Athapascan-1: On-Line Building Data Flow Graph in a Parallel Language

François Galilée, Jean-Louis Roch
Gerson G. H. Cavalheiro*, Mathias Doreille
LMC-IMAG-APACHE Project†
Grenoble, France
http://www-apache.imag.fr

Abstract

In order to achieve practical efficient execution on a parallel architecture, a knowledge of the data dependencies related to the application appears as the key point for building an efficient schedule. By restricting accesses in shared memory, we show that such a data dependency graph can be computed on-line on a distributed architecture. The overhead introduced is bounded with respect to the parallelism expressed by the user: each basic computation corresponds to a user-defined task, each data-dependency to a user-defined data structure.
We introduce a language named Athapascan-1 that allows built a graph of dependencies from a strong typing of shared memory accesses. We detail compilation and implementation of the language. Besides, the performance of a code (parallel time, communication and arithmetic works, memory space) are defined from a cost model without the need of a machine model. We exhibit efficient scheduling with respect to these costs on theoretical machine models.
Keywords: Multithreading, macro-data flow languages, on-line scheduling, parallel complexity.

1. Introduction

Recent work in the field of parallel programming has resulted in the definition of extensions of sequential languages that can be theoretically proven efficient when scheduled on abstract machine models. Such languages allow explicit parallelism independently of a specific architecture. The performance of a program is evaluated directly from a language based performance model [1] that specifies the costs of the primitive instructions and rules for composing costs across program expressions. The adequacy between these theoretical costs and the effective performances of execution on an architecture is then related to the scheduling algorithm used. To cope with programs including branching instructions that are unpredictable at compile time, the schedule is computed on-line. Costs then include scheduling overheads.

Most on-line scheduling algorithms rely on a (greedy) list schedule [7]. This consists of an on-line mapping of ready tasks to idle processors. Such a schedule leads to nearly times optimal executions, including scheduling overheads, on theoretical machine models such as the PRAM.

To achieve efficiency regardless various criteria, some knowledge about the execution is often required. For instance, to bound the amount of memory usage, the knowledge of a sequential schedule (i.e. a total ordering of tasks that results in a correct sequential execution) may be used. In this way, list scheduling leads to parallel computations achieving a linear speed-up while requiring a space related to the one of the sequential execution for certain classes of programs: strict computations [4], nested computations [2] or planar graphs [3]. Furthermore, in practice, due to the magnitude of the ratio between local and remote memory access costs, some significant improvement can be brought to a schedule some knowledge about the data flow corresponding to the execution [8]. Some programming environments use such a graph in input [16].

Several languages have been designed that enable the on-line building of the data flow describing the execution. Most of them are built on top of a standard sequential language commonly used in high performance computing. To bound the related overhead, parallelism is expressed by the user who defines the grain of data and control. Although in Jade [14] and BSP [9] instructions are grouped by block, most languages are based on a parallelism expressed via asynchronous function calls, like Cilk [11]. Synchronizations that will occur during a sequence of instructions is expressed at task creation or by specific statements (often a `sync` instruction [15, 11]) that allows a task to synchronize with others.

Such an explicit synchronization instruction bounds the

*CAPES–COFECUB Brazilian scholarship
†CNRS, INPG, INRIA and UJF

on-line computation of future data flow dependencies that will occur after the synchronization. So, it forbids the on-line use of static strategies although some are of theoretical and practical interest when tasks are of known cost [16]. Besides, to enable efficient scheduling on a distributed architecture, a migration mechanism is then required to eventually move a task that was blocked and becomes ready to another processor.

By typing the memory accesses a task can perform, we exhibit a parallel language, named Athapascan-1 (Ath stands for *Asynchronous Tasks Handling*) with no explicit synchronization instruction that allows the on-line analysis of data-dependencies. Athapascan-1 is mostly inspired from Jade [14] concerning typing of memory accesses and Cilk [11] concerning parallelism expression.

In section 2, we detail the syntax and semantic of the language. The section 3 presents how the macro-data flow can be computed on-line with a bounded overhead; we also prove that space and time efficient executions can be achieved on theoretical machine models without need of migration. Particularities of the implementation, that uses local multithreading, are detailed in section 4; this section presents also some experimental measures on a distributed and on a shared memory architecture.

2. The Athapascan-1 language

2.1. Overview

In order to deal with data and control flow at a grain defined by the user (macro-data flow), parallelism is expressed through asynchronous remote procedure calls, denoted as *tasks*, that communicate and are synchronized only via access to a shared memory.

The Athapascan-1 semantics rely on shared data access and ensure that the value returned by the read statements is the last written value (or a copy of) according to the lexicographic order defined by the program: statements are lexicographically ordered by '`;`'. This choice of such a sequential semantic is motivated by its direct readability on the program source (an obvious example in Fig. 1). This order defines a total ordering on all tasks during the execution.

The control of the accesses semantic during execution is entirely data driven: the precedences between the tasks, the needed communications or the data copies are ensured automatically by the runtime system. It is based on an entry-release consistency scheme; the objects entries are always done at beginning of tasks and the corresponding release at the end of tasks. The prototype of a task specifies accesses performed on shared objects: `r` stands for *read*, `w` for *write*. All tasks are a priori independent; conflicts between two tasks that access a same object are solved using the total lexicographic ordering. For instance, in Fig. 1, the task `update(a)` precedes `print(a)` in the lexicographic order; then, `print(a)` is delayed until `update(a)` resumes. The program will thus print 5 on the output.

Athapascan-1 is implemented as a C++ library and is then fully compatible with C and C++ languages. A simple ANSI C extension is handled by a basic preprocessor. For the sake of simplicity, the syntax presented here is the one recognized by this preprocessor; it makes the use of the C++ library easier by replacing typical C++ constructions by keywords.

```
task update( shared( w )< int > x )
      { x.write( 5 ); }
task print( shared( r )< int > x )
      { printf( "%d", x.read() ); }

task test() {
  shared< int > a;
  fork update( a );
  fork print( a );
}
```

Figure 1. Lexicographic based semantic.

2.2. Syntax and abstract representation

Tasks and closures A *task definition* is similar to a C procedure definition, having simply the `void` returned type replaced by the `task` keyword:
`task user_task(<params>) {<statements>}`

A *task* implements a sequential computation whose granularity is fixed by the user; it is created in program statements by prefixing a standard C++ procedure call by the `fork` keyword: `fork user_task( <args> );`. This statement creates an object, called *closure*, gives it for scheduling to the current scheduler and returns for continuation (asynchronous task creation). A closure is a data structure that contains an instantiation of the user task (that defines the method to run) and the list of its effective parameters. A closure is said to be *ready* if all the arguments that will be read by the task are ready or *waiting* if some argument that will be read is not ready. A parameter is said to be not ready for a task iff this task has a predecessor not yet completed in lexicographic order which will write the same parameter.

The state of a closure is then directly linked to the state of the effective parameters that will be read by the task. Two types of parameters are distinguished: first, the classical parameters by value which are always ready since the closure possesses a copy of them; and second, the parameters which are references to shared data versions.

Shared data and their versions The shared memory, that allows the tasks to synchronize, is composed by shared data. In this memory, an object `x` of type `T` is declared as follows:
`shared< T > x ;`
The type `T` defines the granularity of the data handled in the

algorithm. A succession of *versions* is associated with each shared data: each shared data version represents the value at a certain instant of execution. When declared, a shared data creates an object, called *evolution*, containing pointers toward two *transitions*; these transitions are objects managing the allocation, state and accesses of data versions. The first transition manages the version of the data that will be read (the *current* version of the shared data) and the second the version that will be generated (let us say written) by the execution of the task (the *future* version of the shared data).

The shared data version references possess the following methods:

```
T& access();      void write( const T& );
const T& read();  void cumul( const T& );
```

Example. Fig. 2 shows the different data structures that compose the Athapascan-1 system. All these objects are dynamically allocated in the heap and return to the heap when they are completed: after task execution in the case of closures; when no access can be made on a data in the case of transitions.

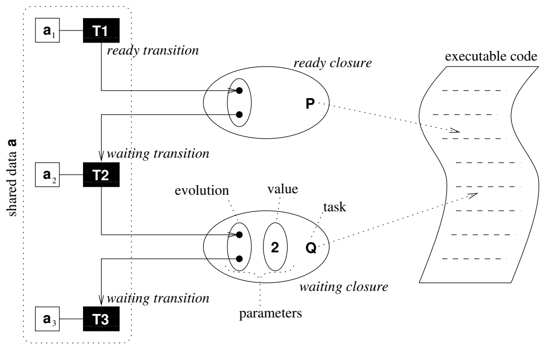

Figure 2. Internal data structures.

Fig. 3 shows two codes for computing the nth Fibonacci number. The first version is the familiar recursive procedure that computes the nth Fibonacci number. If the threshold is reached, the `fibo` task just writes the result into the shared data `res`; else it creates two tasks that will compute concurrently the two previous Fibonacci numbers and a task `sum` that will add these two numbers and write the result in the shared data `res`. This summing task will be delayed until the two Fibonacci tasks have been completed, because the access made on the shared data are incompatible (write and read) and the two Fibonacci tasks have been created before. The second version is a cumulative version of the ”sum tree” developed by the first version.

2.3. Lexicographic ordering and parallelism

In order to respect sequential consistency (lexicographic order semantic), Athapascan-1 has to identify the related

```
task fibo( int n, shared(w)<int> res ) {
 if( n<2 ) res.write( n );
 else {
   shared<int> x, y;
   fork fibo( n-1, x );
   fork fibo( n-2, y );
   fork sum( x, y, res );
 }
}
task sum( shared(r)<int> x, shared(r)<int> y,
          shared(w)<int> z ) {
  z.write( x.read() + y.read() );
}
```

(a) – recursive version

```
cumulative add( int& x, const int& y )
{ x += y; }
task fibo( int n, shared(cw<add>)<int> res ) {
  if( n<2 ) res.cumul( n );
  else {
    fork fibo( n-1, x );
    fork fibo( n-2, y );
  }
}
```

(b) – cumulative version

Figure 3. Two versions of code to compute the nth Fibonacci number.

data version for each read performed on a shared data. However, parallelism detection is easily possible in this context if all the tasks define the shared data objects that will be accessed during their execution (for independent tasks detection), and which type of access will be performed on them (for tasks precedence detection and shared data versions evolution).

For this reason the Athapascan-1 tasks can not perform side effects (all manipulated shared data are located in the parameter list) and shared parameters are typed according to the future manipulation.

Access right: evolution of shared data versions. In the declaration of formal parameters of tasks, the references to shared data version are typed by their *access right*, i.e. the kind of manipulation the task (and all its sub-tasks, due to lexicographic order semantic) is allowed to perform on the shared data. These rights are `r_w` for read&write modifications, `w` for writing, `cw<f>` for accumulation and `r` for read only.

Remark concerning accumulation. The accumulation law is an user defined law that is supposed to be associative and commutative. Note that this law is part of the reference type, so a mix of different laws is not allowed, but obey to lexicographic order semantic (Fig. 3(b)). The initial value is the previous value of the shared data according to the lexicographic access order.

The postponed access right. To improve parallelism, there is a refinement to access right, that denotes if an access will be either performed or not on the shared data. An access is said to be “postponed” (access right suffixed by `p`) if the task will not perform any access on the shared data but will create tasks that may benefit of this right.

3. Data flow building and cost model

Adding restrictions on accesses that can be performed by a task, we establish that a representation of data dependencies can be computed on-line on a distributed architecture with a bounded overhead both in time and memory usage. This enables the definition of a cost model related to the language. We exhibit scheduling algorithms with provable performances with respect to this cost model on theoretical parallel machines.

3.1. Shared data version access graph

In order to be able to determine the state of the transitions and closures, Athapascan-1 dynamically maintains a graph of the shared data versions access. This graph is composed of the closures and transitions for the nodes and the pointers of evolutions for the edges. At each instant it gives a partial description on the data flow dependencies that will occur until the end of program.

The evolution of this graph happens at each task or shared data creation (addition of edges and/or nodes) or task termination (removing of edges and/or nodes, node's state evolution), as shown in Fig. 4. Schedulers can take benefit of this graph which is maintained, for semantic reasons, by the system. Without over-cost they have some relevant information on the application to schedule.

3.2. Bounding cost of Athapascan-1 statements

Due to the access semantic (based on the lexicographic order), the accessed values can be easily determined on the source code. It is also possible to evaluate the time and memory cost of any Athapascan-1 program. In order to bound the cost of the on-line building of the evolution graph, the following restrictions are added:

• $\mathcal{C}_1$: All graph modifications and task creations are local (need no communication). This is discussed in the section 4.1.

• $\mathcal{C}_2$: All shared data versions that can be read (read right and direct mode) by a task are always ready during the task execution.

It follows that the tasks which can directly read a shared data are not allowed to create tasks that would write on this shared data: else, the creator task would have to wait until the resumption of the newly created task before any read of the new shared data version (else the access semantic would not have been respected). So, as a consequence, the type `shared(r_wp)<T>` has no sense, and for example a task having a `shared(r_w)` can not create a task requiring a `shared(w)` as formal parameter.

• $\mathcal{C}_3$: A task creation (`fork`) generates no data copy.

It follows that the tasks which can directly write a shared data are not allowed to create tasks that would read on this shared data: else a copy of the last written value should be stored for the created task, or the creator stopped until the created resumes all reads on the last shared data version. So, as a consequence, the type `shared(rp_w)<T>` has no sense and, for instance, a task having a `shared(r_w)` can not create a task requiring a `shared(r)` as formal parameter.

Definition 1 *A correct Athapascan-1 program verifies both the syntax and conditions $\mathcal{C}_1$ to $\mathcal{C}_3$.*

Note that the strong typing of accesses in shared memory enables the verification of the correctness of a program at compile time. The allowed conversions of shared data version types at task creation are summarized in Fig. 5.

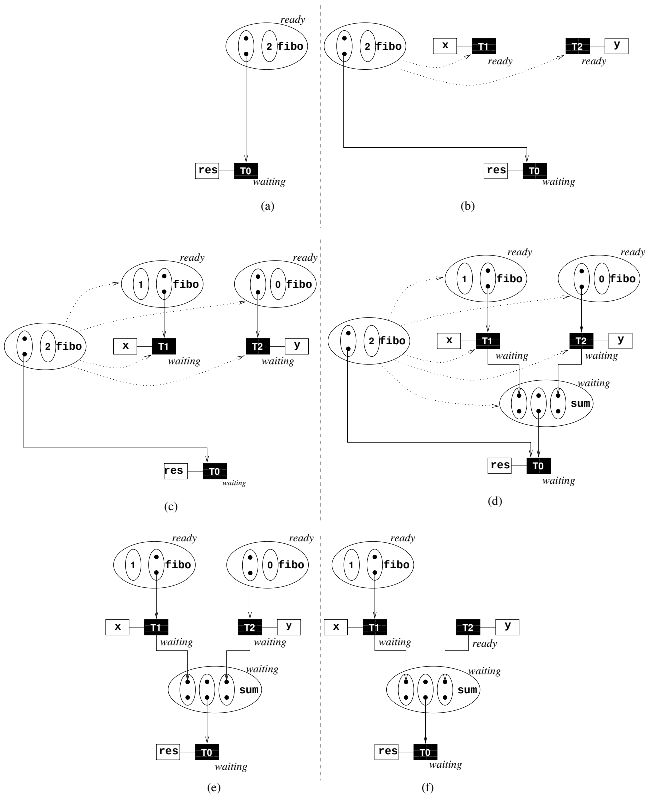

Figure 4. Dynamic evolution of the shared data versions access graph for the recursive version of `fibo(2)`: (a) initial state, (b) after the creation of the two shared `x` and `y`, (c) after the creation of the two `fibo` tasks, (d) after the creation of the `sum` task and just before the end of the `fibo(2)` (e) just after (f) after the execution and completion of `fibo(0)`.

formal parameter	required type for the *effective parameter*
`shared(r[p]     )<T>`	`shared(r[p]     )<T>` `shared(rp_wp    )<T>`
`shared(w[p]     )<T>`	`shared(w[p]     )<T>` `shared(rp_wp    )<T>`
`shared(cw[p]<f>)<T>`	`shared(cw[p]<f>)<T>` `shared(rp_wp    )<T>`
`shared(r[p]w[p])<T>`	`shared(rp_wp    )<T>`

Figure 5. Allowed conversion for passing reference on a shared data version as parameter of a task.

Lemma 1 *In a correct program, any Athapascan-1 statement (i.e.* `fork`, `read`, `write`, `cumul`, `access`, `shared`*) has a bounded cost both in time and memory space. Each graph modification is made in constant time and space.*

This results from shared types and conditions $\mathcal{C}_1$ and $\mathcal{C}_2$. □

Lemma 2 *There is no any precedence relation between a task and the task that creates it.*

This results from shared types and condition $\mathcal{C}_2$. □

This property enables us to exhibit a sequential schedule of tasks, denoted as the *reference order*, that respects the sequential semantics while being different from the classical depth-first sequential one.

Let s denotes a sequence of statements containing no `fork` statement and F denotes a `fork` statement. Then, the trace corresponding to a sequential depth first execution of `fork` statements can be represented by a word $f = s_1 F_1 s_2 F_2 \ldots s_{n-1} F_{n-1} s_n$ (any s_i may be empty). The entire independence between the created task and the creator task (lemma 2) implies that this trace is semantically equivalent to the trace $f' = s_1 s_2 \ldots s_{n-1} s_n F_1 F_2 \ldots F_{n-1}$. This order of execution corresponds to an inner most outer most order of evaluation: it is called the *reference sequential order* of statements evaluation and denoted by $\mathcal{R}$ in the following sections. Fig. 6 illustrates this execution order.

Proposition 1 *The Athapascan-1 reference order of execution respects the semantic and requires no implicit copy.*

This results from the conditions $\mathcal{C}_2$ and $\mathcal{C}_3$. Note that the by value passing mode generates a copy which is considered as explicit. □

All assumptions concerning copies and synchronizations ensure that the Athapascan-1 system is not responsible of over-memory requirement: all decisions are to be taken by the scheduling policies or the user. This enables us to evaluate the cost of a program directly from the code.

```
task user_task( <parameters> ) {
  stmts_1;
  fork task_1( <args> );
  stmts_2;
  fork task_2( <args> );
  stmts_3;
}
```

(a) Depth-first sequential order

```
task user_task( <parameters> ) {
  stmts_1;
  stmts_2;
  stmts_3;
  fork task_1( <args> );
  fork task_2( <args> );
}
```

(b) Reference sequential order

Figure 6. Two equivalent programs.

3.3. Cost measures

Proposition 1 enables non-preemptive schedules: execution of a closure is then delayed until its parent (the task that has created it) resumes. From this reference order, costs (time, space, depth, communications) are defined directly on the code itself by related recurrence equations (max, $+$) [10, 1]. Here, following notations in [1, 4], those costs are defined on the trace of the execution. This trace can be represented by a bipartite DAG G (see Fig. 4, with node sets corresponding respectively to tasks (oval nodes in Fig. 4 and shared data versions (box nodes in Fig. 4). Each task node is weighted by its computation cost, related to the execution of the body and excluding forked tasks; each data node is weighted by the size of the data for a direct access (white box) and a unit constant for a reference or postponed access (black box). Note that those costs, related to the trace, may be unknown until execution completes; they are usually bounded with respect to the size of the input. T_1 and S_1 denote the sequential time and space, T_∞ the parallel time, C_1 and C_∞ the communication volume and delay.

Sequential time T_1 and space S_1. These are defined from a serial execution of the program following ther reference order $\mathcal{R}$. Since all fork statements are pushed in memory until the completion of a task, it can be noted that the space S_1 can be larger than the one required by a depth-first execution. However, if the number of fork statements is bounded by a constant, S_1 is increased only by a constant factor. For instance, the space S_1 related to program in Fig. 3(a) is $O(n)$.

Parallel time T_∞. Arithmetic depth T_∞ (or *parallel time*) is the depth of G taking into account weights of task nodes. T_∞ is then a lower bound of the minimal time required by any non-preemptive schedule on an unbounded number of processors ignoring communications times (PRAM model

[6]).

Communication volume and delay. Communication volume C_1 and delay C_∞ are evaluated from G similarly to T_1 and T_∞ but taking into account only weights of shared data version nodes. C_1 is the sum of the weights over all data nodes; Assuming that the shared memory is emulated in an auxiliary file, C_1 is then an upper bound on the total number of accesses performed in this file during a serial execution. C_∞ is the length of the critical communication path in G.

Scheduling overhead σ. Since the overhead involved by the scheduling of the graph is also involved, we denote by σ the size of G. Scheduling the program will require at least σ scheduling operations that may be performed in parallel.

In the following sections, we study the scheduling of an Athapascan-1 program on various machine models; time and space required by the execution on the machine (including the cost of the scheduling algorithm) are related to above abstract costs defined on the program itself that are independent of the machine.

3.4. Scheduling on a PRAM

We consider a PRAM [6, 10] with a bounded number p of processors; this allows us to eliminate communication overheads. In order to be consistent with Athapascan-1, we consider a CRCW-PRAM with cumulative concurrent write ones. For the sake of simplicity, we assume that, during execution, any task performs a bounded number (say 2) of `fork` statements.

Proposition 2 *Including scheduling overheads, any Athapascan-1 program can be executed on the p-PRAM in time (a) $T_\infty + \frac{T_1}{p} + O(\sigma)$ or*
(b) $T_\infty + \frac{T_1}{p(1-1/\log p))} + O\left(\log(\sigma) + \frac{\sigma}{p}\right)$.
Both schedules are deterministic and non-preemptive.

Both schedules are based on a greedy list strategy [7]: when a processor becomes idle, it gets a ready closure if any and performs its execution. Bounds for (a) and (b) differs from implementation of the strategy. To obtain (a), a global lock is implemented in the PRAM: each modification in the evolution graph is performed in mutual exclusion. The number of such modifications is bounded by $O(\sigma)$; thus both idle time corresponding to busy wait of the lock and management of evolutions of the graph are bounded by $O(\sigma)$.
A distributed management of the list leads to bound (b): all modifications on the evolution of the graph are performed by a specific subset of $\frac{p}{\log p}$ processors in time $\log(\sigma)$. Other $p(1-1/\log p)$ processors are dedicated to execution of closures. □

Bounds on (a) and (b) differs only from the scheduling algorithm; however, without knowledge of weights in G, it is not possible to decide which one achieves the best bound.

3.5. Scheduling on a distributed architecture

We consider now the scheduling on a distributed architecture DCM with p identical processors. The shared memory is emulated with the help of universal hashing functions [12]. The delay occurring for any access is assumed bounded by $h(p)$. In order to obtain efficient emulations ($h(p)$ constant or very small to p), a slackness strategy [13, 15] is used. It consists in emulating a $q(p)$-PRAM on the distributed architecture, $q(p)$ being larger enough compared to p. From here, the distributed machine is assumed to emulate a $q(p)$-PRAM with delay $h(p)$.

Proposition 3 *Including the cost of the schedule computation, any Athapascan-1 program can be executed on a p-DCM in time $\frac{q(p)}{p}T_\infty + \frac{T_1}{p} + h(p)\left[\frac{q(p)}{p}C_\infty + \frac{C_1}{p} + O(\sigma)\right]$. This schedule is non-deterministic and preemptive but requires no migration of running closures.*

The proof is deduced from proposition 2 applied on a PRAM with q processors. The whole number of remote access with delay h is bounded by C_1. This leads to a schedule on the q virtual processors with length bounded by $T_\infty + \frac{T_1}{q} + h\left(T_\infty + \frac{C_1}{q}\right)$. To obtain the emulation on the p processors, q/p virtual processors are emulated on each processor by synchronous preemptive threads. Due to the emulation of the shared memory, the algorithm is non-deterministic. On each processor, threads are dedicated to the execution of closures and are emulated preemptively. However, once a closure starts its execution on a processor, it is not migrated to another one. □

As a corollary, Athapascan-1 programs verifying $C_1 = o(T_1)$ and $\sigma = o(T_\infty)$ can be scheduled asymptotically optimally on a distributed architecture with $p = o\left(\frac{T_1}{T_\infty}\right)$ in time $(1+\epsilon)\frac{T_1}{p}$. This result is similar to those in [4] for Cilk programs.

A negative result however is that computations that involves a large number of communications may not been efficiently scheduled. For instance, any program with linear serial cost $T_1 = O(n)$ where n is the size of the input requires at least $W_c = \Omega(n)$ accesses. The schedule time is then $O(h(p)T_1/p)$: efficiency depends heavily on h.

3.6. Scheduling with space constraint

Previous schedules do not guarantee any bound concerning the space required. However, since they are based on a list of ready closures, sorting this list according to the reference order allows us to bound the memory space required with respect to S_1 [3]. Due to the lexicographic semantic, this list can be maintained sorted with no overhead according to the reference order: all insertions and deletions are performed in constant time. If all tasks are assumed to allocate $O(1)$ memory space for the execution of their own body, then the space required for the resulting schedule on the distributed machine is bounded by $S_1 + O(q(p)T_\infty)$ [3].

4. Distributed implementation

In this section we describe the distributed implementation of Athapascan-1. We focus on the transition distributed management and on the scheduling implementation. Particularly, we exhibits performance results obtained by a list scheduling which theoretical efficiency has been studied in the previous section.

In order to bound the delay occurring for remote accesses, each compute node of a parallel machine emulates a certain number of virtual processors (*threads*) that share a single address space. These threads are implemented by a runtime kernel, called Athapascan-0 [5]: it defines a parallel machine composed of a set of nodes executing a runtime environment providing communication and synchronization facilities and a local scheduling of threads enabling to hide communications latencies for remote memory access. Athapascan-1 is implemented on this runtime environment, using a bounded number of threads on the architecture. It provides the global scheduling of closures on the threads of the whole architecture.

4.1. Shared data versions access graph management

At compile time, besides verification of the correctness of the Athapascan-1 program, a code is generated for each fork statement, in order to create the corresponding closure. The management of the data-dependencies between closures (shared data versions access graph) is distributed: closures and edges are unique in the system, but transitions may be replicated (Fig. 7). So, a closure always locally accesses its connected transitions (via the pointers of its possessed evolutions). Then all the accesses to the shared data versions or the tasks creations are local events and create no communication. The time required for a task creation is therefore proportional to the number of its parameters.

In order to detect termination of accesses to a transition in a distributed asynchronous environment, a termination algorithm is implemented. Each transition is associated with a master node that computes a balance between increasing counters related to each of its replicate. These counters are managed locally on each site that possesses a replicate. When there is no more local access on the site, values of local counters are sent to the master node.

4.2. Scheduling implementation

The global scheduling algorithm is used to determine a site and a date to trigger the execution of a closure. The algorithm implemented by this level of scheduling distributes the work generated by an application attempting to optimize a global index of performance (memory, execution time or other).

After creation (`fork`) and until it is completed, the state of a closure depends on the graph of evolutions. Added to the states *waiting* and *ready* (paragraph 2.2), a closure may also get into the state *running*, when its instructions are executing sequentially and *executed* when the closure is completed. Each change of state can potentially trigger a scheduling action. Then, for each operation in the graph, a signal is sent to the related scheduler allowing it to explore the new graph configuration. So, it can get the information that it requires about the current global state of execution: for instance, closure attributes, parameters, state or precedence constraints.

4.3. Scheduling layers: an experimentation

We present in Fig. 8 some performance results (time in seconds) obtained from the execution of the Fibonacci program (Fig. 3(a)) for an input 40, using values ≤ 25 as threshold to halting the task generation; then, roughly half of the generated closures require less than a micro-second and the other ones (that sequentially compute fibo(25 or 24)) few milliseconds. $\#N = 32000$ tasks were generated, producing about $\#E = 100000$ edges. The experiments were performed on five architectures: two monoprocessors, an IBM RS-6000 (AIX 4.2) and a Pentium (Solaris 2.5.1); two distributed architectures, four nodes of an IBM-SP architecture (IBM RS-6000/370, AIX 4.2, IBM-MPI) and a four nodes network of workstation - NOW (Pen-

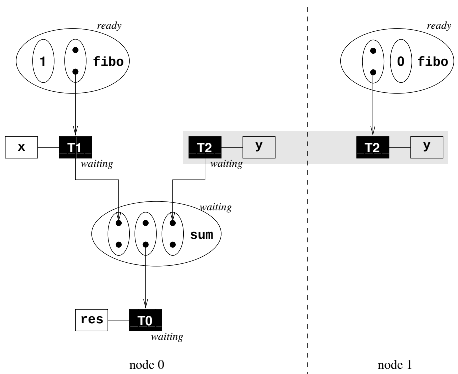

Figure 7. Replicated transitions on a distributed graph. When `fibo(0)` is completes, a message is sent from node `1` to `0` to warn the transition `T2` that no more writers exist on `1`.

	1 node architecture		4 nodes parallel architecture		1 node SMP architecture
	IBM-SP 1 proc/node	NOW 1 proc/node	IBM-SP 1 proc/node	NOW 1 proc/node	4 proc/node
Sequential	43.30	39.90	———	———	22.60
Ath-sequential	43.84	39.94	———	———	23.29
1 Thread	106.45	101.79	27.30	27.76	24.45
2 Threads	80.63	71.25	20.99	21.51	16.29
4 Threads	67.16	59.93	18.34	18.40	13.94
8 Threads	61.66	54.16	18.51	21.05	12.98

Figure 8. Influence of the local scheduling of threads on global greedy on-line scheduling.

tium, Solaris 2.5.1, LAM-MPI, Myrinet); and also under a SMP (4 Pentium Pro, Solaris 2.5.1). In the table, the lines correspond respectively to the execution of a pure C++ sequential algorithm, to an Athapascan-1 program compiled to generate a sequential code and to the parallel execution using n execution-threads in each node.

Comparing the performances of the pure sequential algorithm with the sequential version for the Athapascan-1 code, we verify the overhead introduced by the Athapascan-1 programming style is very small. However this is not true in the parallel version where we can observe the overhead produced by the scheduling and graph management. This overhead is partially overlapped when the number of threads on each node is increased. Besides, on the three parallel architectures, a speed-up of about 2 can be obtained with 4 processors. Speed-ups close to 4 were obtained with thresholds larger than 25.

5. Conclusion

We present the Athapascan-1 language that enables the on-line building of data flow dependencies with bounded overhead. Its semantic is based on a lexicographic order between instructions; it enables an implicit sequential non-preemptive schedule. Both grain of data and computation are explicit but independent of the target architecture. Parallelism is expressed by asynchronous creation of tasks. Any task can be scheduled such that, once started, it can continue its execution with no preemption. This property enables provable scheduling.

The language is related to a cost model that defines parallel depth, work, sequential space, communication volume and delay. Efficient schedules are developed that achieve both optimal time and bounded space on a distributed architecture for a large class of programs. However, for lots of practical applications, the use of this scheduling algorithm on a distributed architecture leads to poor performances, far from the ones obtained by simple static strategies. Since the scheduling can be changed by code annotation, an issue is then to take benefit of the on-line partial knowledge of the annotated data flow graph in order to implement – in an on-line context – such strategies.

References

[1] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

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

[3] G. E. Blelloch, P. B. Gibbons, Y. Matias and G. J. Narkilar. Space efficient scheduling of parallelism with synchronization variables. In *Proc. of the 9th Symp. on Parallel Algorithms and Architectures*. ACM Press, 1997.

[4] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202-229, 1998.

[5] J. Briat, I. Ginzburg, M. Pasin and B. Plateau. Athapascan runtime: efficiency for irregular problems. In *Proc. of EuroPar'97*. Passau. Aug. 1997.

[6] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. of the 10th ACM Symposium on Theory of Computing*, San Diego, CA, 1978. ACM Press.

[7] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–426, 1969.

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

[9] M. Goudreau, J. Hill, K. Lang and B. McColl. A proposal for the BSP worldwide standard library (*preliminary version*). TR, http://www.bsp-worldwide.org, Oxford University, 1997.

[10] J. Jájá. *An introduction to parallel algorithms*. Addison-Wesley, Reading, Massachussets, 1992.

[11] C. Joerg. *The Cilk system for parallel multithreaded computing*. PhD thesis, Massachussets Inst. of Tech., Jan. 1996.

[12] R. M. Karp, M. Luby and F. M. auf der Heide. Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, 16:517-542, 1996.

[13] A. G. Ranade. How to emulate shared memory. In *Proc. 28th Annual Symp. on Found. of Computer Science*, IEEE, 1987.

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

[15] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103–111, 1990.

[16] T. Yang and A. Gerasoulis. Pyrros: static task scheduling and code generation for message passing multi processors. In *Proc. VI ACM Int. Conf. on Supercomputing*, July 1992.