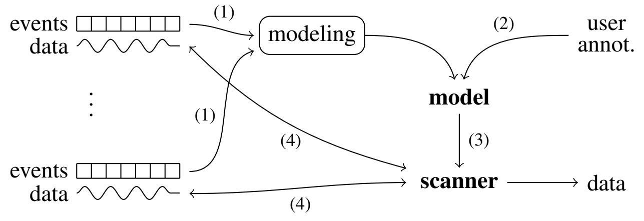


Fig. 7. A workflow for the analysis of quantitative data (see text for details).

precision. The idea is the following: during tracing, topological and quantitative data are stored in distinct trace files (conceptually, at least). Quantitative data related to a given topological event are identified by their position. The data-trace is kept on the local node; it can even be quantized and/or compressed, as long as random access to a record given by position remains possible in a reasonably efficient way. In that setting, we propose the workflow illustrated on Figure 7:

1) events traces are used to build a global model (as explained in the previous sections);
2) the user selects events of interest, along with the data she wishes to obtain, by annotating the model;
3) an *annotation compiler* generates a data-access program;
4) the program is run and retrieves the data stored on the nodes.

The data can then be processed by any appropriate client program.

The user selects one or more constructs in the loop-based model (start and end of loops or loop iterations, collective operations, or even individual messaging or local events). This selection is enough to compute, from the model, a function giving the positions of the occurrences of the selected construct(s) in the trace. This function is always a polynomial in the indices of the loops surrounding the constructs. It is therefore immediate to copy these loops to create a program that enumerates the positions of all instances of the events of interest, seeks into the data-trace(s) to obtain the data, and forwards them to a processing client for analysis, display, etc.

Let us take a short illustrative example. Suppose the user wants to know the time taken by the second inner loop on processes 0, 1, and 5 in the run of LU pictured in Figure 5(b). The annotated model appears on Figure 8.

In this example, C-like comments provide internal positions computed by the scanner generator: `@` indicates position (on all processes covered by the construct, relative to the enclosing context), and `#` in loop bodies indicates duration of one iteration. The `get` annotation requires the value of an attribute (here `time`) for a given set of processes (here `{0,1,5}`) covered by a construct (in our text-based annotation system, arrows indicate whether the annotation is attached to the next or previous construct). Every annotation will produce one output record (labeled with the string after `get`) for each occurrence of the construct. From that annotated model, the (pseudo-)code generated by the annotation compiler appears in Figure 8. Obviously, the generated code depends on the way the data-traces are accessed (this code could even be a distributed program itself). The expressions giving locations of data points are, however, always the same.

Provided a simple annotation user interface, any kind of data extraction can be performed this way.

VI. Related Work

A. Parallel Trace Modeling

There are basically two approaches to trace processing. The first one, profiling, includes timing information and/or aggregates of quantities; see, e.g., [1]. The second is more qualitative in nature, focusing on building an abstract model of the program behavior. This paper addresses a specific form of the latter approach, in the particular case of parallel and distributed systems. The goal is to build a model of all communications between processes. In that field many studies have been focusing on detecting repetitions in traces. For instance, ScalaTrace [9], [4] includes an incremental algorithm not unlike our NLR algorithm. Krishnamoorthy and Agarwal [10] use variations of Sequitur [11] to build one or more grammars from a trace. Xu *et al* [12] use a variation of the Crochemore algorithm [13] to locate the repetitions. In all cases, the trace is made of discrete symbols taken in some specific finite alphabet. Even though NLR theoretically has more expressive power, in the current state of our system any of these algorithms could be used instead.

In a related approach, Xu and colleagues [14] have studied *trace logicalization*, where the underlying topology is explicitly extracted and used to formulate a unified trace. We feel that building a loop nest representing a parallel trace can help in highlighting topological properties of the communication scheme.

B. ScalaTrace

ScalaTrace [9], [4] is a system for deterministic compression and replay of parallel communication traces. ScalaTrace introduced the *model-then-merge* strategy, that we have borrowed in this work: individual traces undergo loop recognition, and at the end of the run are sent to a centralized component which merges all models to produce a global trace.

ScalaTrace's models are made of *regular section descriptors* (RSD) and *power RSDs*, which are essentially nested loops. ScalaTrace also has some knowledge of MPI primitives and calling contexts, and encodes message destinations as offsets from the source, which makes it somewhat specific to MPI programs: local traces actually contain MPI calls (with parameters).

The merging phase in ScalaTrace heuristically solves an iterative multiple sequence alignment problem. Aligning elements in distinct individual models is based on syntactic matching, and targets high compression rates rather than expressive power. The models can be interpreted to replay the program.

Later work on ScalaTrace [15] aims at keeping timing information in the trace. Timestamps are quantized on-the-fly by the use of dynamically balanced histograms. This is

```
for i0 = 0 to 248 // @{0=671,1=999,5=1327...}
    // #{0=644,1=966,5=1288...}
    for i1 = 0 to 159 // @{0=0,1=0,5=0...}
        ...
    done
    get "before" {0,1,5}.time ->
    for i1 = 0 to 159 // @{0=320,1=480,5=640...}
        // #{0=2,1=3,5=4...}
        ...
    done
    get "after" {0,1,5}.time <-
    print start,end
    0 send 1 2 // @{0=640}
    1 send 2 2 // @{1=960}
    0 recv 1 2 // @{1=961}
    ...
done
```

```
for (i0=0 ; i0<=248 ; i0++ ) {
    output_start("before",i0);
    output_val(0,991+644*i0,TIME);
    output_val(1,1479+966*i0,TIME);
    output_val(5,1967+1288*i0,TIME);
    output_end();
    output_start("after",i0);
    output_val(0,1311+644*i0,TIME);
    output_val(1,1959+966*i0,TIME);
    output_val(5,2607+1288*i0,TIME);
    output_end();
}
```

Fig. 8. Generating data extractors: an annotated model on the left, the generated code on the right.

in effect a successful implementation based on a trade-off between regularity and precision (mentioned in Section V-B).

VII. Conclusion

This paper presents a communication trace analysis strategy for distributed programs. The strategy is based on the modeling of individual process traces with loops nests, and the merging of these models into a global graph of loops and events. Examples have shown that a direct visualization of this graph provides a compact representation of the program's execution. The model can also help accessing additional trace data, and data scanners can be generated automatically for a large class of quantitative analysis tasks. We are convinced that our topological approach can be useful for other analyzes.

This work opens several potential research directions. First, all regularities are searched for along the chronological dimension: loops are using counters that are basically abstract (and sometimes multi-dimensional) clocks. Even though our algorithm scales well for large numbers of processes, the results are barely usable, for instance for visualization. New notions of regularity have to be defined to compact models along the "spatial" dimension.

Second, the models we build focus on communication, without ever considering what is carried by the messages. Modeling data transfers somehow would provide deeper insight into the program's behavior. Again, this requires new notions of regularity (NLR's ability to build affine functions may help here). But the result would allow far reaching analyzes.

References

[1] J. Gonzalez, J. Gimenez, and J. Labarta, "Automatic detection of parallel applications computation phases," in *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy*, May 2009, pp. 1–11.

[2] C.-F. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. L. Lusk, and W. Gropp, "From trace generation to visualization: A performance framework for distributed parallel systems," in *Proc. of SC2000: High Performance Networking and Computing*, Nov. 2000.

[3] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 1581–1592.

[4] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 696–710, 2009.

[5] S. Shende and A. D. Malony, "The Tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[6] Message Passing Interface Forum, "Mpi: A message-passing interface standard, version 2.2," Tech. Rep., September 2009.

[7] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Paralell Benchmarks," NASA Ames Research Center, Tech. Rep. RNR-94-007, Mar. 1994.

[8] A. Ketterlin and P. Clauss, "Prediction and trace compression of data access addresses through nested loop recognition," in *6th International Symposium on Code Generation and Optimization (CGO 2008), Boston, MA, USA.*, M. L. Soffa and E. Duesterwald, Eds., Apr. 2008, pp. 94–103.

[9] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *21th International Parallel and Distributed Processing Symposium (IPDPS07)*. IEEE, Mar. 2007, pp. 1–11.

[10] S. Krishnamoorthy and K. Agarwal, "Scalable communication trace compression," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 408–417, 2010.

[11] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, no. 1, pp. 67–82, 1997.

[12] Q. Xu, J. Subhlok, and N. Hammen, "Efficient discovery of loop nests in execution traces," in *MASCOTS 2010, 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Miami, Florida, USA*, Aug. 2010, pp. 193–202.

[13] M. Crochemore, "An optimal algorithm for computing the repetitions in a word," *Inf. Process. Lett.*, vol. 12, no. 5, pp. 244–250, 1981.

[14] Q. Xu, J. Subhlok, R. Zheng, and S. Voss, "Logicalization of communication traces from parallel execution," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC 2009), Austin, TX, USA*. IEEE, Oct. 2009, pp. 34–43.

[15] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz, "Preserving time in large-scale communication traces," in *Proceedings of the 22nd annual international conference on Supercomputing*, ser. ICS '08. ACM, 2008, pp. 46–55.

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399