Athapascan-1: Parallel Programming with Asynchronous Tasks

Gerson G. H. Cavalheiro[*], François Galilée and Jean-Louis Roch

LMC-IMAG-APACHE Project[†]
Grenoble, France
http://www-apache.imag.fr

Abstract

Athapascan-1 is a C++ library for multi-threaded parallel programming. It implements a *macro* data-flow language: both computation and data grains are explicit. Semantics of shared memory access is based on a lexicographic order. The performance of a code (parallel time, communication and arithmetic works, space) can be evaluated from a cost model without need of a machine model. Its important feature is that, forbidding synchronization, it allows efficient on-line analysis of the data flow describing the execution. This enable to separate the scheduling algorithm, that can take benefit of the knowledge of this graph, from the code.
On specific elementary machine models, simple scheduling algorithms can be implemented that provide efficient execution of some specific class of codes. Portability is then achieved by suiting the scheduling algorithm to both the program and the target machine.

1 Introduction

Recent work in the field of parallel programming has resulted in the definition of extension of sequential languages that can be theoretically efficiently scheduled on abstract machine models. Such languages allow to explicit parallelism independently of a specific architecture. The performance of a program is evaluated directly from it using a language based performance model [1] that specifies costs of the primitive instructions and rules for composing costs across program expressions. The adequation between those theoretical costs and the effective performances of execution on an architecture is then related to the scheduling algorithm used.

Most scheduling strategies are based on a greedy list scheduling [9]. Such a schedule is based on an allocation of ready tasks to idle processors. Using it, several results have been obtained that proves that well defined classes of parallel programs can be executed in asymptotic optimal time on theoretical machine models such as the PRAM or the local-PRAM, including scheduling overheads [2, 5, 3].

Basically, a list scheduling do not require much information about processes in the application, although some knowledge may appear useful 1. For instance, a total sequential ordering of tasks allow to bound the amount of space required for the computation while achieving linear speed-up for certain classes of programs: strict computations [5] nested computations [2] or planar graphs [3]. Furthermore, in practice, due to magnitude of the ratio between local and remote memory access costs, some significant improvement can be brought to a schedule by some knowledge about the data flow graph corresponding to the execution [10]. Exploiting this knowledge leads often to a schedule well suited to a specific application on a given architecture.

Several languages have been designed that allow the on-line building of the data-flow graph describing the execution. Most of them are built on top of a standard sequential language. Although in Jade [16] and BSP [11] instructions are grouped by block, most languages are based on a parallelism expressed via asynchronous function calls: Sisal [14], Nesl [1], Cilk [4]. Synchronization that will occur during a sequence of instructions is expressed at task creation or by specific statements that allow a task to

[*]CAPES–COFECUB Brazilian grant
[†]CNRS, INRIA, INPG and UJF

synchronize with the computation it has previously spawned. To support both execution on a distributed architecture and the use of a list scheduling, a migration mechanism is required by any of this language to move a task that was idle and becomes ready to an idle processor if any.

In this paper, we present a C++ library that implements a parallel language, named Athapascan-1 (Ath stands for *Asynchronous Tasks Handling*). Requiring no migration, it leads to space and time efficient execution Its semantics is based on a lexicographic order on the program and is independent from the scheduling strategy used. Choice of the scheduling is annotated in the code.

The paper is organized as follows. Section 2 introduces the language and its semantics. Section 3 explicit the cost model related to the program and proves the existence of optimal scheduling for some classes of graphs on a distributed architecture with a sparse network. Section 4 is devoted to implementation. In section 5, some experimental measures on a distributed and on a shared memory architecture are presented.

2 The Athapascan-1 Language

In this section the Athapascan-1 language is described, its semantics, syntax, and some informations about its implementation are given.

2.1 Overview

Athapascan-1 is a data-flow language designed for parallel computation. The explicit 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 semantic relies on shared data access, and ensures that the value returned by the read statements is the last written value[1] according to the lexicographic order (statements in the program are lexicographically ordered by ';'). The main advantage of this semantic is its direct readability on the program source (figure 1 shows an obvious example).

The respect of the access 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. The scheduling of the created tasks is enforced by customizable schedulers that are fully separated from the application.

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 easier the use of the C++ library 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( rp_wp )< int > a;
  fork update( a );
  fork print( a );
}
```

Figure 1: The semantic access is based on lexicographic order: the execution of the `test` task will print 5 on output: the execution of `print(a)` is delayed until `update(a)` resumes. The `fork` keyword asynchronously creates a task, the `shared(w)<int>` type designs an object located in the shared memory, object that will be written (due to `w` ; `r` stands for read).

[1] or a copy of

2.2 Syntax and "Athapascan-1 objects"

2.2.1 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( <parameters> )
{ <statements> }
```

The preprocessor translates this declaration into a C++ class function.

A *task* implements a sequential computation which 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( <parameters> );
```

This statement creates an object, called a *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, the list of effective parameters and an execution method (called by the scheduler to run the task on the arguments). A closure is said *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.

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. These last parameters have a state associated to them: a shared data version is said *ready* if no task is able to write the data (directly or not) else *waiting*.

2.2.2 Shared data and their versions

The shared memory, that allows the tasks to cooperate, is composed by shared data. An object `x` in shared memory is declared as follow:

```
shared< T > x;
```

where `T` is the type of the data that will be shared. This type `T`, that can be user defined (communications of such types are made with the help of a data format descriptor related to type `T`), defines the granularity of the data handled in the algorithm. A succession of *versions* is associated to each shared data: each shared data version represents the value of the shared data at a certain state of program execution.

The declaration of a shared data creates an object, called *evolution*, containing pointers toward two *transitions*, objects managing the allocation, state and access 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();
const T& read();
void write( const T& );
void cumul( const T& );
```

2.2.3 Example

The figure 2 shows the different main 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 any more on a data in the case of transitions.

The figure 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

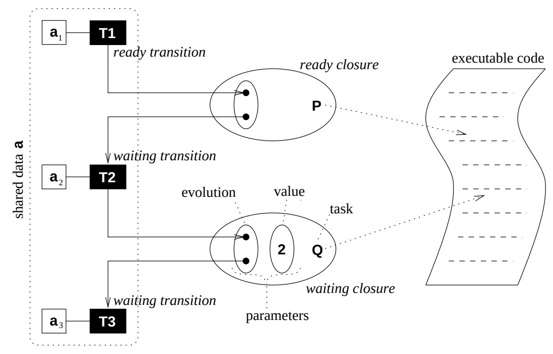


Figure 2: The Athapascan-1 internal data structures

`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.

```
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)

```
struct add {
  void operator()( 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)

Figure 3: Two versions of Athapascan-1 code computing the nth Fibonacci number. (a) presents a recursive way and (b) a cumulative one. The access mode of the shared are identified in the task declaration formal parameter: `r` stands for read, `w` for write and `cw<add>` for accumulation according to the `add` accumulation law.

2.3 Lexicographic ordering and parallelism detection

In order to respect the 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 the tasks precise 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).

That's for this reason that in Athapascan-1 tasks can not perform side effects (all manipulated shared are located in the parameter list) and that these shared parameters are typed according to the future manipulation.

2.3.1 Access right: evolution of shared data versions

In the task declaration of the formal parameters, the references to shared data version are typed by their *access right*, i.e. what 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.

`shared(r)<T>`, `shared(r_w)<T>` for example. Figure 4 shows which access rights conversions are possible at the tasks creation.

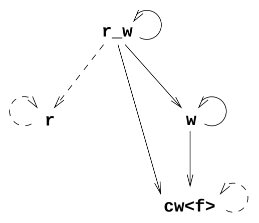


Figure 4: Allowed access rights conversions at task creation between the given effective parameter and the formal one. The solid lines translates the creation of a new version for the concerned shared data, contrary to dashed ones.

Remark concerning accumulation: The accumulation law is an user defined one (set to `+` operator by default) that is supposed to be associative and commutative. Note that this law is part of the reference type, so mix of different law is not allowed, but obey to lexicographic order semantic. Initial value for accumulation is the previous value of the shared according to the lexicographic access order.

2.3.2 Access mode: improving the parallelism extraction

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

With this refinement Athapascan-1 is able to decide more precisely if two tasks have a precedence relation or not: a task requiring a shared parameter with a direct read access `r` has a precedence relation with the last[2] task taking this same parameter with a writing right, whereas a task taking this same parameter in a postponed read access `rp` has no precedence because no access will be made (the precedence will be delayed to a sub-task created with an `r` type).

`shared(rp)<T>`, `shared(rp_wp)<T>` for example.

Note: The `shared` type used at shared data creation is nothing more than an alias to `shared( rp_wp )`.

2.4 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.

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 figure 5.

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.

2.5 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.

[2] according to the lexicographic order

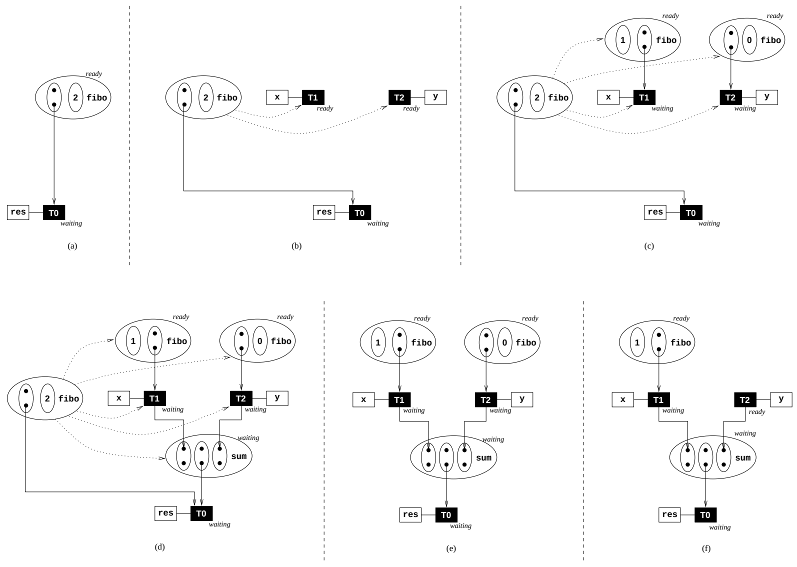


Figure 5: Dynamic evolution of the shared data versions access graph for the recursive version of `fibo(2)` execution. State (a) represents the initial graph, (b) the graph after the creation of the two shared `x` and `y`, (c) after the creation of the two `fibo` tasks (d) the graph after the creation of the `sum` task and just before the end of the `fibo(2)` task, (e) just after and (f) after the execution and completion of `fibo(0)`. The dotted lines represent the "mother-child" relation, for help in graph comprehension (it's not maintained in the system).

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). It 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 imposes 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.

  It imposes 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 example a task having a `shared(r_w)` can not create a task requiring a `shared(r)` as formal parameter,...

The strong typing of shared data version references allows the verification of the respect of these conditions $\mathcal{C}$ at compile time.

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

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

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

Proposition 2 *There's no any precedence relation between a task and the task that creates it.*

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

The task computation statements block is a word of $(S+F)^*$, where F denotes a block **fork** statement and S a classical block computation statement containing no task creation: let us say $S_1F_1S_2F_2\ldots S_{n-1}F_{n-1}S_n$ (S_1 and S_n can be empty). The entire independence between the created task and the creator task (proposition 2) involves that the trace $S_1F_1S_2F_2\ldots S_{n-1}F_{n-1}S_n$ is equivalent to $S_1S_2\ldots S_{n-1}S_nF_1F_2\ldots F_{n-1}$. This order of execution which corresponds to an inner most outer most order of evaluation is called in Athapascan-1 the reference sequential order of statements evaluation and is denoted by $\mathcal{R}$ in the following sections of this article (figure 6 illustrates this execution order).

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

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

Figure 6: In Athapascan-1 the two tasks writing (a) and (b) produce the same results. The task writing (b), where all tasks creations are performed at the end of execution, corresponds to the Athapascan-1 reference sequential order of statements execution.

Proposition 3 *The Athapascan-1 reference order of instruction evaluation respects the lexicographic order based access semantic and requires no copy.*

This results from the conditions $\mathcal{C}_2$ and $\mathcal{C}_3$. □

2.6 Summary

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.

The allowed conversions of shared data version types at task creation are summarized in the following table, figure 7.

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 7: Allowed conversion for passing reference on a shared data version as parameter of a task (`[p]` stands `p` for *postponed* access mode or nothing for direct access mode).

To conclude this presentation, a more complex example of an Athapascan-1 program that implements a two-dimensionnal block LU Factorization is presented below. Only task `ParBlockFact_LU` contains parallelism. Once this task has been completed, the synchronization that will occur till the end of the lgorithm between all remaining tasks ared konown. The figure 8 shows the graph generated for this program when applied to a 8×8 matrix (elements being blocks of arbitrary size).

```
TASK Factorisation_LU( shared(r_w)<matrix<double> > a ) { ... }
TASK M_TimesInverse_U( shared(r_w)<matrix<double> > m, shared(r)<matrix<double> > u ) { ... }
TASK Inverse_L_Times_M( shared(r_w)<matrix<double> > m, shared(r)<matrix<double> > l ) { ... }
TASK MinusTimesEqual( shared(r_w)<matrix<double> > a, shared(r)<matrix<double> >   b, shared(r)<matrix<double> > c) { ... }

TASK ParBlockFact_LU(array<shared(rp_wp)<matrix<double> > > A, int col_dim, int row_dim) {
 for( int k = 0 ; k < col_dim ; k++ ) {
   FORK Factorisation_LU, A[k*(row_dim+1)]);
   for( int i = k+1 ; i < col_dim ; i++ )
     FORK M_TimesInverse_U( A[i+k*row_dim],  A[k*(row_dim+1)] );
   for( int j = k+1; j < row_dim ; j++ )
     FORK Inverse_L_Times_M( A[k+j*row_dim], A[k*(row_dim+1)]);
   for(i = k+1 ; i < col_dim ; i++ )
     for( j = k+1; j < row_dim ; j++ )
       FORK MinusTimesEqual( A[i+j*row_dim], A[i+k*row_dim], A[k+j*row_dim]);
 }
}
```

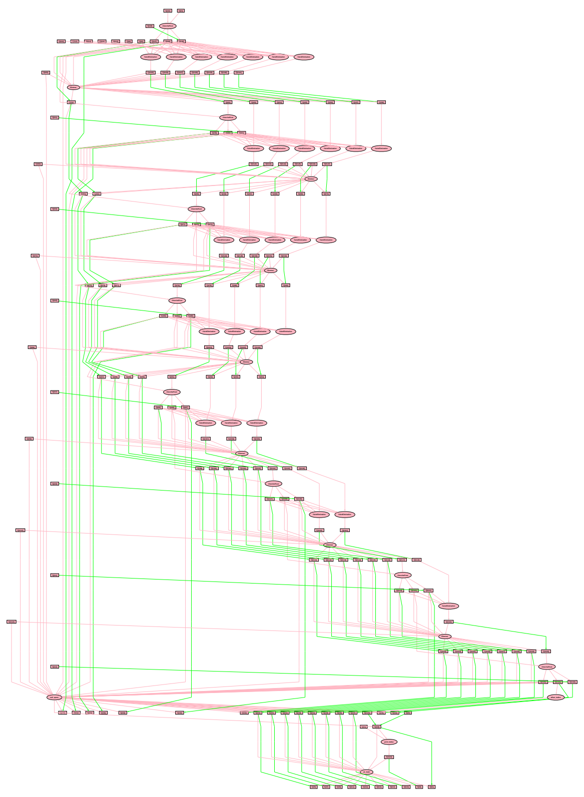


Figure 8: Gauss graph for an 8×8 dense matrix.

3 Cost model

In this section, theoretical bounds on the time and space required to execute an Athapascan-1 program on a distributed architecture with p identical processors are given. Bounds are related to abstract cost measures on the program itself.

3.1 Weighted macro data-flow graph

An instant of the execution of an Athapascan-1 program (with a specific input) can be represented by the related shared data access graph (9) which is direct and acyclic. Superposing all the graphs in the

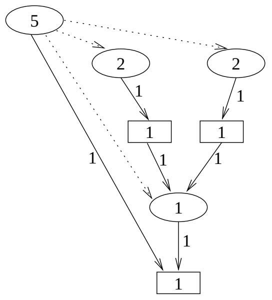


Figure 9: The shared data access graph can be annotated by arithmetical cost for nodes and communication costs for the edges. Here, the arithmetical cost is take as the number of task's statements and communication costs correspond to the communication of an integer and account for 1.

evolution (see figure 9) leads to a direct acyclic graph G. which is an abstract representation of the whole execution that describes both shared data and tasks (control) flow. G is bipartite with node sets $J = \{j_1, \dots, j_n\}$ corresponding to closures (j meaning *job*) and $T = \{t_1, \dots, t_m\}$ corresponding to shared data versions (t meaning transition). An edge goes from t_k (resp. j_i) to j_i (resp. t_k) if t_k is accessed in a read (resp. writing) right by t_k.

Since G is at coarse-grain, it is weighted. Each node (either closure node or transition node) is weighted by the number of elementary arithmetic operations performed (either to execute the related task or to elaborate the version in case of cumulative access for instance). Each edge is weighted according to the size of the shared data related to it: any shared data passed as effective parameter accounts for 1 unit; moreover, when the edge corresponds to an access within a direct mode, it accounts for the size of the data pointed to by the shared.

3.2 Cost measures of an Athapascan-1 program

Due to the lexicographical semantics, costs can be evaluated directly on the program itself from related recurrence equations (max, +) on the size of the input [12, 1]. Here, following notations in [1, 5], we define abstract costs from G in order to relate them to a specific input.

Weights in G reflects directly an execution that respects the reference order $\mathcal{R}$ of execution (inner most, outer most). Execution of a closure is then delayed until the task that has created it resumes.

Let T_1 and S_1 denote the serial time and space required by a serial execution of the program. It can be seen that order $\mathcal{R}$ does not intrinsically increase by more than a factor 2 both the time $\tilde{T_1}$ and the space $\tilde{S_1}$ required by a classical serial execution. In effect, since from proposition 1 **fork** statements are of constant cost, $\tilde{T_1} = T_1$. Besides, due to the independence property 2, a sufficient condition to have $\tilde{S_1} = S_1$ is that any task creates at most two closures just before resuming. Another semantically equivalent program – i.e. computing the same values – satisfying this condition can be obtained by compiling continuation of tasks; it is sufficient to replace the block of instructions following a fork statement by the forking of a task which body is this block. This transformation increases nor the time neither the space by more than a factor 2. Any Athapascan-1 program with serial time $\tilde{T_1}$ and space $\tilde{S_1}$ can thus be directly translated in another one for which $T_1 = O(\tilde{T_1})$ and $S_1 = O(\tilde{S_1})$. It can be noted that parallel time on an unbounded number of processors remains unchanged though communication work and delay may be increased during this transformation by more than a constant factor.

Arithmetic depth T_∞ (or *parallel time*) and work T_1 (or *sequential time*) are evaluated from G taking into account weights of nodes. T_∞ is a lower bound of the minimal time required by any schedule on an unbounded number of processors ignoring communications times (PRAM model [8]). T_1 is the number of operations required by a sequential execution of the algorithm. Since the best schedule may replicate some arithmetic nodes in order to minimize completion time, note that T_1 is also a lower bound on the number of operations performed by any schedule.

Communication delay C_d and work C_w are evaluated similarly from G taking into account only weights of edges. C_d is the length of the critical path in G; C_w is the sum the weights over all edges. Note that C_w is then an upper bound on the total number of remote access (out of the local memory or out of the cache) performed during a serial execution.

Since the overhead involved by the scheduling of the graph is also concerned, we denote respectively by $\#N$ and $\#E$ the total number of nodes and edges in G.

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 from the machine.

3.3 Scheduling on a PRAM

The PRAM model [8, 12] allows to get rid off communication overheads. In order to be consistent with Athapascan-1, we consider a CRCW-PRAM with cumulative concurrent write ones. We focus here on the time required to schedule tasks on such a machine.

With no restriction, we assume that during execution, any task has performed a bounded number (let say 2) of `fork` statements. We firstly consider an unbounded PRAM (∞-PRAM), consisting in an unbounded processors. Each processor is indexed by an integer.

Proposition 4 *Including the cost of the schedule computation, any Athapascan program can be executed on the ∞-PRAM in time $O(T_\infty)$.*

Taking benefit of the lexicographical semantics of Athapascan-1, any closure t can be uniquely assigned to a sequence $s(t)$ in $\{0+1\}^\star$. The first closure (implicitly defined in C++ by `main`) is indexed by '1'. If $s(t)$ is the sequence associated to a closure t that sequentially creates closures t_0, t_1, then $s(t_i) = s(t).i$. For any t, $s(t)$ is computed in constant time and t is then mapped on processor with index $s(t)$. It will start execution when it becomes ready. The detection, that corresponds to an unbounded fan-in boolean operation, is performed in constant time. This leads to an execution time $O(T_\infty)$. $\square$

We consider now a PRAM with a bounded number of processors p. We give two bounds; the first consider

Proposition 5 *Including the cost of the schedule computation, any Athapascan program can be executed on the p-PRAM in time (a) $T_\infty + \frac{T_1}{p} + O(\#N + \#E)$ or (b) $T_\infty + \frac{T_1}{p(1-1/\log p))} + O\left(\log(\#N + \#E) + \frac{\#N+\#E}{p}\right)$. Both schedules are obtained by deterministic non-preemptive algorithms.*

Both scheduling algorithms are based on a greedy list strategy [9]: 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 $O(\#N + \#E)$; thus both idle time corresponding to busy wait of the lock and management of evolutions of the graph are bounded by $O(\#N + \#E)$.
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 $\log(\#N + \#E)$. Other $p(1 - 1/\log p)$ processors are dedicated to execution of closures. $\square$

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

3.4 Scheduling on a distributed architecture

We consider here scheduling of an Athapascan program on a distributed architecture with p identical processors. The shared memory on this architecture is emulated with the help of universal hashing functions [13]. The delay occurring for any access is bounded by $h(p)$. In order to obtain efficient emulations ($h(p)$ constant or very small to p), a slackness strategy [15, 17] is used. It consists in emulating a $q(p)$-PRAM on the distributed architecture, $q(p)$ being larger enough compared to p. In the following, the distributed machine is assumed to emulate a $q(p)$-PRAM with delay $h(p)$.

Proposition 6 *Including the cost of the schedule computation, any Athapascan program can be executed on a distributed architecture with p identical processors in time* $q(p)T_\infty + \frac{T_1}{p} + h(p)\left[q(p)C_d + \frac{C_w}{p} + O(\#N + \#E)\right]$. *This schedule is obtained by a non-deterministic preemptive algorithm but with no migration of running closures.*

The proof is deduced from proposition 5 applied on a PRAM with q processors. The whole number of remote access with delay h is bounded by C_w. This leads to a schedule on the q virtual processors with length bounded by $T_\infty + \frac{T_1}{q} + h\frac{C_w}{q}$. 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 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 $C_w = o(T_1)$ and $(\#N + \#E) = o(T_\infty)$ 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 obtained in [5] 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 $W_c = \Omega(n)$ access. The schedule time is then $O(h(p)T_1/p)$ and efficiency depends on h.

In this case, other scheduling algorithms can be used. For instance, if we consider the previous 2-dimensional block Gaussian elimination 8, after execution of closure `gauss` the whole graph G is computed and remains unchanged till end of computation. Then, if cost informations can be given in the program in order to annotate the graph with computation and communication costs (which is possible in case of a dense matrix), a specific block-cyclic schedule can be computed which results in efficient executions.

3.5 Scheduling to minimize space

Previous schedules do not guarantee any bound concerning the space required. However, they all realize strict multi-threaded computations [5]: a closure does not start its execution until all its arguments are available.

Since they are based on a list of ready closures, sorting this list according to an order that minimizes sequential space allow to bound the memory space required [5, 2]. Due to the lexicographic semantics, this list can be directly sorted with no overhead according to the reference order.

Space required for the resulting schedule is then bounded by $q(P)S_1$ on the distributed machine [5].

Besides, after execution of some closures in the program, the graph G may be completely built and annotated (see Gaussian elimination, fig. 8). Then a specific scheduling algorithm can be used that delivers better bounds on the space required.

For instance, if the graph is known to be planar [3], it is possible to sort ready closures in the list in order to achieve a space $S_1 + q(p)\log q(p)T_\infty$.

3.6 Conclusion

Abstract cost measures are defined on an Athapascan-1 program that allow to analyze its theoretical performances. This allow the implementation of various scheduling algorithms with provably performances for specific class of programs.

For a given program, execution costs on a given architecture depend heavily on the scheduling algorithm used, which appear very difficult to choose.

However, since semantics of the program do not depend on the schedule used, it can be implemented apart from the program itself. In following sections, the effective implementation of the language is presented and a framework that allow the use and the implementation of various scheduling algorithms is detailed.

4 Athapascan runtime and distributed implementation

In this section we describe the distributed implementation of Athapascan-1. This implementation relies on a multi-threaded, portable, parallel programming runtime system, Athapascan-0, that overlaps communication delays with computation.

4.1 Shared data versions access graph management

The maintained graph is distributed: closures and edges are unique in the system, but transitions may be replicated (as shown in figure 10). These transitions are replicated so that a closure always locally accesses its connected transitions (via the pointers of its possessed evolutions). So 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 then proportional to the number of its parameters.

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

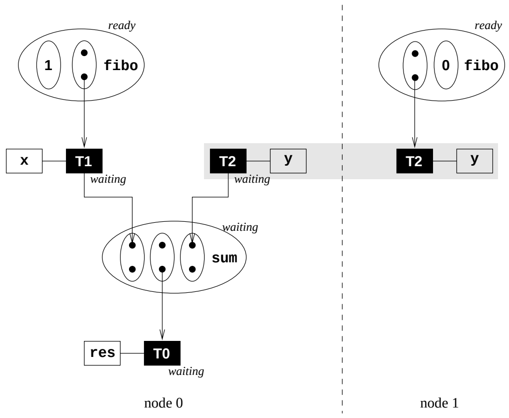


Figure 10: The graph can be distributed. In that case, only transitions are replicated and have a distributed coherence maintained. When the task `fibo(0)` will be completed, a message will be sent from node 1 to node 0 to warn the transition `T2` that no more writers exist on this node.

Communications are handled by a runtime system called Athapascan-0. It allows the remote creation of threads furnishing communications and synchronization facilities. 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 provide the global scheduling of closures (tasks) on the whole architecture.

4.2 Athapascan-0: an efficient and portable integration of communications and multi-threading

Athapascan-0 [6] is a multi-threaded parallel programming runtime system. It is designed to support portable development and efficient execution of irregular, fine-grained parallel programs, usually with a number of processes much larger than the number of available processors. Processes are supported by an inexpensive mechanism, the threads.

Athapascan-0 is designed for a general parallel machine that consists in 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 communications on disjoint routes.

The Athapascan-0 runtime is built on top of standard unmodified message passing and thread libraries and much care has been taken on communication efficiency and communication and computation overlapping.

An Athapascan-0 parallel machine is composed by 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.

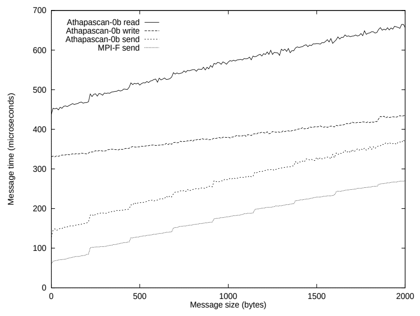
Athapascan-0 offers a library with the basic communication primitives extended to manipulate formatted data. A format holds a data description, which can have a regular organization in memory such as MPI data-types or a complex packing procedure, as in XDR. This description allows the user to communicate variable sized structures without the need to manage data packing and buffers.

4.3 Athapascan-0 performances

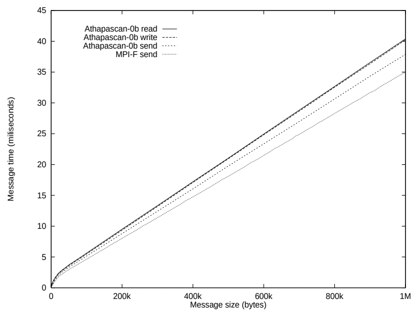
In this section, the performance of Athapascan-0 is compared to the thread and communication libraries on top of which it is implemented.

The necessary time for a remote write (figure 11 (a)) accounts only for the local delay and not for the time the write would take to complete on the remote node. This explains the different slope of the write curve compared to the message send curve. The difference is also explained by the fact that a write is implemented by two messages (one for the data address and format, the other for the data). The read curve has the same slope as the send curve because a read involves a round trip of messages, the first of which is of fixed size, regardless of the size of the read.

Large messages performance is shown figure 11 (b). Athapascan-0 overhead is small in percentage for larger messages. Maximum throughput for MPI-F is 28.5 MBytes/sec and for Athapascan-0 27 MBytes/sec. What may seem to be a per-byte overhead is in fact due to the polling rate of the MPI library in Athapascan-0.



(a)



(b)

Figure 11: Communication performance of Athapascan-0 compared to the thread and communication libraries on top of which it is implemented. (a) concerns short messages and (b) large ones. The machine is an IBM SP/1 with a High Performance Switch network (HPS) running AIX 3.2.5. The libraries used are IBM DCE threads (POSIX draft 1003.4a) and MPI-F, a thread aware research prototype designed by IBM to run on the HPS.

5 Scheduling implementation

Athapascan-0 handles local scheduling of threads on a node. In this section, we present the way Athapascan-1 closures are globally scheduled. A scheduling algorithm is then used to determine a site and a date to trigger the execution of a closure (an Athapascan-1 *task*). The algorithm implemented by this level of scheduling does a distribution of the work generated by an application attempting to optimize a global index of performance, as memory use or, more typically, the execution time. The scheduling which allows the different threads to access a (set of) real processor(s) depends on the scheme adopted by the lowers layers.

For Athapascan-1, we propose a framework to supports this scheduling layer. With this framework we intent to separate the scheduling from the application and from the execution support. Some static and dynamic scheduling algorithms are delivered with the framework, but new ones can be easily added following the rules defined for the framework interface.

5.1 Annotating a program by scheduling

The scheduler framework offers different scheduling algorithms, implemented as C++ classes. At execution time, Athapascan-1 allows the user to specify the one used by code annotation, a work-stealing algorithm being the default one. The scheduler is selected by prefixing the `fork` operator:

```
Schedule( sched ) fork user_task( <parameters> );
```

The task `user_task` will be scheduled by the scheduler `sched` and `sched` will be the algorithm default to schedule closures created by `user_task`. Some attributes can be added to the task, such as priority, computational cost or locality; they can be exploited by `sched`. Those attributes are specified in the following way:

```
Schedule( sched ) Information(p, c, l, e) fork user_task( <parameters> );
```

where `p`, `c` and `l` correspond respectively to priority, computational cost and locality. The information given by `e` corresponds to an extra attribute which semantic depends on `text`.

5.2 Relationship between the graph and the schedule

Informations about a task are stored as attributes of the associated closure. After creation (`fork`) and until it is completed, the state of the closure depends on the graph of evolutions. The main states are:

- *waiting*: the closure has at least one predecessor not yet completed;
- *ready*: the closure can be executed;
- *running*: the closure runs 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 required about the current global state of execution: for instance, closure attributes, parameters, state or precedence constraints.

Its is relatively easy to identify when a scheduling action may be sparked; harder is to identify when it *must* be do. As seen in 3, this decision depends from the application characteristics and the machine resources. Often is necessary a flexible scheduler environment to support both machine and application characteristics.

5.3 Framework for scheduler development

We structure our flexible scheduler support as a framework [7] composed of four modules and a set of *callback* functions (figure 12). In this scheme the Job Builder is the graph generator (an Athapascan-1 program in the context of this paper) and the Executor is a module implemented over the Athapascan-0 to handle a pool of threads to execute the closure. The number of threads of this pool may be suited for an application in order to overlap remote access delay.

The Policy Manager is the kernel of the scheduler framework; it defines interfaces with the Job Builder and the closure. By the interface with Job Builder, the Policy Manager receives, by a service call, information about any graph modification.. Each scheduling algorithm may have a specific implementation of these services (default implementations are provided). The interface defined for the closure provides services used by the Policy Manager to read its attributes, parameters and successors and predecessors in the graph and to move it from a processor node to another.

Associated to the Policy Manager, a Load Estimator books the load of processor nodes.

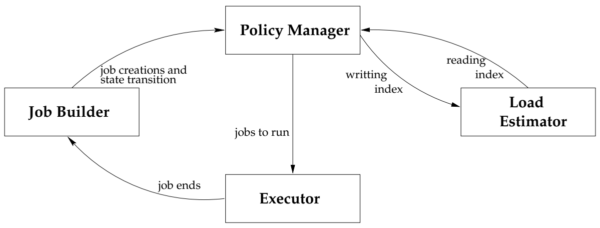


Figure 12: Framework for a general scheduler support

5.4 Some schedulers

With the framework described above we intent to have a software layer providing facilities to implement a large variety of scheduling algorithms independently from the application program and from the machine resources. Then, the scheduling algorithm may be tuned to a tuple $\{application, machine\}$.

5.4.1 Work-stealing

There are a lot of works in the literature about distributed greedy scheduling algorithms, most often being based on a work-stealing principle [4].

In the work-stealing scheduling, each processor node stores in a queue Q closures ready to be executed. The size q_i of Q_i represents the reserve of work in the node i. The q_i value is always compared with the system constants q^{max} and q^{min} (where $0 \leq q^{min} \leq q^{max}$). When $q_i \leq q^{min}$ the node i is considered under-loaded: it chooses another node l to send a message to *thief* work. If $q_l \geq q^{max}$ the node l is overloaded then it can answer with a work; otherwise, the message is forwarded to another node l'.

Concerning implementation, different solutions are proposed whose efficiency may depend on a tuple $\{application, machine\}$. For instance, it is possible to discuss the influence of the threshold q^{max} and q^{min}, the algorithm to choose a node to send a thief message and the algorithm to choose the best closure to answer a thief message.

For a selection of a node to thief, in general, a randomly choice may be a good solution when there are no global load information. However unsuccessful thief messages may be sent. A central node may be used providing a global reserve list, avoiding unsuccessful thief messages but introducing an overhead to actualize this reserve.

Also various policies can be used to select a closure to answer a thief message or to be executed locally. The queue list may be eventually ordered. As seen before, the order may respect a sequential ordering of the execution in order to optimize space; or, closures may be sorted according to other criteria related to their attributes, such as computational cost.

5.4.2 Static algorithms

At a given instant of the execution, the evolution graph may represent all the closures and their dependancies to be executed until a global synchronization. For instance, consider the program in section 2.6; once the closure `ParBlocFact_LU` has been completed, the whole dependancy graph is known since other tasks implements sequential computations on a block.

If some informations are known about the graph such as computation costs for instance, a static scheduling such as DSC implemented in Pyrros [18] can be used. If not, it is well known that on a distributed uniform architecture, a two-dimensionnal block-cyclic allocation of tasks to processors leads to good performances.

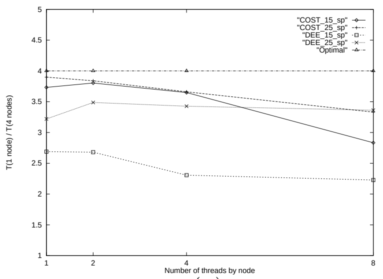
Due to the relationship between the application program and the framework we proposed, such algorithms may be implemented.

6 Experimental behavior

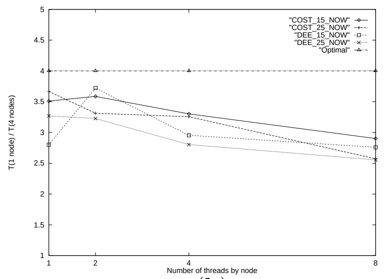
In this section we present some performance results obtained by the execution of a recursive Athapascan-1 program (with the same schemes than figure 3(a)) using two different schedulers. We used two distributed architectures and a shared memory multiprocessor in our experiments.

6.1 Experiment on distributed architectures

Figure 13 shows some performance curves obtained by the execution of a recursive algorithm: in this algorithm each task generates new tasks until a threshold, defined by the program; then, the task starts a sequential algorithm. Each curve represents the speed-up, ratio between execution times obtained on a machine with 4 nodes versus an one node machine. The execution times for each case was measured 50 times and the average value calculated after elimination of abnormal results are represented. The axis X shows the impact of different numbers of threads employed to execute the user services; note that there are some Athapascan-0 and Athapascan-1 threads running to support the execution. The graph (a) is the execution in a IBM-SP2 (using 4 nodes, AIX 4.2.1, MPI-IBM) parallel machine and the graph (b) is the same experiment under a homogeneous network of workstations, NOW (4 monoprocessors, Solaris 2.5.1, LAM, Mirynet network).



(a)



(b)

Figure 13: Work-stealing under (a) a parallel machine (b) a homogeneous NOW

To achieve this experiments, with total cost equivalent to 40, two thresholds were used: 15 and 25. The first one to generate a large number of tasks with a small grain and the second one to produce less tasks with a large grain. The application was scheduled with two versions for the work-stealing, named in figure 13 by DEE and COST; they differ by the algorithm used to select the closure sent to a thief processor. The DEE algorithm selects the last closure that became ready; the COST algorithm selects the closure that will generate the larger number of tasks (the first in a depth first execution). For both algorithms, the node to be thief is chosen randomly and values for q^{min} and q^{max} were respectively 1 and 2.

Looking both figures 13(a) and 13(b), we can observe that performances decreases when the number of threads on one node increases; note that in both NOW and SP2 architecures, nodes are mono-processor. Also we can verify the parallel machine is more sensitive to the better distribution of the work provided by the COST algorithm which selects the larger closure to migrate. An analysis equivalent is not possible for the NOW where we observe a less constant behavior due at least to three factors:

- the NOW architecture proposed is not adequate for the application;
- MPI-LAM introduces an extra overhead for communications;
- the threads were created as *user threads*; at execution Solaris 2.5.1 adjust automatically the number of virtual processors, changing the way how the threads are scheduled. Under AIX 4.2.1 all threads are created as *system threads*.

6.2 Experiment on SMP

The same application was ran in a SMP architecture (Intel Quadriprocessor, Solaris 2.5.1, LAM). The table 6.2 shows the execution times measured for this SMP architecture and for two monoprocessors: a node of each distributed architecture discussed by the former paragraph.

Also the execution times for the sequential executions are presented. We distinguish the sequential execution of a C++ equivalent program ("Sequential" in the table) from the one delivered by a depth-first sequential scheduling of the Athapascan-1 program.

	Threshold=15			Threshold=25		
	Monoproc IBM-SP2	Monoproc Intel	Quadriproc Intel	Monoproc IBM-SP2	Monoproc Intel	Quadriproc Intel
Sequential	43.30	39.90	22.60	43.30	39.90	22.60
Ath-sequential	43.81	40.09	23.37	43.84	39.94	23.29
1 Thread	320.02	283.75	99.51	106.45	101.79	24.45
2 Threads	272.68	201.21	106.54	80.63	71.25	16.29
4 Threads	238.27	172.14	149.85	67.16	59.93	13.94
8 Threads	194.89	154.04	185.91	61.66	54.16	12.98

Table 1: Execution under a node of different architectures

From these results, it appears that the performances obtained on the SMP architecture are poor with fine grain tasks. Increasing the number of threads introduces more overhead for the system to manage the closures produced. However, a larger computational grain (threshold=25) leads to a speed-up when the number of threads grows, this overhead being overlapped.

6.3 Overheads in Athapascan-1

Figure 6.2 allows to have an idea of the overhead of the distributed implementation of Athapascan-1. A program Athapascan-1 may be compiled to generate a sequential version. Comparing the time obtained with this version to the pure sequential program with the times for Athapascan-1 in sequential we verify a very small overhead.

The overheads generated by Athapascan-1 (graph manipulation and scheduling framework) and by Athapascan-0 (run time support) can be observed by comparing sequential execution (with the equivalent C++ sequential code) with a parallel execution of Athapascan-1 using only one execution thread. These overheads are attributed mainly to the implementation of Athapascan-1(graph management and scheduling framework) and to Athapascan-0 runtime daemons. If the tasks are large enough, these overheads however are overlapped.

7 Conclusion

We presented a language that enables the on-line building of the DAG that describes execution of a program with bounded overhead. Semantics is based on the lexicographic order between instructions; this allow efficient scheduling (space bounded computations) and to separate the program from the scheduling algorithms used, that can be tuned by annotation for execution on a specific architecture.

Both grain of data and computation are explicit but independent from the target architecture (fine-grain model). Parallelism is expressed via asynchronous creation of tasks. Any task can be scheduled such that once started it can keep on its execution with no preemption.

A cost model allows to define parallel depth, work, sequential space, communication work and delay. Efficient scheduling are given that achieve bot optimal time and bounded space on an distributed architecture with sparse interconnection network for a large class of programs. However, in practice, the use of this algorithm on even simple distributed architecture is far from performance obtained by simple static strategies.

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 a strategy. In this context, scheduling algorithms that uses task replication in order to achieve optimal execution time with communication are attractive.

Another issue concerns the choice of the total order on tasks that allows to bound space. Since parallel time is related to sequential space, the knowledge of memory requirements of the tasks in the graph brings information to improve the space required by a sequential scheduling. A problem here is also to bound the number of task replications.

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 *Proceedings of the 7th Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa-Barbara, California, 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 *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures.* ACM Press, 1997.

[4] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* Ph.D. Thesis. MIT. Massachusetts. 1995.

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

[6] J. Briat, I. Ginzburg, M. Pasin and B. Plateau. Athapascan Runtime: Efficiency for Irregular Problems. In: *Proc. of EuroPar'97.* Passau. Aug. 1997.

[7] Gerson G. H. Cavalheiro, Yves Denneulin and Jean-Louis Roch. A general modular specification for distributed schedulers. In: *Proc. of EuroPar'98.* Southampton. Sept. 1998.

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

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

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

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

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

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

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

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

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

[18] 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*, Washington, July 1992, pp 428-437.