Athapascan-1: A multithreaded execution model based on data flow

Gerson G. H. Cavalheiro, Mathias Doreille, François Galilée,
Thierry Gautier, Jean-Louis Roch

LMC-IMAG-Apache Project

100 rue des Mathématiques
38041 Grenoble CEDEX 9, France

Grenoble, France
http://www-apache.imag.fr

{Gerson.Cavalheiro, Mathias.Doreille, François.Galilée, Thierry.Gautier, Jean-Louis.Roch}@imag.fr

CAPES–COFECUB Brazilian scholship
CNRS, INPG, INRIA and UJF

Abstract

We present Athapascan-1, a language implemented as a C++ library that enables the on-line computation of the macro-data flow derived from the data-dependencies of a parallel application. The parallelism is explicit but synchronization implicit. The semantics of Athapascan-1 is data driven; it is independent from the scheduling algorithm used to execute the application. 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. Then, the performance of a code (parallel time, communication and arithmetic works, memory space) can be evaluated directly from a cost model without need of a machine model.

Keywords: *Multithreading, macro-data flow languages, on-line scheduling, parallel complexity.*

I. Introduction

Recent work in the field of parallel programming has resulted in the definition of extensions of sequential languages that can be efficiently scheduled on theoretical abstract machine models. Those parallel programming interfaces are usually built on top of a standard sequential language commonly used in high performance computing; parallelism is described at an arbitrary grain (data and control) independently of a specific architecture.

Since the parallelism of an execution is directly related to the input of the program, it is generally unknown at compile time. Then the schedule is computed at run-time. On-line schedules are mainly based on a mapping of ready tasks to idle processors (greedy schedules). If neither memory space nor communication overheads are taken into account, such a schedule leads to provable performances [10]. However, in practice, due to magnitude of the ratio between local and remote memory access costs on a distributed architecture [11] and memory space exhaustion [6], some significant improvements can be brought to such a schedule by some knowledge about the data flow graph corresponding to the execution. For instance, DSC [20] and Metis [14] enable to compute efficient schedules on distributed architectures for various numerical applications from the description of the macro-data flow related to the execution.

Being based on asynchronous primitives, most parallel programming interfaces with provable performances give raise to such a knowledge.

In BSP [12] (and basically in MPI), parallelism is explicit. Instructions are grouped by blocks; each block is a sequence of instructions, either local computation or communication

and access to remote data. No explicit synchronization occurs during execution of a block; each block is delimited by a specific statement that defines a virtual global clock. Then the parallel computation is structured in macro-steps, each step consisting in the parallel execution of the blocks that have the same global clock value. Each `sync` statement is implicitly annotated by the remote access that will be performed during the execution of the corresponding block. Then the data flow between two global steps could be determined at run-time. On a theoretical point of view, this enables the implementation of an on-line data routing algorithm that achieves provable performance [19].

Similarly, the parallelism in the Jade language [17] is expressed at the level of blocks, each block corresponding to a computational task. However, synchronization between tasks is implicit and defined by the access that will be performed in a virtual shared memory. Each block is annotated by the type of access (read or write) that will be performed on any shared object involved in any of its execution (`withonly` construct). The Jade implementation uses this data access information to automatically extract the concurrency and map the application onto the machine. Restrictions on access that can be performed during a block ensure that a sequential execution of the program is valid. The resulting parallel execution preserves the semantics of the related serial program. So, the use of data access specification enables to determine at run-time the data flow between tasks in order to guide the parallelization.

Furthermore, the validity of a non-preemptive sequential schedule enables to bound the memory space required for a parallel execution [6]. In the Cilk language [5], parallelism is described by spawning a procedure call. Synchronization is explicit: the execution of instructions following a `sync` statement is delayed until all previously spawned calls – according to the sequential depth first order – are completed. Then a sequential depth first execution of the program is a valid schedule. A work-stealing algorithm, based on a greedy on-line schedule, takes benefit of this particularity. On the one hand, it bounds the memory space needed for a parallel execution with respect to the one required by a sequential execution. On the other hand, the overhead involved for the execution of a spawned call on an idle processor is amortized by replacing spawn statements on a critical path by local procedure calls without lost of theoretical performances. Cilk enables to unfold at run-

time the task precedences; however, Cilk is control-based and data dependencies cannot be predicted.

An explicit synchronization instruction bounds the on-line computation of future data flow dependencies that occur after the synchronization. This limits the on-line use of static strategies although some are of theoretical and practical interest when tasks are of known costs [20]. 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 memory access a task can perform, we exhibit a parallel imperative language, named Athapascan-1 (Ath stands for *Asynchronous Tasks Handling*) with no explicit synchronization instruction. It allows the on-line analysis of data-dependencies. Athapascan-1 is mostly inspired from Jade [17] concerning typing of memory accesses and Cilk [5] concerning the expression of parallelism. The semantics is data driven; restrictions on access ensure that a sequential depth first execution of a program is valid. The run-time implements a default order of execution with a bounded overhead both in time and space. However, the scheduling policy that computes the data and location to execute tasks is not part of Athapascan-1 runtime. It is a plug-in and it is completely independent both from the application and Athapascan-1. So, tuning the schedule with an application for a specific architecture could eventually be performed by code annotation.

The reminder of this article is structured as follows. In section 2, we introduce the syntax and semantics 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. This extends the result in BSP, Cilk and Jade, that require migration of tasks. Particularities of the implementation, that uses low level local multithreading and high level scheduling policies are detailed in section 4; the overhead related to the management of the data flow is evaluated by experimental measures on a distributed and on a shared memory architecture. The last section presents experiences with different applications programmed with Athapascan-1 and their performances on various parallel and distributed architectures.

II. The Athapascan-1 language

A. Overview

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

The Athapascan-1 semantics rely on shared data access and ensures that the value returned by a read statement 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 semantics is motivated by its direct readability on the program source: Fig. 1 shows an obvious example. This also favors a direct reuse of existing sequential code.

The respect of the access semantics 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 access performed on shared objects: `R` stands for *read* (`shared_R`), `W` for *write* (`shared_W`). 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 the task `print(a)` in the lexicographic order. The execution date of `print(a)` is delayed until `update(a)` resumes. The program will thus print the value 5.

Athapascan-1 is implemented as a C++ library; in order to avoid typical C++ constructions, the codes of the examples in this paper use a simple syntaxic extension of C++ handled by a preprocessor.

B. Syntax and abstract representation

B.1 Tasks and closures

A task corresponds to the execution of a procedure which declaration is similar to a C procedure declaration having simply the `void` returned type replaced by the `ath` keyword:

```
ath user_task( [parameters] ) { [statements] }
```

```
ath update( shared_W< int > x ) {
  x.write( 5 );
}
ath print( shared_R< int > x ) {
  printf( "%d", x.read() );
}
ath sample() {
  shared< int > a;
  fork< update >() ( a );
  fork< print >() ( a );
}
```

Fig. 1. Example of basic Athapascan-1 constructs.

More precisely, a task corresponds to the execution of a C++ function object [18], *i.e.* an object that gets only one method overloaded by the typed operator '`()`' and that returns no value.[1]

Prefixing any call to such a procedure by the keyword `fork` creates a task:

```
fork< user_task >() ( [parameters] );
```

This statement creates an object, called a *closure*, that is given to the scheduler and returns for continuation (asynchronous task creation). A closure is a data structure that contains an instantiation of the user procedure (that defines the method to run) and the list of its effective parameters. A closure is *ready* if all this effective parameters with read access are ready or *waiting* if some argument that will be read is not ready. A parameter is not ready for a task if and only if this task has a predecessor (with respect to the lexicographic order) not yet completed 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 stores a copy of them; and second, the parameters which are references to a version of a shared data.

[1]For instance, in Fig. 1, the declaration of procedure `ath print(...) { .. }` is written in C++: "`struct print { void operator() ( shared_R<int> x ) {` *body* `} }`".

B.2 Shared data and their versions

The shared memory, that allows 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 by the algorithm. A sequence of *versions* is associated to each shared data: each shared data version in the sequence represents the value of the shared data at a certain instant of execution. A declaration of a shared data creates an object, called *evolution*, that contains pointers towards two *transitions* which manage the allocation, the state and the access to 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();
const T& read();
void write( const T& );
void cumul( const T& );
```

B.3 Example

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

Fig. 3 shows two codes for computing the kth Fibonacci number. The first version is the schooler recursive procedure. If the threshold is reached, the `fibo` task writes the result into the shared data `res`. Else the execution of “fibo(n)” creates two tasks for the computation of “fibo(n-1)” and “fibo(n-2)” and a third task `sum` for the addition into the result `res`.

This summing task is delayed until the two Fibonacci tasks “fibo(n-1)” and “fibo(n-2)” have been completed, due to read the access. The second version is a cumulative version

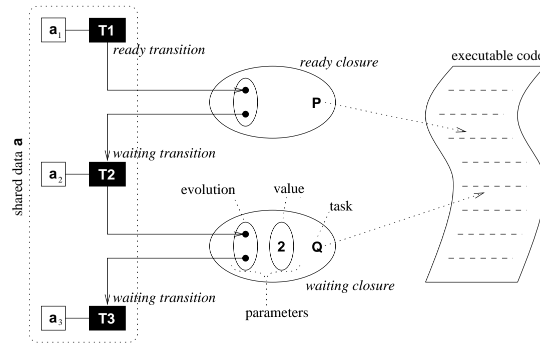

Fig. 2. Internal data structures.

of the "sum tree".

```
ath sum( shared_R<int> x, shared_R<int> y,
         shared_W<int> z ) {
  z.write( x.read() + y.read() );
}

ath 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 );
 }
}
```

(a)

```
struct add {
  void operator() ( int& x, const int y ) {
    x += y;
  }
};

ath 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)

Fig. 3. Two versions of code to compute the nth Fibonacci number: (a) recursive (b) cumulative.

C. Lexicographic ordering and parallelism

In order to respect sequential consistency (lexicographic order semantic), Athapascan-1 has to identify for each read performed on a shared data the related data version. However, parallelism detection is easily possible in that context if all tasks precise the shared data objects accessed for 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 are located in the parameter list).

C.1 Access right: evolution of shared data versions

In the declaration of the formal parameters of tasks, the references to shared data version are typed by their *access right*, *i.e.* what kind of access is allowed to perform on the shared data. These rights are `R_W` for read&write modifications, `W` for writing, `R` for read only and `CW<f>` for accumulation only.

C.2 Concurrent access and accumulation.

The read only – `R` – and accumulation only – `CW<f>` – access rights enable tasks to perform a concurrent accesses on a same shared data version. The accumulation law is user defined: the function `f` is supposed to be associative and commutative. Mix of different law is not allowed on a same version; a new version is then created. The initial value is the previous value of the shared data according to the lexicographic access order.

C.3 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 "postponed" (access right suffixed by `p`) if the task will not perform any access on the shared data but creates tasks that will.

III. Data flow building and cost model

Adding restrictions on accesses that can be performed by a task in shared memory, 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.

A. 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 G of the shared data versions access. G is a bi-partite graph with vertices corresponding respectively to the closures and to the transitions; an edge denotes access to a shared data version. G gives at each instant a partial description of the data flow dependencies that will occur till the end of program.

The evolution of this graph takes place at each task creation or shared data declaration (addition of edges and/or nodes) or task termination (removing of edges and/or nodes, node's state evolution), as shown in Fig. 4.

This graph is maintained by the system: the scheduling policies could take benefit of it. Without adding overhead it provides some relevant information on the application to schedule.

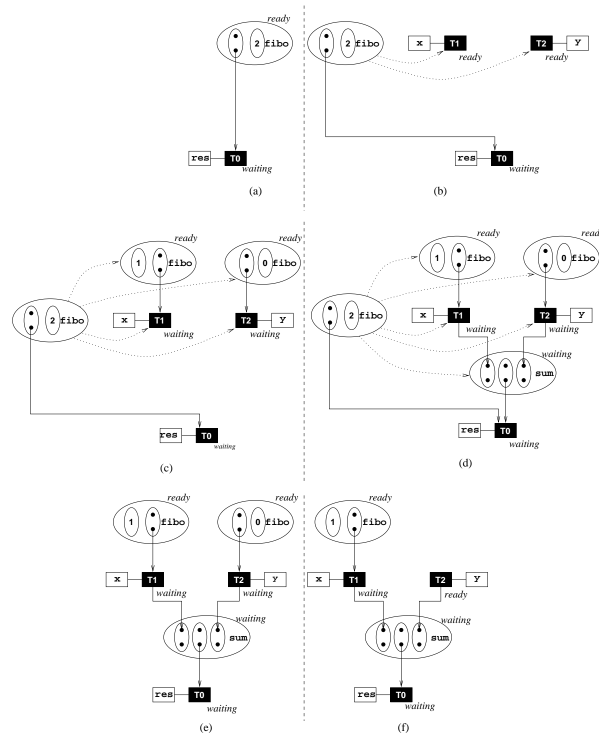

Fig. 4. Dynamic evolution of the shared data versions access graph for the recursive version of `fibo(2)`: (a) initial state, (b) after the creation of the two shared `x` and `y`, (c) after the creation of the two `fibo` tasks, (d) after the creation of the `sum` task and just before the end of the `fibo(2)`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

B. Bounding cost of Athapascan-1 statements

Due to the access right and the lexicographic order, the semantic of read statements is easily determined in 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 updates and task creations are local (need no communication). This is discussed in section IV-C.

• $\mathcal{C}_2$: All shared data versions that can be read by a task are always ready at the beginning of the task execution and until its completion.

It means 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 resuming of the new 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.

The tasks which can directly write a shared data are not allowed to create tasks that would read this shared data: else a copy of the last written value should be stored for the created task, or the creator would stop until the created one completes 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 $\mathcal{C}_3$.

Conditions C_1, C_2 and C_3 are ensured by restrictions, at task creation, on the allowed conversions of shared data version types. Those restrictions are summarized in the table Tab. I. Note that the strong typing of access in shared memory enables the verification of the correctness of a program at compile time.

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

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>

TABLE I

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

modification is made with constant time and space.

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

Lemma 2: There is no precedence relation between a task and the task that creates it. This results from shared types and condition $\mathcal{C}_2$. □

This property enables 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.

C. Athapascan-1 reference order

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 a block of statements B can be represented by a word $t = s_1 F_1 s_2 F_2 \ldots s_{n-1} F_{n-1} s_n$ (s_1 and s_n can be empty). The independence between the created task and the creator task (lemma 2) implies that this trace is semantically equivalent to the trace $t' = s_1 s_2 \ldots s_{n-1} s_n F_1 F_2 \ldots F_{n-1}$. 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. 5 illustrates this execution order (note that semantics of `push` and `pop` operations on the `deque` respects the FIFO order).

```
ath t ( [parameters] ) {
  stmts_1;
  fork< t_1 >() ( [args_1] );
  stmts_2;
  fork< t_2 >() ( [args_2] );
  stmts_3;
}
```

(a) Depth-first sequential order

```
ath t ( [parameters]) {
  deque d;
  stmts_1;
  push( d, [args_1] );
  stmts_2;
  push( d, [args_2] );
  stmts_3;
  fork< t_1 >() ( pop( d ) );
  fork< t_2 >() ( pop( d ) );
}
```

(b) Reference sequential order

Fig. 5. Two semantically equivalent programs.

Proposition 1: The reference sequential order of execution respects the semantics 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 explicit. □

All the 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 to evaluate the cost of a program directly from the code.

D. Cost measures

Proposition 1 enables non-preemptive schedules: the execution of a closure is delayed until its parent (the task that created it) resumes. Following notations [2], [6], costs are defined from this reference order: sequential time T_1 and space S_1, arithmetic depth T_∞, communication volume C_1 and delay C_∞.

More precisely, we define those costs on the trace of the execution. As seen in §III-A, this trace can be represented by a bipartite DAG $G = (G_t, G_d)$ (see Fig. 4), with node sets G_t and G_d corresponding respectively to tasks (oval nodes in Fig. 4) and shared data versions (rectangular nodes in Fig. 4). G is a weighted graph: in G_t, each task node is weighted by its computation cost, related to the execution of the body; in G_d, each data node is weighted by the size of the data. Similarly, the edges are weighted by the

size of the shared data for a direct access and by a unit cost for a reference or postponed access. Note that G is usually unknown until execution completes; however, any execution of a correct deterministic program on a same input will be related to the same graph G. Moreover, we assume that the weights are also independent from the execution. Then G is independent of the scheduling policy used to perform the execution on a bounded number of processors.

D.1 Sequential time T_1 and space S_1.

Sequential costs are defined from a serial execution of the program following the 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, S_1 related to program `fibo` in Fig. 3(a) is $\Theta(n)$ which is the space required by a depth-first execution too.

D.2 Parallel time T_∞.

Arithmetic depth T_∞ (or *parallel time*) is the length of a critical arithmetic path in G_t. Then, T_∞ is a lower bound of the minimal time required by any schedule on an unbounded number of processors ignoring communications times.

D.3 Sequential volume and delay.

Communication delay C_1 and work C_∞ are evaluated similarly from G taking into account only weights of edges in G. C_∞ is the length of the critical path in G_d; C_1 is the sum of the weights over all edges in G.

D.4 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.

E. Provable execution time bounds on a DCM

On a distributed architecture with p identical processors, denoted by DCM(p), the parallelism of an Athapascan-1 program has to be bound: tasks and shared data versions are to be scheduled on the p processors. For a general program, neither the shape of the data flow graph nor the time of its tasks is known before the execution completes. The schedule is computed at run-time. Most on-line scheduling algorithms are based on a greedy strategy [10]: when a processor becomes idle, it gets a ready closure if any and performs its execution.

To bound the delay of remote accesses, the algorithms proposed in [13] are used to emulate a global memory on the p processors with the help of universal hashing functions. In order to obtain near-optimal emulation (constant or very small to p delay), a multithreading technique, often referred as "parallel slackness" [16], [19], is often used. It consists in emulating preemptively several virtual processors on each processor of the architecture. In the following, we assume that q is the number of virtual processors emulated on the DCM(p) and that the resulting delay (related to any remote access performed by one of the q virtual processors) is bounded by h.

Then, an execution of an Athapascan-1 program has to be scheduled on the q virtual processors. Applying the greedy scheduling strategy proposed in [4] in order to bound the memory space, the next theorem gives time and space bound for any execution of an Athapascan-1 program including the cost of the scheduling algorithm.

Proposition 2: Let M be a distributed architecture with p identical processors that emulates a global memory on q virtual processors with a delay h. Then, any Athapascan-1 program can be executed on M in time

$$\frac{T_1}{p} + \frac{q}{p}T_\infty + h\left[\frac{q}{p}C_\infty + \frac{C_1}{p} +\right] + h\frac{q}{p}O(\sigma).$$

This time includes the overhead due to the computation of the schedule which requires no migration of running closures.

Proof. Let G be the graph related to the execution. The schedule is performed on $q(p)$ virtual processors. To compute the schedule, a data structure s is stored in the global memory which access is protected by a lock e. This structure contains both the graph G

– transitions and closures that are not yet completed – and two lists, R_t of the ready tasks and P_i of the idle virtual processors.

G is direct and stored as depicted in Fig. 4 but with additional backward pointers. So, each task in G has backward pointers to the shared data versions it reads and forward pointers to the shared data versions it writes. Similarly, each shared data version has a pointer to the tasks that write it (may be several in case of accumulation) and to the ones that read it. Furthermore, two counters are related to each data version that indicate the number of tasks that can respectively write and read the data; similarly, a counter is related to each task that indicates the number of shared data version it is waiting for reading. Besides, during its execution, a task may create new tasks by executing `fork` statements. Those tasks are linked with respect to the reference order. Moreover, a task has a backward pointer to its creator if it is yet in G. All changes to s are performed in mutual exclusion[2] using the lock e. They occur at task creation and or at shared data version declaration or evolution. The total number of those changes is clearly bounded by σ. When a task becomes ready (related counter becomes 0), it is added in constant time in R_t which is maintained sorted according to the reference order.

The schedule is computed as follows. Each time a new task is added to R_t, it is assigned to an idle processor which is removed from the list P_i. When a processor completes a task, it takes a new ready one in the list R_t if any. If R_t is empty, the processor is added to the list P_i of idle processors.

Let T denotes the execution time on the DCM(p) using this schedule. We assume that, on each processor, $\frac{q}{p}$ virtual processors are executed preemptively at the grain of the elementary machine operation. Let $T' = \frac{p}{q}T$; for convenience, elementary tops are assumed to range in $\{1, \ldots, T'\}$. Following [10], T' is bounded by the characterization of a subset of tasks in G that are on a critical path. The tops in $\{1, \ldots, T'\}$ are partitioned into three disjoint subsets:

- A (active computation): at each top in A, all virtual q processors are executing an instruction of the application (including access to a remote data).
- O (schedule overhead): at each top in O, a schedule operation (either graph management

[2] Note that, due to access restrictions, most modifications are local: thus, they do not require a global lock to be implemented.

or virtual processor assignment) is being performed on one virtual processor.

- I (idle time): at each top in I, the list P_i is not empty but no schedule operation is being performed.

Then, we have $T' = \#O + \#A + \#I$ which leads to the following inequality:

$$pT = q.T' \leq q(\#O + \#A + \#I). \quad (1)$$

Clearly, $\#A$ is bounded by $\frac{T_1 + hC_1}{q}$. When the lock e is free, the time required to acquire it is bounded by $O(h)$. When e is not free, a processor is performing an operation in O (eventually taking or releasing the lock). Then the numbers of tops in O is bounded by $\#O \leq O(h\sigma)$. Besides, $\#I$ is bounded as in [10]; when a processor is idle and no processor is executing a schedule operation, there exists a task on a critical task that is being performed on a processor. So, $\#I$ is bounded by $T_\infty + hC_\infty$. Replacing in (1) concludes the proof.

□

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 [6] 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 $O(hT_1/p)$: efficiency depends heavily on h.

Bounding the space. The previous schedule executes task with respect to the reference sequential order. Let S_1 denotes the space required by this sequential schedule. The result in [4] leads to bound the space required for the execution on DCM(p) with respect to S_1. 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 $S_1 + O(qT_\infty)$ [4].

IV. Distributed implementation

In this section we describe the distributed implementation of Athapascan-1. At compile time, correctness of the program is verified using the strong typing of accesses that will

be performed. For each fork statement, code is generated to create the corresponding closure. Then the closures must be scheduled to be executed on a processor. Two levels of scheduling are distinguished (Fig. 6).

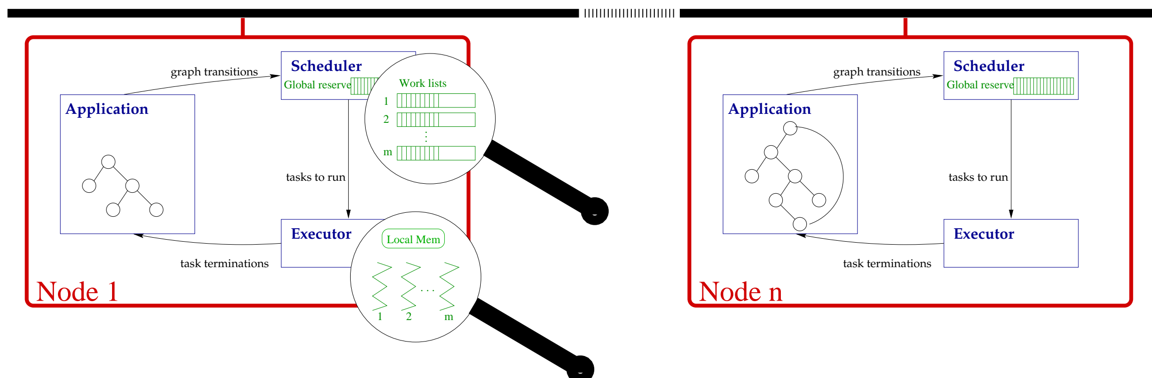

Fig. 6. Two levels scheduling: at a high level, a global scheduling manages the load distribution between nodes while, at a lower level, a local multithreaded kernel optimizes the use of each processor. The zoomed regions show details of the lower level scheduling.

A high level scheduling distributes the load among the nodes of the parallel architecture. From the knowledge of macro-data flow dependencies, various schedules can be implemented, from greedy to static ones.

A low level scheduling, on any particular node, overlaps part of system overheads by effective computation using multithreading. Each node of a parallel machine emulates a certain number of virtual processors (*threads*) that share a single address space. In theoretical works (see section III-E), this technique, often referred as "parallel slackness" [19], enables to provide near-optimal emulation of a global memory on a network of identical processors [13]. On a practical point of view [9], various experimentations show that a certain amount of local multithreading enables to hide part of communication delays. This low level local scheduling is implemented by a multi-threaded, portable, parallel programming runtime system, named Athapascan-0 [7].

A. Athapascan-0: an efficient and portable integration of communications and multithreading

Athapascan-0 allows remote creation of threads and provides communications and synchronization facilities. The local scheduling of threads in a node is managed by the runtime

in order to hide communication latencies. Athapascan-1 is implemented on this runtime and provides the global scheduling of closures (tasks) on the whole architecture.

Athapascan-0 is designed for a general parallel machine that consists of a network of symmetric multi-processors (SMPs) that offers the four following types of exploitable physical parallelism: between processing nodes, between computation and communication processors within a node, between computation processors within a node and between communication on disjoint routes. Athapascan-0 is built on top of standard, unmodified, message passing and threads libraries. Much care has been taken on communication efficiency and communication overlapping with computation.

An Athapascan-0 parallel machine is composed of a set of nodes executing the Athapascan-0 runtime. The operators that express a parallel computation are: local and remote thread creation; local and remote communication between threads; local synchronization between threads; and remote access to shared memory regions.

B. Athapascan-0 and the implementation of Athapascan-1

Athapascan-0 is the system layer that allows practical implementation of parallel slackness in an effective distributed architecture with p processors that may be heterogeneous. In the implementation of Athapascan-1, a certain number of Athapascan-0 threads are created on each processor in order to emulate identical processors. So, a scheduling algorithm may assume that the machine is a set of q identical virtual processors that communicate with a bounded delay h. This practical emulation is inspired from the theoretical one presented in section III-E to perform scheduling on a distributed architecture, DCM(p), with p identical processors connected by a network.

C. Shared data versions access graph management

With the help of Athapascan-0, 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 access to the shared data versions or the tasks creations are local events and make no communication. The time required for a task creation is therefore proportional to the

number of its parameters.

In order to detect termination of access to a transition in a distributed asynchronous environment, a classical termination algorithm is implemented [15]. Each transition is associated to a master node that computes a balance between increasing counters related to each of its replicates. Those counters are managed locally on each site that possesses a replicate. When there is no more local access on the site, the values of local counters are sent to the master node.

D. Annotating a program by a scheduling algorithm

The distributed implementation of Athapascan-1 ensures that the semantics of the program is respected whatever the global scheduling algorithm is. To be used as a plug-in, a scheduling algorithm has to be implemented as the specification of a virtual class. Then, Athapascan-1 allows the user to specify the schedule used at task creation by code annotation at creation of a task:

```
fork< user_task > ( sched, attributes ) ( [parameters] );
```

The scheduling attributes specifies the global scheduling to be used; the task `user_task` will be scheduled by the scheduler `sched` and `sched` will also be the default algorithm to schedule the closures created by `user_task`. Additional attributes (priority, computational cost and locality) can be exploited by `sched`.

Schedule attributes of a task are stored in its closure. After creation (`fork`) and until it is completed, the state of the closure depends on the graph of evolutions. The possible

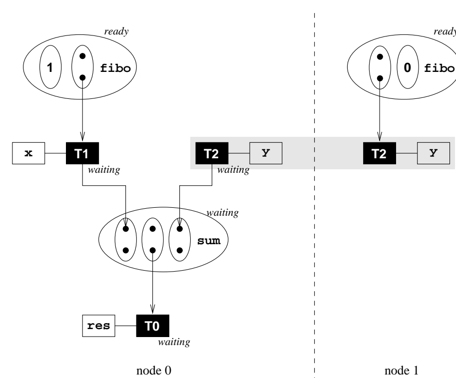

Fig. 7. Replicated transitions on a distributed graph. When `fibo(0)` is completes, a message is sent from node `1` to `0` to warn the transition `T2` that no more writers exist on `1`.

states are:

- *waiting*: the closure has at least one predecessor not yet completed;
- *ready*: the closure can be executed;
- *running*: the closure is running sequentially its instructions;
- *executed*: the closure is completing.

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 informations that it needs about the current global state of execution: for instance, closure attributes, parameters, state or data dependencies.

E. Examples of implemented scheduling policies

Different scheduling algorithms have been implemented in Athapascan-1Ẇe present here two basic schedules that we have used to evaluate the performances of Athapascan-1: a greedy distributed algorithm (which is the default one) and a static algorithm that uses the knowledge of the macro-data flow graph.

E.1 Greedy schedule

Classical scheduling algorithms to minimize completion time on a given number of virtual nodes are based on a greedy strategy: when a node becomes idle, it steals a ready task available on a non-idle processor in order to execute it. As seen in section III-E, such a strategy achieves provable time performances but, in order to prevent for memory exhaustion, some precautions have to be taken [6]. In Athapascan-1 on each processor, tasks are ordered with respect to the reference sequential order which is computed in constant time. However, it can be overloaded by assigning priorities to each task at creation. Then, scheduling tasks with respect to this order allows to bound the memory space with respect to the one required by such a sequential execution [4].

Such a strategy has been implemented in the Cilk parallel language where the lexicographic order is used both to bound the memory space and to provide efficient sequential execution for dynamic series-parallel graphs; it has lead to very good experimental performances [1] on parallel architectures with uniform memory access.

Similarly to Cilk, the high level scheduling implements a work-stealing algorithm. On

each node of the architecture, a list of ready tasks is managed. Each node uses it to store tasks that become ready. When a node completes a task, it picks another ready task from its local list to execute it. When the size of its list reaches a minimal threshold, a thief procedure is trigged: a *victim* node is randomly selected and a task is *stolen* from its reserve list. Moreover, if the size of the victim reserve list is null, the stolen message is forwarded to another victim.

In order to optimize the memory space, the management of the reserve on each node is not symmetric. Each node adds new ready tasks on the head of its reserve. Then, when it completes a task, it takes a new one also from the head, privileging a depth traversal of the task graph. However, tasks are stolen by other processors from the tail of the list. Especially for recursive programs, this strategy enables to bound the memory space required on each processor [6] while not increasing the critical path.

Moreover, when a task t is stolen by an idle node P_I on a non-idle one P_S, all successors of t located on P_S will wait for the completion of t on P_I to become ready. In order to avoid extra communications related to t completion, it thus can be worth fully to export not only t to P_I but also some of its successors (at least all successors of t that have no other predecessor on P_S except t). This strategy tries to reduce the number of three kinds of communications:

- Tasks state management: since the graph is distributed, immediate successors of t that are waiting for its completion will be locally updated.
- Data transfers: data updated by t that are required by its successors already in P_I can be accessed directly on the local memory cache of P_I with no communication.
- Work-steal messages: it can be expected that when t will be completed on P_I, one of its successor will be ready on P_I.

E.2 Static algorithms

At a given instant of the execution, the evolution graph represents all the already created closures and their dependencies. Each closure being assumed to correspond to a sequential algorithm, the related macro-data flow graph then represents the execution. Its annotation by cost information enables the use of static strategies.

Since the runtime allows the global scheduling to have access on those dependencies,

we have been able to use the scheduling algorithm DSC, implemented in the parallel programming environment Pyrros [20], as a plug-in.

An additional information is required by the scheduling to trigger the computation of this schedule, which is performed sequentially on one node. Then each tasks is assigned to a node; the order of execution of the tasks on each node is managed by Athapascan-1 runtime.

F. Overhead of the distributed implementation

TABLE II

INFLUENCE OF THE LOCAL SCHEDULING OF THREADS ON GLOBAL GREEDY ON-LINE SCHEDULING.

	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

To analyze the overhead related to the distributed implementation of Athapascan-1 and the overhead related to local multithreading, we present in table II some performance results (time in seconds) obtained from the execution of the Fibonacci program (Fig. 3(a)) for an input value of 40, using a threshold of 25 to halt the task creations ≤ 25; then, roughly half of the generated closures require less than a micro-second to run to completion and the other ones (that sequentially compute fibo (25 or 24)) a few milliseconds; $\#N = 32000$ tasks are generated producing about $\#E = 100000$ edges. The experiments were achieved on five architectures: two mono-processors, a IBM RS-6000 (AIX 4.2) and a Pentium (Solaris 2.5.1); two distributed architectures: a four nodes IBM-SP architecture (IBM RS-6000/370, AIX 4.2, IBM-MPI) and a four nodes network of workstation - NOW (Pentium, 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. When several

nodes are used, the default greedy high-level scheduling is used.

Comparing the performances of the pure sequential algorithm with the sequential version for the Athapascan-1 code, we verify that the overhead introduced in Athapascan-1 is small. This is not true in the parallel version where we can observe the overhead produced by the scheduling policy and graph management. This overhead is partially overlapped when the number of threads on each node is increased, though. 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 35.

G. Analysis of the overhead and amortizing

Most of the overcost of the execution of Athapascan-1 program is due to the closure creation (`fork` statement). The dynamic memory allocation and the graph management are costly related to a function call. Following [5], this cost can be amortized using a lazy scheme in the implementation of a work-stealing algorithm. However, in order to not increase the critical path while avoiding migration of running closure, the serialization of tasks is performed according to the reference sequential order.

At runtime, we split the graph management of forked tasks into two parts. The one corresponds to closure creations; the other to updates of the data flow graph when a request of work is received from an idle processor. At each `fork` statement, we copy into a stack the related closure. Then, when the tasks completes, the next closure (according to the reference sequential order) is executed locally as a function call.

When a request of work steal is received, the data flow graph is built from the local shared data and closures into the stack. Under assumption that they are few work steal requests compared with the number of created tasks – which is verified for programs having small T_∞ –, this strategy could be used to achieve parallel execution with a cost closed to pure C++ sequential execution.

Table IV-G presents the times to execute the same program Fibonacci (described in Fig. 3(b)) with input $n = 10$ and $n = 20$ at the finest grain: recursive splitting is not stopped (Pentim II, Solaris 2.6, Posix threads). The first column shows the running time of the program in one thread one node machine using the parallel runtime presented in the previous section; the second column implements the reference sequential order as described

TABLE III

Times in second to execute the program Fibonacci with different runtimes and inputs.

input	Parallel runtime	Reference sequential order	Pure C++ function
10	0.028	1.5e-4	11.4e-6
20	37.2	18.3e-3	1.4e-3

in this section. The last column gives the time obtained by an equivalent C++ program where `fork` statements are replaced by function calls and shared objects implemented by pointers. The ratios of the time per task creation are for input 10: 1.6e-4 for the parallel runtime, 8.5e-7 for the reference sequential order and 6.4e-8 for the pure C++ program. For input 20, these ratios are respectively: 1.4e-3, 8.4e-7, 6.4e-8.

The reference sequential order can then be implemented with a constant overhead of about 10 with respect to a pure C++ function call. The improvement expected with respect to the parallel implementation is within a factor of about 10000 in the best case for the creation of an elementary task.

V. Experimental evaluation

In this section, we present experimental performances of Athapascan-1 on two groups of applications:

- Recursive algorithm: the computation of the Mandelbrot set and the resolution of the eight queens problem.
- Numerical algorithm: the LU factorization of a matrix using Gauss elimination.

For each example the scheduling policy used is described and the obtained performances commented.

A. *Recursive parallelism*

Recursive algorithms leads to data flow graphs with small critical path. Then, the greedy schedule presented in the previous section, that privileges a depth-first traversal of this graph, should achieve good performances.

A.1 Mandelbrot

In this section we present some performance results obtained for a Mandelbrot computation. The Mandelbrot algorithm implemented is recursive: it takes in input a start region and a threshold. While the threshold is not reached, the region is splinted in four sub-regions; otherwise, the sub-region is sequentially computed. For each experiment, 314 tasks were generated: among those, 256 tasks correspond to the sequential computation of a sub-region (with size lesser than the threshold) each of unknown computational cost; the others compute the recursive splitting.

The Fig. 8 shows a distribution of work obtained after the execution of a Mandelbrot computation. The threshold for halting the recursive splitting is fixed: it corresponds to the size of the smallest squares in the image on the right. Note that in this experiment, 256 additional tasks are executed by the node 0 (the dark-grey one in Fig. 8) to display the plotting; it can be noted that this processor among the five is the one that computes the smallest surface. Besides, as it can be seen on the right image, only few fine grain tasks are exported to other processors (except at the end of the execution). This is related to the implementation of the greedy scheduling described in section IV-E.1; it privileges local depth first execution. As it appears on the right image, this enables to bound the space of the distributed execution and to avoid unnecessary steal messages.

A.1.a Performances of the two levels scheduling . To evaluate the speed-up, plotting tasks have been suppressed; only the 314 tasks corresponding to recursive splitting and computation are executed. Experiments have been performed on two different architectures: a SMP machine with 4 Pentium Pro (Solaris 2.6, Posix threads) (performances are presented in the table on the left of fig. 9) and a IBM-SP (IBM RS-6000/370, AIX 4.2, IBM-MPI, kernel threads) using from 1 to 6 monoprocessor nodes (performances are presented in the graphic on the right of Fig. 9). On both machines, speed-up (sp) has been computed by: $sp = T_1^1/T_m^p$, where T_m^p is the time obtained for the parallel execution on a p node parallel machine using m threads by node; then T_1^1 is the time for the execution on an one node machine with only one execution thread.

Observing the given measures on the SMP, we deduce that local multithreading enables to overlap the system overhead: a speed-up 3.6 is obtained which is near to the number

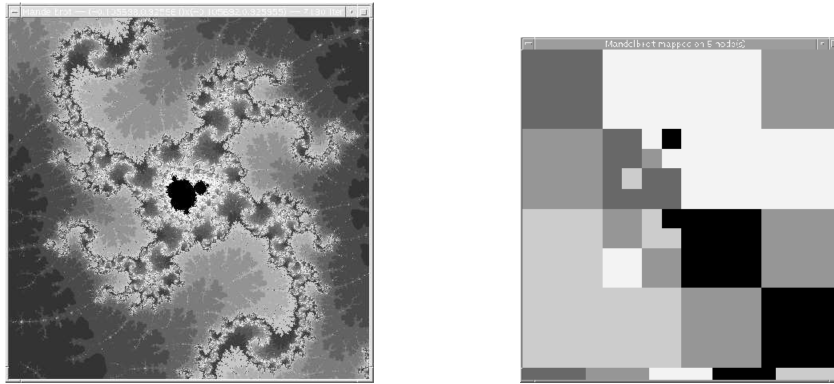

Fig. 8. A Mandelbrot plotting and the related distribution of the work on a 5 node parallel machine. The algorithm is recursive: it cuts the original region in four sub-regions until a fixed threshold (which corresponds to the size of the smallest squares on the right). On the right image, each color corresponds to the specific node that has computed the corresponding area in the Mandelbrot set on the left; the line on the bottom of this image depicts the five colors, each one corresponding to a node. The processor that starts the computation and performs the display of the Mandelbrot set is the left-most one (indexed 0 and colored dark-grey).

# threads	T_m^1	sp
1	23.80	1.0
2	12.04	1.9
4	7.71	3.0
6	7.22	3.3
8	7.02	3.3
10	6.68	3.6
12	6.74	3.5
14	6.73	3.5
16	6.74	3.5

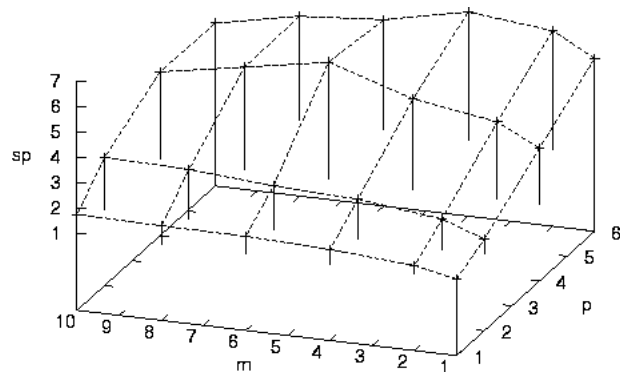

Fig. 9. Performance measurements for Mandelbrot on 2 architectures. In left side, a table presents the time (in seconds) and the speed-ups obtained on a 4-processors SMP architecture. The graph in the right side presents the speed-ups on a IBM-SP, using from 1 to 6 nodes.

of available processors (4). Moreover, the basic time T_1^1 is very close to the cost of a pure sequential execution (with no parallelism overhead) on this architecture.

On the IBM-SP also, the system overheads (including the one related to the high level

work-stealing when $p \geq 2$) are partially overlapped by using local multithreading. In this experiment, it can be seen that the best value for m depends on p. Besides, a good speed-up (about p with the best value for m) is obtained related to T_1^1. However, a large overhead is introduced here with respect to a pure sequential computation on one node (a factor of about 3), mainly due to the polling of communications; this overhead is recovered when using several computational resources ($p \geq 4$).

A.2 Queens

The goal of this famous problem is to place n queens on a chess board, of dimension $n \times n$, so that none is in a position to take another one. The implemented parallel algorithm computes the number of positions; it creates tasks recursively to explore the searching tree of the problem. As input of each task is given the chess board with the already placed positions. The algorithm uses a compressed representation of the chess board of size $5n$ instead of n^2. The parallel program takes as input the number n defining the size of the problem and a threshold to stop the recursive generation of tasks.

The tables IV and V show times for the execution of the n-queens problem on 3 parallel architectures with 3 problem sizes: $n = 13, 14$ and 15, using a threshold of 3 on each case. The given times are the average of ten executions for each case (aberrant values have been deleted). The effective number of generated tasks are given as additional information. The scheduler chosen to execute this experience uses the same depth first greedy algorithm (section IV-E.1) employed for Mandelbrot plotting.

TABLE IV

Times in seconds on the IBM SP machine for n-queens problem with 3 problem sizes.

		IBM SP						
size	# Tasks	Sequential	1 node	2 nodes	4 nodes	8 nodes	12 nodes	16 nodes
13	1178	36.66	38.04	20.03	9.28	5.06	2.92	2.40
14	1537	223.13	225.34	113.95	57.01	28.44	18.65	14.65
15	1964	1451.92	1450.74	779.39	364.84	188.41	130.80	91.60

The table IV presents the times on a dedicated IBM-SP distributed architecture (IBM RS-6000/370, AIX 4.2, IBM-MPI, kernel threads) using from 1 to 16 mono-processor nodes. On each node there is only one thread dedicated to execute the tasks generated by

TABLE V

TIMES IN SECONDS ON A SMP ARCHITECTURE AND ON AN HETEROGENEOUS NOW FOR n-QUEENS PROBLEM WITH 3 PROBLEM SIZES.

		SMP		Heterogeneous NOW with $p(q)$: p nodes adn q processors							
size	# Tasks	Sequential	Parallel	Sequential	1(2)	2(4)	3(8)	4(9)	6(11)	8(13)	10(15)
13	1178	6.51	1.83	3.91	2.50	1.61	1.31	1.25	1.60	1.36	1.14
14	1537	39.50	11.40	23.78	14.19	8.37	6.60	6.25	5.02	4.84	4.96
15	1964	255.18	70.56	153.57	91.55	51.67	34.90	38.80	27.73	25.24	23.51

the program.

The performances show a good utilization of the machine, obtaining speed-up close to the optimal expected. The results in table V present the performances obtained with an SMP architecture (4 Pentium Pro, Solaris 2.6, Posix threads) and under a not dedicated heterogeneous network of workstations (NOW), composed by 10 Intel based nodes with different frequencies and memory capacities (two bi-processor nodes, one four-processor node and seven mono-processor nodes running Solaris 2.6, MPI-LAM, Posix threads). On the SMP, eight threads are used to explore the parallelism of the program; as on the IBM-SP architecture, the performances show values near the optimal.

To get the performances on heterogeneous NOWs, we have used different configurations, composing sub-networks with different number of processors in order to analyze the evolution of the performance. Each subnetwork with p nodes is always build with the faster machines among the ten. On each node there are three threads dedicated to execute the tasks generated by the program. From the results obtained, in most cases the performances are good even if better on the IBM SP architecture. We attribute this to the ratio between communication and computation costs, because performances are improved when more computational work is added.

B. Numerical computing: The Gauss elimination

In this section, we present a two-dimensional block LU factorization as example of a numerical algorithm; the matrix to be factorized has size n and is partitioned in $(\frac{n}{k})^2$ blocks of dimension $(k \times k)$. The factorization is achieved on these matrix of blocks using the classical Gauss elimination algorithm where the scalar operations are replaced by block

operations. The sequential complexity for this algorithm is $\Theta\left(\frac{2}{3}n^3\right)$.

The code of this algorithm in Athapascan-1 is the direct implementation of the sequential two-dimensional block LU factorization at the grain of a block. The Fig. 10 shows this code. Tasks consist in block computations: factorization, multiplication or triangular system resolution. The parallel time is then $T_\infty = \Theta\left((\frac{n}{k})^3 + nk^2\right)$. Since all the tasks have a fixed number of arguments, the cost to build the data flow graph is bounded by the number of tasks, i.e $\sigma = \Theta\left((\frac{n}{k})^3\right)$. The volume of communications is bounded by $C_1 = O\left(\frac{n^3}{k}\right)$.

By specializing the schedule on a distributed architecture, the number of communications can be reduced without increasing the parallel arithmetic time if the nodes are identical. A classical schedule is based on a bi-dimensional cyclic mapping of the blocks of the matrix (e.g., ScaLAPACK [8]). Equivalently, each task can be indexed by the indices (i, j) of the block it updates. Assuming that $q^2 = p$ nodes are available, the task (i, j) is mapped on the node $(i \bmod q)q + (j \bmod q)$. The resulting total amount of communications is reduced $O(n^2\sqrt{p})$ on a parallel machine with p nodes. To implement this schedule in Athapascan-1, each `fork` is annotated by the indices of the related task and a bi-dimensional cyclic mapping scheduling policy is used.

The tables VII and VI show performances of this scheduling policy on various architectures with identical nodes. The SMP architecture is the same than the one used in the previous example (n-queens problem); performances are presented in table VI. The distributed architecture used in table VI is an homogeneous NOW (2 bi-processor nodes, Pentium II, Solaris 2.6, MPI-LAM, Posix threads); on this architecture, each node has five threads dedicated to execute the tasks generated by the program.

On the SMP architecture, good performances are obtained. This proves that the parallelism of the algorithm is well exhibited. However, performances are poor on the distributed architectures where the ratio between communication and execution costs induces bad speed-ups. We attribute this to the overhead introduced by the current version of Athapascan-1 in the distributed management of the transitions: the master node site related to a transition remains unchanged during the execution while it could be changed to the node where the corresponding data is updated.

```
ath Factorization_LU( shared_R_W<matrix<double> > a ) {
        // Sequential LU factorization of block a ...
}

ath M_TimesInverse_U( shared_R_W<matrix<double> > m,
                      shared_R<matrix<double> > u ) {
        //  Sequential computation of "m = m * Inverse(up)"  where up
        //  is the upper triangular part of u ...
}

ath Inverse_L_Times_M( shared_R_W<matrix<double> > m,
                       shared_R<matrix<double> > l ) {
        // Sequential computation of "m = Inverse(low) * m" where low
        // is the lower triangular part, with unit diagonal, of l ...
}

ath MinusTimesEqual( shared_R_W<matrix<double> > a,
                     shared_R<matrix<double> > b,
                     shared_R<matrix<double> > c) {
        // Sequential computation of "a = a - b * c" ...
}

ath ParBlockFact_LU(matrix<shared <matrix<double> > > A ) {
  for( int k = 0 ; k < A.dim ; k++ ) {
   fork<Factorization_LU> () ( A[k,k] ) ;
    for( int i = k+1 ; i < A.dim ; i++ )
      fork<M_TimesInverse_U> () ( A[i,k],  A[k,k] ) ;
    for( int j = k+1; j < A.dim ; j++ )
      fork<I nverse_L_Times_M> () ( A[k,j], A[k,k] ) ;
    for(i = k+1 ; i < A.dim ; i++ )
      for( j = k+1; j < A.dim ; j++ )
        fork<MinusTimesEqual> () ( A[i,j], A[i,k], A[k, j] ) ;
  }
}
```

Fig. 10. The program implements the classical block LU factorization. The type matrix is a container for two dimensional square matrices (**dim** gives the dimension of the matrix). This container is related to a format descriptor, allowing its use both to put objects in shared memory (**shared<matrix<**...**>>**) and to pass by value to a task a data structure containing several shared objects (**matrix<shared<**...**>>**).

TABLE VI

Times in seconds of LU block-factorization for various matrix sizes on a SMP architecture with block size = 120 and on a NOW architecture with block size = 200.

	SMP			NOW			
size	# Tasks	Sequential	4 procs	# Tasks	Sequential	1 node, 2 procs	2 nodes, 4 procs
1000	287	13.02	4.25	57	15.64	8.32	10.43
1400	652	35.67	11.09	142	38.95	26.39	25.29
1800	1242	75.71	23.26	287	89.60	48.75	64.80
2200	2472	138.58	42.65	508	158.83	90.72	206.16

Besides, another more general static schedule can be used that takes benefit of the knowledge of the macro-data flow graph, once the main task – `ParBlockFact_LU` in Fig. 10 – is completed. By schedule annotation, we have experimented on this application the DSC schedule (presented in section IV-E.2) on the IBM-SP architecture previously described (n-queens problem) The table VII allows comparison with the bi-dimensional cyclic schedule The overhead due to the computation of the schedule when DSC is used is only related to the size of the macro-data flow graph and independent of the number of processors. If this overhead is not considered, the DSC scheduling algorithm leads to similar performances than the one obtained with the bi-dimensional cyclic schedule (which is computed "at hand").

TABLE VII

Times in seconds on the IBM SP machine for LU block-factorization with matrix size 1280, block size = 128 for two scheduling polices.

		IBM SP						
scheduling	# Tasks	Sequential	1 node	2 nodes	4 nodes	8 nodes	12 nodes	16 nodes
Cyclic	387	134.04	140.36	79.25	46.49	30.48	23.6	23.46
DSC	387	134.04	142.02	83.27	52.62	35.62	31.70	28.51

VI. Conclusion

In order to achieve practical efficient execution on a parallel architecture, the knowledge of the data dependencies related to the application motivates the development of various on-line and off-line scheduling algorithms. By typing and restricting access in shared memory, we show that such a data dependency graph can be computed on-line

on a distributed architecture. We introduce an imperative language named Athapascan-1 implemented as a C++ library, that enables the on-line computation of the embedded macro-data flow. Athapascan-1 is inspired both from Jade [17] concerning the typing of memory access and from Cilk [5] concerning the expression of parallelism. Parallelism is explicit but synchronization implicit. The semantics of Athapascan-1 only defines the data flow; it is independent from the scheduling algorithm used to execute the parallel program. Moreover, the semantics enables the use of a sequential and non-preemptive schedule: the reference sequential order.

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. So, the theoretical performance of a code (parallel time, communication and arithmetic works, memory space) can be evaluated directly from a cost model without need of a machine model. We exhibit on-line schedules with provable performances related to those costs on a distributed architecture. This extends the work on Jade where no time bound have been given and the work on Cilk where only strict multithreaded computations are considered. Using the result from [4], the bound on the memory space is achieved by taking benefit of the reference sequential order which is computed in constant time.

Athapascan-1 has been implemented on various parallel architectures. We detail its compilation and implementation on a general parallel architecture (cluster of symmetric multi-processors). The possibility of unrolling on-line the macro-data flow allows the use of various scheduling policies, from greedy on-line algorithms [5] to static tasks clustering such as DSC [20]. We present various experiments to study the impact of the scheduling policy on two classes of programs: recursive computations and numerical computations with predicted data flow.

We find that the semantics of Athapascan-1 makes easier the development of parallel programs and the portability of the code by changing the scheduling policy. However, although bounded by a constant, the overhead related to the computation of the macro-data flow, is important especially on a distributed architecture; so it requires the programmer to take care of tasks granularity. On considered applications, this appears natural in the sense

that corresponding efficient sequential programs implement a different algorithm when the size becomes too small: unrolling function call for recursive programs and blocked algorithms for linear algebra computations.

An issue to decrease this overhead is to take benefit of the reference sequential order to avoid the building of the data flow with no loss of parallelism. Experiments performed to evaluate the overhead of this schedule with respect to a pure sequential code shows that it could be used to amortize the cost of task creation. Then a future work consists in implementing a work-stealing algorithm similar to the one developped in Cilk [1]. The difference here is that the schedule will be based on the reference sequential order to execute efficiently programs with neither resctrictions on the pattern of the synchronizations nor migration of running closures.

References

[1] N. S. Arora and R. D. Blumofe and C. G. Plaxon. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of X SPAA*, Puerto Vallarta, Mexico, June, 1998.

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

[3] 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.

[4] 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.

[5] R. D. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall and Y. Zhou. *Cilk: An efficient multithreaded runtime system.* In *Proc. of 5th ACM-SIGPLAN Symp. on Principles and Practice of Parallel Programming.* ACM, New-York. 1995.

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

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

[8] J. Dongarra and D. Walker. Software Libraries for Linear Algebra Computations on High Performance Computers, *SIAM Review*, 37(2), June 1995.

[9] I. Foster, C. Kesselman and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication, *IEEE Journal of Parallel and Distributed Computing*, 1997.

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

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

[12] 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.

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

[14] G. Karypis and V. Kumar. Multilevel k-way partitionning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96-129, 1998.

[15] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press. Nov. 1994.

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

[17] M. Rinard. *The design, implementation and evaluation of Jade.* ACM Transactions on Programming Languages and Systems, 20(3):483-545, May 1998.

[18] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachussets, 3rd. ed., 1997.

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

[20] 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.