INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

ATHAPASCAN:
an API for Asynchronous Parallel Programming

User's Guide

Jean-Louis Roch — Thierry Gautier — Rémi Revire

N° 0276

Février 2003

THÈME 1

Rapport technique

9-0803

ATHAPASCAN:
an API for Asynchronous Parallel Programming

User's Guide

Jean-Louis Roch[*], Thierry Gautier[†] , Rémi Revire[‡]

Thème 1 — Réseaux et systèmes
Projet APACHE

Rapport technique n° 0276 — Février 2003 — ?? pages

Abstract: ATHAPASCAN was an macro data-flow application programming interface (API) for asynchronous parallel programming. The API permits to define the concurrency between computational tasks which make synchronization from their accesses to objects into a global distributed memory. The parallelism is explicit and functional, the detection of the synchronizations is implicit. The semantic is sequential and a ATHAPASCAN's program is independent from the target parallel architecture (cluster or grid). The execution of program relies on an interpretation step that builds a macro data-flow graph. The graph is direct and acyclic (DAG) and it encodes the computation and the data dependencies (read and write). It is used by the runtime support and the scheduling algorithm to compute a schedule of tasks and a mapping of data onto the architecture. The implantation is based on using light weight process (thread) and one-side communication (active message). This report presents the C++ library of the API of ATHAPASCAN.

Key-words: parallel programming, macro data flow, scheduling, cluster and grid computing

[*] MdC INPG, Leader of the ATHAPASCAN's team, jean-louis.roch@imag.fr
[†] CR INRIA, thierry.gautier@inrialpes.fr
[‡] Doctorant MENSR, remi.revire@imag.fr

Athapascan :
une Interface pour la Programmation Parallèle Asynchrone

Manuel de l'utilisateur

Résumé : Athapascan est une interface de type macro data-flow pour la programmation parallèle asynchrone. Cette interface permet la description du parallélisme entre tâches de calcul qui se synchronisent sur des accès à des objets à travers une mémoire globale distribuée. Le parallélisme est explicite de type fonctionel, la détection des synchronisations est implicite. La sémantique est de type séquentielle et un programme écrit en Athapascan est indépendant de l'architecture parallèle (grappe ou grille). L'exécution est basée sur une interprétation du programme qui permet la construction d'un graphe macro data-flow. Ce graphe, orienté et sans cycle, décrit les calculs et les dépendances de données (lecture et écriture); il est utilisé par le support d'exécution pour contrôler l'ordonnancement des tâches de calcul et le placement des données sur les ressources de l'architecture. L'implantation repose l'utilisation de threads et de communications undirectionnelles (messages actifs). Ce rapport présente l'utilisation de l'API d'Athapascan en tant que bibliothèque C++.

Mots-clés : programmation parallèle, macro data flow, ordonnancement, grappe et grille de calcul

1 Information and contacts

The Athapascan project is still under development. We do our best to produce as good documentation and software as possible. Please inform us of any bug, malfunction, question, or comment that may arrise.

More information about Athapascan and the APACHE project can be found online at:

`http://www-apache.imag.fr.`

The user can subscribes to the following mailing-lists:

- `http://listes.imag.fr/wws/info/id_a1_hotline`: to have help from the Athapascan's group about installation or programming pitfalls or bug report.
- `http://listes.imag.fr/wws/info/id_a1_devel`: to reach the developers of Athapascan's group about implementation details (questions, remarks, design, ...).

The authors thank all the people who has worked on this project:

PhD Students :

- François Galilée
- Mathias Doreille
- Gerson Cavalheiro
- Nicolas Maillard

Engineers, students :

- Arnaud Defrenne
- Jo Hecker

Contents

2 Introduction

Athapascan-1 is the C++ application programming interface of Athapascan. It is a library designed for the programming of parallel applications.

2.1 About Athapascan

Athapascan is build on a multi-layered structure:

1. Athapascan-0 is the communication layer based upon MPI and POSIX threads; the extension independent from the transport library is called Inuktitut;
2. Athapascan-1 is the user-end API;
3. Athapascan also contains visualization tools for debugging purposes.

Athapascan is a **“high level”** interface in the sense that no reference is made to the execution support. The synchronization, communication, and scheduling of operations are fully controlled by the software.
Athapascan is an **explicit parallelism language**: the programmer indicates the parallelism of the algorithm through Athapascan ’s two, easy-to-learn template functions, `Fork` and `Shared`. The programming semantics are similar to those of a sequential execution in that each “read” executed in parallel returns the value it would have returned had the “read” been executed sequentially.

Athapascan is implemented by an easy-to-use C++ interface. Therefore any code written in either the C or C++ languages can be directly recycled in Athapascan.

The Athapascan interface provides **a data-flow language**. The program execution is data driven and determined by the availability of the shared data, according to the access made. In other words, a task requesting a read access on shared data will wait until the previous task processing a write operation to this data has ended.

Athapascan is **portable and efficient**. The portability is inherited from the Athapascan-0 communication layer of the environment, which may be installed on all platforms where a POSIX threads kernel and a MPI communication library have been configured.
The efficiency with which Athapascan runs has been both tested and theoretically proven. The Athapascan programming interface is related to a cost model that enables an easy evaluation of the cost of a program in terms of work (number of operations performed), depth (minimal parallel time) and communication volume (maximum number of accesses to remote data). The execution time on a machine can be related to these costs [?].

Athapascan has been developed in such a way that one does not have to worry about specific machine architecture or optimization of load balancing between processors. Therefore, it is much simpler (and faster) to use Athapascan to write parallel applications than it would be to use a more “low level” library such as MPI.

2.2 Reading this Document

This document is a tutorial designed to teach one how to use Athapascan’s API. Its main goal is not to explain the way Athapascan is built.

If new to Athapascan, it is recommened to read all of the remaining text. However, if the goal is to immediately begin writing programs with Athapascan , feel free to skip the next two chapters. They simply provide an overview of:

- how to install Athapascan’s librairies, include files, and scripts (Chapter **??**),
- how to test the installation performed (Chapter **??**),
- the API (Chapter **??**).

The other sections will delve deeper into Athapascan’s API, so that the user can benefit from all of its functionalities. They explain:

- the concepts of “tasks” and “shared memory” (Chapters **??** and **??**, respectively);

- how to write the code of desired tasks (Chapter **??**);
- how to make shared types communicable to other processors (Chapter **??**);
- which type of access right to shared memory should be used (Section **??**);
- how to design parallel programs through complex examples (Chapter **??**);
- how to select the proper scheduling algorithm for specific programs (Appendix **??**);
- how to debug programs using special debugging and visualizing tools (Appendix **??**).

3 Installation of Athapascan-1 version 2.0 Beta

Athapascan is easy to install. This chapter only covers the installation of Athapascan-1 version 2.0 Beta on a UNIX system.

The lastest releases of Athapascan software are available for download on APACHE's web-site: `http://www-apache.imag.fr/software/ath1/archives/`.

3.1 Installation of Inuktitut and Athapascan

The entire installation is based upon the couple, `configure/makefile`. There is a **makefile** in the top level of the both the Inuktitut (the excecution support) and the Athapascan-JIT-1.3 folders that provide sound settings for the installation. Modify the settings at the beginning of the file in order to finalize the installation to the desired folder.

Next execute:

```
make config
```

In order to define certain variables at the time of compilation, use the command `make "CXXGLAGS=...."` in each of the folders, Inuktitut and Athapascan-JIT-1.3.

4 Getting Started (API)

This chapter presents an overview of Athapascan 's API and demonstrates how to build Athapascan programs through simple examples.
NB: The source codes of the examples presented in this tutorial are available online on our web-site.

4.1 Overview of Athapascan

4.1.1 Starting an Athapascan Program

The execution of an Athapascan program is handled by a "community." A community restructures a group of nodes (Inuktitut processes) so that they can be distributed to the different parallel machines. Therefore, prior to the declaration of any Athapascan object or task, a community must be created. Currently, this community only contains the group of nodes defined at the start of the application.

A community is created by executing the instruction:

```
a1::Community com = a1::System::create_initial_community( argc, argv );
```

Once the community has been created the following methods will be available:

- `com.is_leader()` : that returns true on one and only one of the nodes (processes) of that community
- `com.sync()` : that waits for all of the created tasks to finish executing on the collection of nodes before execution on this node resumes.
- `com.leave()` : indicates to the current process to leave that community

Usually a community is defined in the `main` method of the program. To ensure a proper creation of a commuunity, it is also necessary to catch any exceptions that might be thrown by the intialization procedures. The skeleton of an Athapascan program should resemble the following block of code.

```
int doit (int argc, char** argv)
{
  ...
  return 0;
}

int main(int argc, char** argv)
{
  try {
    a1::Community com =
     a1::System::create_initial_community(argc, argv);
    std::cout << "count argc" << argc << std::endl;
    if (argc != N) {
      for (int i=0; i<argc; ++i)
        std::cout << argv[i] << std::endl;
      std::cout << "usage: " << argv[0] << "PROPER USAGE "<<std::endl;
      return 0;
    }

    doit( argc, argv );
    com.leave();
  }
  catch (const a1::InvalidArgument& E) {
    std::cout << "Catch invalid arg" << std::endl;
  }
  catch (const a1::BadAlloc& E) {
    std::cout << "Catch bad alloc" << std::endl;
  }
  catch (const a1::Exception& E) {
   std::cout << "Catch : ";
   E.print(std::cout);
   std::cout << std::endl;
  }
  catch (...) {
    std::cout << "Catch unknown exception: " << std::endl;
  }

  return 0;
}
```

The function `doit` should contain the code to be executed in parallel. The function `main` in this case simply creates the community, executes `doit`, and catches any exceptions thrown. The variable N represents the constant number of inputs needed to run the program (when coding this line, replace N with the desired number). If there are not enough inputs specified at run-time, the program should terminate and output a message describing the proper usage of the program. It is necessary to execute the `com.leave()` function to facilitate the termination of the Athapascan application. Calling `doit` from `main` simplifies the code, making it easier to read.

Note: From this point on, all examples contained in this document will define a method `doit`. It is to be assumed that the program is executed with a `main`, as defined above. That is to say that the `main` method will not be shown in the examples.

4.1.2 Fork

Fork is the keyword used by Athapascan to define tasks that are to be parallelized. To Fork a task, one must:

- **write the code to be parallelized** (overload the “operator()” of the class to be Forked);
  syntax:

  ```
  struct my_Task {
    operator()( formal parameters )
      task body
    }
  } ;
  ```

- **invoke the task** (call to the method “Fork”);
  syntax:

  ```
  a1::Fork <my_task>()( effective parameters );
  ```

  Example:

  ```
  struct PrintHello {
    void operator()(int id ) {
         printf("Hello world from task number % n", id) ;
    }
  } ;

  int doit( int argc, char** argv) {
      ...
       a1::Fork < PrintHello > ()(i) ;
      ...
      return 0;
  }

  int main( int argc, char**argv )
  {
    // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

    return 0;
  }
  ```

  NB: All the formal parameters must be made communicable (Cf. Chapter **??**).

- **parameters:** A task parameter can be:

  1. a regular object or variable
     ex: a1::Fork<myTask> ()(class T), with T communicable).

  2. a Shared data that can be used by different tasks
     ex: a1::Fork< myTask > () (a1::Shared_r< myClass > T), with myClass communicable.
     A Shared data can have different access rights (Cf. Chapter **??**).

     NB: Shared data must be initialized (A runtime error occurs if this is not done).

  Example:

```
struct print_A_Shared {
  void operator()(a1::Shared_r< int > T ) {
       printf("The Shared data parameter has the value: % d", T.read())) ;
  }
} ;


int doit ( int argc, char** argv) {
    ...
    a1::Shared<int> myShared(new int(10)) ;
    a1::Fork < print_A_Shared > ()(myShared) ;
    ...
    return 0;
}

int main( int argc, char**argv )
{
  // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

- **communicable type**:
  Only serialized classes can be communicated. The standard classes and types (`short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `char`, `unsigned char`, `float`, `double`, `long double`, `char*`, `void*`, Standard container classes of the STL) are already communicable by default. User defined classes may be complex. It is therefore necessary to explain to the library how these classes should be serialized. These classes and types have to be explicitly made communicable by implementing specific class methods (Cf. Chapter **??**).

4.1.3 Shared Object

A task can access only the objects it has declared or received as effective parameters. The access to a shared object can be:

- **read only** (`a1::Shared_r<class T>`),
- **write only** (`a1::Shared_w<class T>`),
- **cumulative write** (`a1::Shared_cw<class F,class T>`),
- **read/write** (`a1::Shared_r_w<class T>`).

NB1: A shared object can implement only communicable classes.
NB2: A pointer given as parameter has to be considered as lost by the programmer (all further access through this pointer is invalid).

Declaration :

- `a1::Shared< T > x;` the reference `x` must be initialized before use.
- `a1::Shared< T > x( 0 );` the reference `x` can be used but no initial data is associated with it.
- `a1::Shared< T > x( new T( ... ) );` the reference `x` can be used and possesses an initial value.

NB: Be aware that non-initialized shared data is a common programming error that gives no compile-time warning.

Access Rights and Methods :

Each task specifies as a parameter the shared data objects that will be accessed during its execution and which type of access will be performed on them.

According to the requested access right, tasks should use this methods:

- read only access (`a1::Shared_r<T> x`):

  use `x.read()`
  prototype: `const T& read() const;`

- write only access (`a1::Shared_w<T> x`):

  use `x.write(p)`
  prototype: `void write(T*);`

  NB: Deallocation is made by Athapascan .

- read write access (`a1::Shared_r_w<T> x`):

  use `x.write(p)` or `x.access()`
  prototype: `void write(T*);`
  prototype: `T& access();`

- accumulation write access (`a1::Shared_cw<F,T> x`):

  use `x.cumul(v)`
  prototype: `void cumul( const T& );`

  NB: The call `x.cumul( v )` accumulates `v` into the shared data according to the accumulation function `F`. A copy of `v` may be made during the first accumulation if the data present is not yet valid.

  Example:

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

struct addToShared  {
  void operator()(a1::Shared_ cw<add, int> T, int i ) {
       T.cumul(i);
  }
} ;


int doit( int argc, char** argv) {
    ...
    a1::Shared<int> myShared(new int(10)) ;
    a1::Fork < addToShared > ()(myShared, 5) ;
    ...
    return 0;
}

int main( int argc, char**argv )
{
  // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

Conversion Rules :
Due to the several types of Shared data with specific reading and writing capabilities, there are restrictions on how Shared objects may be passed with respect to their access right. Since this material is rather extensive, Chapter **??** page **??** is devoted to the study of the Shared object and thus not covered in this chapter.

Adding thread information :
Some scheduling policies benefit from thread information. Athapascan uses four variables to determine the best method of ececution. Each datum has a default value but a better scheduling may be obtained if the programmer assigns significant values. The variable information retained is:

- The **cost** of the thread (a C++ double).
- The **locality** of the thread (a C++ int).
- The **priority** of the thread (a C++ int). Low values traduce higher priorities.
- An **extra** attribute (a C++ double) that represents whatever the scheduler wants it to represent.

This information is given at the thread creation:
`a1::Fork<user_task> (SchedAttributes( infos )) (<parameters>);`
Where `<infos>` represent the list of the four possible thread attributes. Here is an example of how to use the scheduling attributes:
`a1::Fork<user_task> (SchedAttributes( prio, cost, loc, extra )) (<parameters>);`

Note that if both a specific scheduler and some information are given,the order in which the variables are passed is not important:

```
a1::Fork<user_task> (SchedAttributes( prio, cost, loc, extra ), sched_group ) (<parameters>);
a1::Fork<user_task> (sched_group, SchedAttributes( prio, cost, loc, extra )) (<parameters>);
```

4.1.4 System Information

It is possible to get the following runtime information:

- `a1_system::node_count()` returns the number of nodes
- `a1_system::self_node()` returns the node identification number on which the function is invoked, an integer from 0 to `a1_system::node_count()-1`
- `a1_system::thread_count()` returns the number of a0 threads dedicated to a thread's execution on the virtual processor
- `a1_system::self_thread()` returns the a0 thread identification number that hosts the execution of the thread, an integer from 0 to `a1_system::thread_count()-1`

4.1.5 Compilation and Execution

The compilation of an Athapascan program is performed by the `make` command using the **Makefile** created upon installaion. Be sure to modify the **Makefile** as needed, to personalize the folders containing the include and library files.
An Athapascan program is executed in the same manner as one would execute any other program from the command line. For example a common execution may resemble: `sh ./program_name <inputs>`

4.2 Basic Examples

This section is a brief tutorial of how to build simple Athapascan programs; it proceeds by teaching through examples. The two examples presented in the section are getInfoTask.cpp (a program that demonstrates how to retrieve system information), and Fibonacci.cpp (a program which commputes the Fibonacci number of a given input). More thourough examples are offered in Chapter **??** but use concepts that have not been discussed thus far.

4.2.1 Simple Example 1: "getInfoTask.cpp" (1 Fork - 0 Shared)

Let's start with an example implementing the Athapascan keyword `Fork` (We will see later how to use `Shared`). Assume we want to print to the console data about the task execution, for example: which processor is involved, which node number, etc... Here is a basic example

code :

```
#include "athapascan-1.h"

struct getInfoTask {
        void operator()(int i) {
                cout << "Task number: " << i << endl;
                cout << "Node number" << a1_system::self_node()
                        << " out of " << a1_system::node_count() << endl;
                cout << "Thread number" << a1_system::self_thread()
                        << " out of " << a1_system::thread_count() << endl;
        }
};

int doit(int argc, char** argv)
{
    for (int i=0; i<10; i++) {
        a1::Fork<getInfoTask>()(i);
    return 0;
}

int main( int argc, char** argv )
{
   //MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

   return 0;
}
```

This program is very simple and shows you how to write a task. Fork is instanciated with the class getInfoTask and will execute the code of the method overloading operator().

Compiling :
Recall that compiling an ATHAPASCAN program is done by executing the `make` function from the command line. For the **getInfoTask** example execute:

```
sh> make getInfoTask
```

To execute this newly created program enter the following on the command line:

```
sh> ./getInfoTask
```

NB: To run a program build upon LAM-MPI (like the ATHAPASCAN library) or Inuktitut, you have to configure your cluster of machines so that they can run a "rsh" to each other.

4.2.2 Simple Example 2: "Fibonacci.cpp" (multiple Fork and Shared)

algorithm :
The Fibonacci series is defined as:

$$\begin{cases} F(0) = 0 \\ F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \quad \forall n \geq 2 \end{cases}$$

There are different algorithms to resolve Fibonacci numberds, some having a linear time complexity $\Theta(n)$. The algorithm we present here is a recursive method.

It is a very bad implementation as it has an exponential complexity, $\Theta(2^n)$, (as opposed to the linear time complexity of other algorithms). However, this approach is easy to understand and to parallelize.

Sequential implementation :
First, let's have a glance at the regular recursive sequential program:

```
#include <iostream.h>
#include <stdlib.h>

int fibonacci(int n) {
        if (n<2)
                return n;
        else
                return fibonacci(n−1)+fibonacci(n−2);
}

int main(int argc, char** argv)
{
        //make sure there is one parameter
        if (argc <= 1) {
                cout << "usage: fibonacci <N>" << endl;
                exit(1);
        }
        cout << "result= " << fibonacci(atoi(argv[1])) << endl;

        return 0;
}
```

Parallel implementation :
We assume that you wish to make this program parallel. An easy way to do it would be to use the same algorithm (recursive). Well, it's not that easy. Two reasons for that:

1. in the sequential program, we used a function while ATHAPASCAN only supports procedure (void function).

2. you can access shared data only from a task having this data as a parameter (ex: you can't display the value of a Shared from the main)

To write this parallel program, we will then need:

- a task doing the same job the sequential function was doing (recursive);
- a task to add the result of the two recursive calls to fibonacci;
- a task to print the result to `stdout`.

```
#include <athapascan−1>
#include <iostream>
#include <stdlib.h>

struct add {
        // This procedure sums two integers in shared memory and writes the
        //      result in shared memory.
        void operator() ( a1::Shared_r<int> a, a1::Shared_r<int> b, a1::Shared_w<int> c ) {
                c.write( new int( a.read() + b.read() ) );
        }
};

struct fibonacci {
        // This procedure recursively computes fibonacci(n), where n is an int, and
        // writes the result in the shared memory.
        void operator() ( int n, a1::Shared_w<int> res ) {
                if( n < 2 )
                        res.write( new int( n ) );
                else {
                        a1::Shared<int> res1( 0 );
                        a1::Shared<int> res2( 0 );

                        a1::Fork< fibonacci > () (n − 1 , res1 );
                        a1::Fork< fibonacci > () (n − 2 , res2 );

                        a1::Fork< add > () ( res1, res2, res );
                }
        }
};

struct print {
        // This procedure writes to stdout, the result of fibo(n), where n is an int, in the
        // shared memory.
        void operator() ( int n, a1::Shared_r<int> res ) {
                cout << "Fibonacci(" << n << ")= " << res.read() << endl;
        }
} ;


int doit(int argc, char** argv)
{
    a1::Shared<int> res = int(0) ;

    a1::Fork<fibonacci> () (atoi(argv[1]), res );
    a1::Fork<print> () (atoi(argv[1]), res );
    return 0;
}

int main( int argc, char** argv )
{
   //MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

   return 0;
}
```

Explanation: The purpose of this excercise is to share the Fibonacci computation with others processors. Therefore, we need to define a Shared data in order to hold the result and to create a task calling the Fibonacci function. The fibonacci task works the same way the sequential function works: first we check if $n < 2$; if not we make recursive calls to get $F(n-1)$ and $F(n-2)$.

The difference here is that we cannot directly access Shared data. Thus, we cannot add the two results directly, and therefore need temporary Shared variables and a specific task to add them.

An other parallel implementation using concurrent write :

The parallel implementation we have just studied runs correctly but is not very efficient. This program will run faster if we use one of Athapascan's features called "concurrent write". This kind of Shared data is designed so that every task can access the data and perform a concurrent `cumul` operation. This means less synchronization needs to take place and thus, less time is wasted waiting for other tasks to complete.
Ideally, the granularity of the tasks we wrote is not big since forking tasks can sometimes consume more CPU time than a regular sequentially executed task. The best way to increase the speed of the computation is then to Fork enough tasks for each processor to be busy, and then let them carry on sequentially to avoid excessive communication. For that purpose, we introduce a threshold variable. As indicated in the text, the threshold is user defined. Now examine the second parallel implementation of a Fibonacci number algorithm, Section **??** page **??**.

NB: The parallel programs we present here are just for educational purpose. The granularity of the tasks executed on remote nodes is small while the number of tasks is high. If you try to run these programs on different architecture in order to compare the performances, you will realize that it can take even more time to execute in parallel than sequentially. This is a normal behavior for these kinds of programs.

```
#include "athapascan-1.h"
#include <iostream.h>
#include <stdlib.h>

struct add {
        //this method is instanciated by the cumul method of
        //the concurrent write Shared data (see fibonacci)
   void operator()(int & x, const int& a ) { x += a ; }
};

//sequential fibonnaci
int fibo_seq( int n ){
   if( n<2 )
      return n;
   else
      return( fibo_seq( n-1 ) + fibo_seq( n-2 ) );
}

struct fibonacci {
        void operator()( int n, int threshold, a1::Shared_cw< add, int > r ){
      if( n <= threshold ) {
         r.cumul( fibo_seq( n ) );
      } else {
         a1::Fork<fibonacci> () ((n-2),threshold , r );
         fibonacci() ((n-1), threshold, r );
      }
   }
};

struct print {
        // This procedure writes to stdout, the result of fibo(n),
        //where n is an int, in the shared memory.
        void operator() ( int n, a1::Shared_r<int> res ) {
                cout << "Fibonacci(" << n << ")= " << res.read() << endl;
        }
} ;


int doit(int argc, char** argv)
{
    a1::Shared<int> res = int(0) ;

    a1::Fork<fibonacci> () (atoi(argv[1]), atoi(argv[2]), res );
    a1::Fork<print> () (atoi(argv[1]), res );
    return 0;
}

int main( int argc, char** argv )
{
   //MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

   return 0;
}
```

As you reach this point you should now be able to write, compile, and run a simple Athapascan based program. If you desire to write a real-life, more complex application you must further study the Athapascan library. There are many useful concepts that have not yet been introduced.

5 Tasks

The granularity of an algorithm is explicitly given by the programmer through the **creation** of **tasks** that will be **asynchronously** executed. A task is an object representing the association of a procedure and some effective parameters. Tasks are dynamically created at run time. A task (creation + execution) in Athapascan can be seen as a standard procedure call. The only difference is that the created task's execution is fully asynchronous, meaning the creator is not waiting for the execution of the created task to finish to continue with its own execution. An Athapascan program can be seen as a set of tasks scheduled by the system and distributed among nodes for its execution.

5.1 Task's Procedure Definition

A task corresponds to the execution of a C++ function object, *i.e.* an object from a class having the `void operator()` defined:

```
struct user_task {
  void operator() ( [...parameters...] ) {
     [... body ... ]
  }
};
```

A sequential (hence *not* asynchronous !) call to such a function class is written in C++:

```
user_task()  ( [... effective parameters ...] ) ;
  // user_task is executed according to depth-first ordering.
```

Pay attention to the type of the effective parameters when performing a sequential call of a struct. The type of the effective parameters must exactly match the type of the corresponding formal parameters! For example, a `Shared` parameter cannot be used where a `Shared_r` parameter is required. However when `Forking` a task, this behavior is allowed (see **??** page **??** to learn more about passing Shared parameters in Athapascan).

A task is an object of a user-defined type that is instantiated with Fork :

```
  a1::Fork< user_task > ()  ( [... effective parameters ...] ) ;
    // user_task is executed asynchronously.
    // The synchronizations are defined by the access on
    // the shared objects ; the semantic respects the
    // reference order.
```

Example The task `hello_world` displays a message on the standard output:

```
  struct hello_world {
    void operator() ( ) { // No parameters here
      cout << "Hello world !" << endl ;
    }
  } ;
 int  doit () {
    hello_world() () ; // immediate call (*not* asynchronous !)

    a1::Fork< hello_world > () (); // Creation of a task executed asynchronously.
    a1::Fork< hello_world > () (); // Creation of another task executed asynchronously.

    return 0;
  }
 int main( int argc, char**argv ) {
   // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

   return 0;
 }
```

Remark: encapsulating a C procedure. Obviously, a C procedure (i.e. a C function with `void` as return type) can be directly encapsulated in a C++ function class and thus parallized with Athapascan.Here is an example:

```
void f ( int i ) {
  printf( "what could I compute with %d ? \n", i );
}
struct f_encapsulated {
  void operator() ( int i ) {  /* i is some formal parameter */
    f( i );
  }
};
int doit() {
  int a = 3 ;
  ...
  f( a );                         // Sequential call to the function f
  f_encapsulated() ( a );         // Sequential call to the function class
                                  // f_encapsulated
  a1::Fork< f_encapsulated >() ( a ); // Asynchronous call to the function class
                                  // f_encapsulated
  ...
  return 0;
}
int main( int argc, char**argv ){
  // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

5.2 Allowed Parameter Types

The system has to be able to perform the following with a task in order to “Fork” it:

1. detect its interactions with the shared memory in order to be able to determine any synchronizations required by the semantic.
2. move it to the node where its execution will take place.

Here are the different kinds of parameters allowed for a task

- `T`, to designate a classical C++ type that does not affect shared memory. However this type must be communicable (see Chapter **??** entitled Shared Memory for a definition of communicable types).
- `Shared_...< T >` to designate a parameter that is a reference to a shared object located in the shared memory. `T` is the type of this object. It must be communicable.

In the case of a classical `T` class, the type `T` should not refer to the shared memory. For example, when initializing a shared object `a1::Shared< T > s(d);` where `d` points to an object of type `T` , this pointer should no longer be used in the program. [1].

5.3 Task Creation

To create an Athapascan task the keyword **a1::Fork** must be added to the usual procedure call. Here, user_task is a function class as described in the previous section:

C++ function object call	Athapascan task creation

`...`
`user_task() ( [args] );`
`...`
	`...`
	`a1::Fork< user_task >() ( [args] );`
	`...`

The new created task is managed by the system which will execute it on a virtual processor of its choice (*cf.* Appendix **??**).

5.4 Task Execution

The task execution is ensured by the Athapascan system. The following properties are respected:

- The task execution will respect the synchronization constraints due to the shared memory access;
- all the created tasks will be executed once and once only,
- no synchronization can occur in the task during its execution. Hence, for most but not all implementations of Athapascan programs the system guarantees that every shared data accessed for either reading or updating is available in the local memory before the execution of the task begins.

5.5 Kinds of Memory a Task Can Access

Each Athapascan task can access three levels of memory:

- the **stack**, a local memory private to the task. This local memory contains the parameters and local variables (it is the classical C or C++ stack).
  This stack is automatically deallocated at the end of the task.
- The **heap**, the local memory of the node (Unix process) that executes the task. Objects are allocated or deallocated in or from the heap directly using C/C++ primitives: `malloc`, `free`, `new`, `delete`...
  Therefore, all tasks executed on a given node share one common heap[2]: consequently, if a task does not properly deallocate the objects located in its heap, then some future tasks may run short of memory on this node.

[1] Caution: this is *not* verified, either at compile nor at execution time. The user has to take care of not including any reference on the shared memory in classical types.

[2] In the current implementation, the execution of a task on a node is supported by a set of threads that share the heap of a same heavy process representing the node.

- The **shared memory**, accessed concurrently by different tasks. The shared memory in Athapascan is a non-overlapping collection of physical objects (of communicables types) managed by the system.

6 Communicable Type

Using a distributed architecture means handling data located in shared memory (mapping, migration, consistency). In order to make Athapascan able to do this, the data must be serialized. This serialization has to be explicitly done by the programmer to suit the specific needs of the program.

NB: All the classes and types used as task parameters must be made communicable.

6.1 Predefined Communicable Types

The following types are communicable:

- The following C++ basic types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `char`, `unsigned char`, `float`, `double`, `long double`, `char*`[3], `void*`.
- all types from the STL[4].

Note that two generic functions for packing/unpacking an array of contiguous elements are provided:

```
a1::Ostream& a1_pack   ( a1::Ostream& out, const T* x, int size ) ;
a1::Istream& a1_unpack ( a1::Istream& in,        T* x, int size ) ;
```

Both functions require the number of elements. They are specialized for basic C++ types.

6.2 Serialization Interface for Classes Stored in Shared Memory

A communicable type `T` must have the following functions:

- The empty constructor: `T()`
- The copy constructor: `T( const T& )`
  NB: the copy shall not have any overlap with the source.
- The destructor: `~T()`
  NB: only one task executes the deallocation at a time.
- The "sending" operator, `a1::Ostream& operator<<( a1::Ostream& out, const T& x )` puts into the output stream `out` the information needed to reconstruct an image of `x` using the operator `>>`.
- The "receiving" operator, `a1::Istream& operator>>( a1::Istream& in, T& x )` takes from the input stream `in` the information needed to construct the object `x`; it allocates and initializes `x` with the value related to the information.
  Note that the system always calls this function with an object `x` that has been constructed with the empty constructor.

6.3 Examples

This section teaches through examples how to define a class or type as being communicable. The three examples provided in this section are Complex Numbers (which creates a simple communicable class), Basic List with Pointers (which generates a singly-linked list which behaves as a queue data structure), and Resizable Array (which generates a class for the creation of a list of dynamic size).

6.3.1 Example 1: Complex Numbers

For example, let us consider the complex numbers type. This type can be set "communicable" by simply implementing the four communication operators.
NB: Note that the C++ provided defaults can often be used to impliment the empty constructor, the copy constructor and the destructor. In the case of the `complex` type, the defaults operators are used (refer to a C++ guide to learn more about these defaults constructors).

[3] You must be careful when communicating pointers : in fact if your program is executed on several nodes (option -a0n 2 for example), the communication can be performed between the nodes and the pointer is often meaningless on other nodes.

[4] By default the system considers that all types possess iterators that run all over the data: this is the case of STL types. For all others, the necessary functions have to be provided to override this default definition.

```
/**
        class complex is an Athapascan-1 communicable class
        complex: z = x + i*y
        NB: This class implements only the methods needed
        by Athapascan-1.
*/
class complex {
public:
        double x;
        double y;

        //empty constructor
        complex() { x=0; y=0;}

        //copy constructor
        complex( const complex& z) { x=z.x; y=z.y;}

        //destructor
        ~complex() {}

};

//packing operator
a1::Ostream& operator<< (a1::Ostream& out, const complex& z) {
        out << z.x << z.y;
        return out;
}

//unpacking operator
a1::Istream& operator>> (a1::Istream& in, complex& z) {
        in >> z.x >> z.y;
        return in;
}
```

Figure 1: Making the user defined class, complex, communicable

6.3.2 Example 2: Basic List with Pointers

Let's go a bit deeper in the serialization and find out how to write a communicable class implementing a dynamic data structure based upon a list of pointers.

NB: Even though the container classes from the STL are optimized and have been made communicable, there is little use for these classes in a real-life Athapascan application. Therefore the class in this example is not optimized, it is just an example. This class implements a chain structure using pointers. When running parallel application on a cluster of machines, it is meaningless to communicate addresses. Therefore, we just communicate values:

The following class implements a chain structure using pointers. When running a parallel application on a cluster of machines, it is meaningless to communicate addresses. Therefore, only values are communicated:

- *a1::Ostream*: we send first the number of values, then the values themselves (but we don't send the pointers!);
- *a1::Istream*: we receive first the number of values, then we insert the values in the chain (using local pointers).

```
#include <iostream.h>
#include <stdlib.h>
#include "athapascan-1.h"

/**
   class myList is an Athapascan-1 communicable class
   We use a chain structure to store the values.
   NB: T has to be communicable too.
*/

template<class T>
class myList {
public:
   T value;
   myList* next;

   //empty constructor
   myList() : value(), next(0) {}

   //constructor
   myList(T v, myList* n): value(v), next(n) {}

   //copy constructor
   myList(const myList<T>& d) {
      value = d.value;
      if (d.next != 0) {
         next = new myList<T> (*(d.next));
      } else next = 0;
   }

   //destructor
   ~myList() {
      if (next != 0) delete next;
   }

   //return the size of the list
   int size() {
      int s(0);
      myList<T>* x = this;
      while(x->next != 0) {
         s++;
         x = x->next;
      }
      delete x;
      return s;
   }

   //we push the data at the end of the list
   void push_back(T newval) {
      myList<T>* x = this;
      while(x->next != 0) x = x->next;
      x->next = new myList(newval, 0);
   }

   //we pop th efirst data from the list, remove it end return its value
   T pop_front() {
      if (!next) return -1;
      else {
         myList<T>* x = next;
         T ret = next->value;
         next = x->next;
         x->next = 0;
         delete x;
         return ret;
      }
   }
};

//packing operator
template<class T>
```

```
a1::Ostream& operator << (a1::Ostream& out, const myList<T>& z) {
   myList<T> x = z;
   out << x.size();
   while(z.next != 0) {
      out << z.value;
      x = *(x.next);
   }
   out << x.value;
   return out;
}

//unpacking operator
template<class T>
a1::Istream& operator >> (a1::Istream& in, myList<T>& z) {
   int size;
   int temp(0);

   in >> size;
   for(int i(0); i<size; i++) {
      in >> temp;
      z.push_back(temp);
   }
   return in;
}


//test tasks to see if the class is communicable
struct myTaskW {
   void operator()(a1::Shared_r_w<myList<int> > x) {
      for(int i(0); i<100 ; i++) {
         x.access().push_back(i);
      }
   }
};

struct myTaskR {
   void operator()(a1::Shared_r_w<myList<int> > x) {
      myList<int> z = x.access();
      while (z.next!=0) {
         cout << z.pop_front() << " ";
      }
   }
};

int doit( int argc, char** argv )
{
   a1::Shared<myList<int> > x(new myList<int>());
   a1::Fork<myTaskW> () (x);
   a1::Fork<myTaskR> () (x);

   return 0;
}
```

6.3.3 Example 3: Resizable Array

A simple example of a dynamic structure is a mono-dimensional array with two fields: a size 'size' and a pointer to an array with 'size' number of elements.

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include "athapascan-1.h"

/**
   class myArray is an Athapascan-1 communicable class
   implementing a resizable myArray.
   NB: T has to be communicable too.
*/

template<class T >
class myArray {
  public:
    unsigned int size; // The size of the myArray.
    T* elts;           // ith entry is accessed by elts[i].

   //empty constructor
   myArray() : size(0), elts( 0 ) {}

   //constructor
   myArray(unsigned int k) : size(k) {
      if (size==0) elts=0;
      else elts = new T [size];
   }

   //copy constructor
   myArray(const myArray<T>& a) {
      size = a.size;
      elts = new T [size];
      for (int i=0; i<size; i++) elts[i] = a.elts[i] ;
}

   //destructor
   ~myArray() { delete [] elts ; }

   //resize the myArray
   void resize(unsigned int newSize);

};

// Packing operator
template<class T >
a1_ostream& operator<<( a1_ostream& out, const myArray<T>& z ) {
   out << z.size ;
   for (int i = z.size-1; i>=0 ; i--) out << z.elts[i] ;
   return out;
}
// Unpacking operator
template<class T >
a1_istream& operator>>( a1_istream& in, myArray<T>& z ) {
   in >> z.size ;
   z.elts = new T [ z.size ] ;
   for (int i = z.size-1; i>=0 ; i--) in >> z.elts[i] ;
   return in;
}

void myArray<T>::resize(unsigned int newSize) {
      //erasing the data
      if (newSize <= 0) {
         size = 0;
         if (elts != NULL) {delete elts; elts=0;}
         return;
      }

      //new myArray is smaller
    if (newSize <= size) {
      T* newtab = new T[newSize];
      memmove(newtab, elts, newSize*sizeof(T));
      delete elts;
      elts = newtab;
      size = newSize;
      return ;
    }

    //then new myArray is bigger
    T* newtab = new T[newSize];
    memmove(newtab, elts, size * sizeof(T));
    delete elts;
    elts = newtab;
    size = newSize;
    return ;
}

//test tasks to see if the class is communicable
struct myTaskW {
  void operator()(a1::Shared_r_w<myArray<int> > x) {
    int z = x.access().size;
    for(unsigned int i(0); i < z ; i++) {
      x.access().elts[i] = 2*i;
    }
  }
};

struct myTaskR {
  void operator()(a1::Shared_r_w<myArray<int> > x) {
    cout << "size of the array: " << x.access().size << endl;
  }
};

struct resizeTab {
  void operator()(a1::Shared_r_w<myArray<int> > x, unsigned int n) {
    x.access().resize(n);
  }
};

int doit( int argc, char** argv )
{

  a1::Shared<myArray<int> > x(new myArray<int>(100));
  a1::Fork<myTaskW> () (x);
  a1::Fork<myTaskR> () (x);
  a1::Fork<resizeTab> () (x, 10);
  a1::Fork<myTaskR> () (x);

  return 0;
}
int main( int argc, char** argv )
{
  //MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

7 Shared Memory: *Access Rights* and *Access Modes*

Shared memory is accessed through typed references. One possible type is `Shared`. The consistency associated with the shared memory access is that each "read" sees the last "write" according to the lexicographic order.

Tasks **do not make any side effects** on the shared memory of type `Shared`. Therefore they can only access the shared data on which possess a reference. This reference comes either from the declaration of some shared data or from some effective parameter. A reference to some shared data is an object with the following type: `a1::Shared_RM < T >`. The type `T` of the shared data must be communicable (see Section **??** page **??**). The suffix `RM` indicates the access right on the shared object (read – `r` –, write – `w` – or cumul – `c` –) and the access mode (local or postponed – `p` –) `RM` can be one of the following suffixes:
`r`, `rp`, `w`, `wp`, `cw`, `cwp`, `r_w` and `rp_wp`.
Access rights and modes are respectively described in section **??** page **??** and **??** page **??**.

7.1 Declaration of Shared Objects

If `T` is a communicable type, the declaration of an object of type `a1::Shared<T>` creates a new shared datum (in the shared memory managed by the system) and returns a reference to it.

Depending on whether the shared object is initialized or not, three kinds of declarations are allowed:

- `a1::Shared< T > x( new T( ... ) );`
  The reference `x` is initialized with the value pointed to by the constructor parameter. Note that the memory being pointed to will be deallocated by the system and should not be accessed anymore by the program. `x` can not be accessed by the task that creates it. It is only possible to Fork other tasks with `x` as an effective parameter.
  Example:

```
a1::Shared<int> x ( new int( 3 ) ) ;
  // x is initialized with the value 3.

double* v = new double ( 3.14 ) ;
a1::Shared<double> sv ( v ) ;
  // sv is initialized with the value v;
  // v can not  be used anymore in the program
  // and will be deleted by the system.
```

- `a1::Shared< T > x( 0 );`
  The reference `x` is declared but not initialized. Thus, the first task that will be forked with `x` as parameter will have to initialize it, using a write statement (**??** page **??**). Otherwise if a task recieves this reference as a parameter and attempts to read a value from it, dead-lock will occur.
  Example:

```
a1::Shared<int> a (new int(0) ) ;
  // a is a shared object initialized with the value 0.
a1::Shared<int> x ( 0 ) ;
  // x is a NON initialized shared object.
```

- `a1::Shared< T > x;`

  The reference `x` is only declared as a reference, with no related value. `X` therefore has to be assigned to another shared object before forking a task with `x` as a parameter. Such an assignment is symbolic, having the same semantics as pointer assignment.

  Example:

```
a1::Shared<int> x ;
  // x is just a reference, not initialized.
a1::Shared<int> a (new int(0) );
  // a is a shared object initialized with the value 0.
x = a ;
  // x points to the same shared object as a.
```

The following operations are allowed on an object of type `Shared`:

- Declaration in the stack, as presented above.
- Declaration in the heap, using the operator `new` to create a new shared object. In the current implementation, the "link" between a task and a shared data version is made through the C++ constructor and destructor of the shared object. So, to each construction must correspond a destruction, else dead-lock may occur. Therefore, in the case of allocation in the heap, the `delete` operator corresponding to the already exectured `new` has to be performed.
- Affectation: a shared object can be affected from one to another. This affectation is symbolic, having the same semantics as pointer affectation. The "real" shared object is then accessed through two distinct references.

7.2 Shared Access Rights

In order to respect the sequential consistency (lexicographic order semantic), Athapascan has to identify the value related to a shared object for each read performed. Parallelism detection is easily possible in the context that any task specifies the shared data objects that it will access during its execution (on-the-fly detection of independent tasks), and which type of access it will perform on them (on-the-fly detection of a task's precedence). Therefore, an Athapascan task can not perform side effects. All manipulated shared data must be declared in the prototype of the task. Moreover, to detect the synchronizations between tasks, according to lexicographic semantic, any shared parameter of a task is tagged in the prototype of `t` according to the access performed by `t` on it. This tag indicates what kind of manipulation the task (and, due to the lexicographic order semantics, all the sub-tasks it will fork) is allowed to perform on the shared object. This tag is called the access right; it appears in the prototype of the task as a suffix of the type of any shared parameter. Four rights can be distinguished and are presented below: read, write, update and accumulation.

7.2.1 Read Right: `Shared_r`

`a1::Shared_r< T >` is the type of a parameter thats value can only be read. This reading can be concurrent with other tasks referencing this shared object in the same mode.

In the prototype of a task, the related type is:
`a1::Shared_r< T > x`
Such an object gets the method:
`const T& read () const ;`
that returns a constant reference to the value related to the shared object `x`.

For example, using the Class `complex` that is defined in **??**:

```
class print {
  void operator() ( a1::Shared_r< complex > z ) {
     cout << z.read().x << " + i." << (z.read()).y ;
  }
} ;
```

7.2.2 Write Right: `Shared_w`

`a1::Shared_w< T >` is the type of a parameter whose value can only be written. This writing can be concurrent with other tasks referencing this shared data in the same mode. The final value is the last one according to the reference order. In the prototype of a task, the related type is:
`a1::Shared_w< T > x`

Such an object gets the method:
`void write ( T* address ) ;`
that assigns the value pointed to by `address` to the shared object.

This method assigns the value pointed to by `address` to the shared object. No copy is made: the data pointed by <address> must be considered as lost by the programmer. Further access via this address are no more valid (in particular, the deallocation of the pointer: it will be performed by the system itself).

Example:

```
class read_complex {
  void operator() ( a1::Shared_w< complex > z ) {
     complex* a = new complex ;
     cin >> a.x >> a.y ;
     z.write ( a ) ;
  }
} ;
```

Note To clarify the rule that each `read` "sees" the last `write` due to lexicographical order being observed, follow the example below:

```
#include "athapascan-1.h"
#include <stdio.h>

struct my_read {
  void operator()( a1::Shared_r<int> x ) {
    printf( " x=%i\n", x.read() );
  }
};

struct my_write {
  void operator()( a1::Shared_w<int> x, int val ) {
    x.write( new int(val) );
  }
};

int doit( int argc, char** argv )
{
  a1_system::init(argc, argv);
  a1_system::init_commit();
  if( a1_system::self_node() == 0 ) {
    a1::Shared<int> i( new int( 1 ) );
    a1::Fork<my_write>()( i, 1 ); // line A
    a1::Fork<my_read>()( i ); // line B
    a1::Fork<my_write>()( i, 2 ); // line C
    a1::Fork<my_read>()( i ); // line D
    a1::Fork<my_write>()( i, 3 ); // line E
    a1::Fork<my_read>()( i ); // line F
  }
  a1_system::terminate();

  return 0;
}

int main( int argc, char** argv )
{
   //MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

   return 0;
}
```

It is possible that the operations in line `E` and then in line `F` will execute before the preceeding lines because the rule described above is not broken. So do not be surprised to see the following result on the screen:

```
x=3
x=2
x=1
```

Keep this in mind, especially when measuring the time of computations. In that case of adding some extra synchronization variables to the code. But be careful because this can decrease the efficiency with which the program runs.

7.2.3 Update Right: Shared_r_w

a1::Shared_r_w< T > is the type of a parameter thats value can be updated in place; the related value can be read and/or written. Such an object represents a critical section for the task. This mode is the only one where the address of the physical object related to the shared object is available. It enables the user to call sequential codes working directly with this pointer.

In the prototype of a task, the related type is:
a1::Shared_r_w< T > x
Such an object gets the method:
T& access ( ) ;
that returns a reference on the data contained by the shared referenced by x. Note that &(x.access()) is constant during all the execution of the task and can not be changed by the user.

Example:

```
class incr_1  {
  void operator() ( a1::Shared_r_w< int > n ) {
     n.access() = n.access() + 1 ;
  }
}
```

7.2.4 Accumulation Right: Shared_cw

a1::Shared_c< T > is the type of a parameter whose value can only be accumulated with respect to the user defined function class F. F is required to have the prototype:

```
struct cumul_fn {
  void operator() ( T& x, const T& y ) {
    ... // body to perform x <-- accu(x, y)
  }
} ;
```

Example:

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

The resulting value of the concurrent write is an accumulation of all other values written by a call to this function. After the first accumulation operation is executed, the initial value of x becomes either the previous value or remains the current value, depending on the lexicographic access order. If the shared object has not been initialized, then no initial value is considered. Since an accumulation enables a concurrent update of some shared object, the accumulation function F is assumed to be both *associative* and *commutative*. Note that only the accumulations performed with respect to a same law F can be concurrent. If different accumulation functions are used on a single shared datum, the sequence of resulting values obeys the lexicographic order semantics.

In the prototype of a task, the related type is:
a1::Shared_cw< F, T > x
Such an object gets the method:
void cumul (T& v) ; that accumulates (according to the accumulation function F) v in the shared data referenced by x. For the first accumulation a copy of v may be taken if the shared data version does not contain some valid data yet.

Example:

```
// A generic function class that performs
// the multiplication of two values.
template < class T >
class multiply {
  void operator( T& x, const T& val ) {
    x = x * y ;
  }
} ;

// A task that multiplies by 2 a shared object
class mult_by_2  { //
  void operator() ( a1::Shared_cw< multiply<int>,  int > x) {
     x.cumul ( new int(2) ) ;
  }
} ;
```

Note
Keep in mind that a program written in ATHAPASCAN can benefit at run-time from the associative and communative properties of the accumulation function `F`. It is therefore possible that the execution of the code in Figure **??** will result in:

```
x=3 val=2
x=5 val=1
```

```
#include "athapascan-1.h"
#include <stdio.h>

struct F {
  void operator()( int & x, const int & val ) {
      printf( " x=%i val=%i\n", x, val );
      x += val;
   }
};

struct add {
  void operator()( a1::Shared_cw<F,int> x, int val ) {
      x.cumul( val );
   }
};

int doit( int argc, char** argv )
{
   a1::Shared<int> i( new int( 1 ) );
   a1::Fork<add>()( i, 2 );
   a1::Fork<add>()( i, 3 );
   return 0;
}

int main( int argc, char** argv )
{
   //MAIN METHOD AS PREVIOUSLY DEFINED IN THE API CHAPTER

   return 0;
}
```

Figure 2: Demonstration of associativity and commutativity of cumulative mode

It may seem as though the program was implemented according to the sequential depth-first algorithm:

```
......
a1::Shared<int> i( new int( 3 ) );
```

```
    a1::Fork<add>()( i, 2 );
    a1::Fork<add>()( i, 1 );
    ......
```

This is not the case. Naturally the above code is semantically correct as well and could produce the same result as the program in ??. It is therefore important to realize that since the function `F` is associative and commutative, the precise manner in which the reductions are performed cannot be predicted, even in the case where initial values are known.

7.3 Shared Access Modes

In order to improve the parallelism of a program when only a reference to a value is required - and not the value itself - Athapascan refines its access rights to include *access modes*. An access mode categorizes data by restricting certain types of access to the data. By default the access mode of a shared data object is "immediate", meaning that the task may access the object using any of the `write, read, access` or `cumul` methods during its execution. An access is said to be "postponed" (access right suffixed by p) if the procedure will not directly perform an access on the shared data, but will instead create other tasks that may access the shared data. In functional languages, such a parallelism appears when handling a reference to a future value.

With this refinement to the access rigths, Athapascan is able to decide with greater precision whether or not two procedures have a precedence relation. A procedure requiring a shared parameter with a direct read access, `r`, has a precedence relation with the last procedure to take this same shared parameter with a write access. However, a procedure taking some shared parameter with a postponed read access, `rp` , has no precedence relation. It is guaranteed by the access mode that no access to the data will be made during the execution of this task. The precedence will be delayed to a sub-task created with a type `r` . In essence, the type Shared can be seen as a synonym for the type `a1::Shared_rp_wp<T>`; it denotes a shared datum with a read-write access right, but on which no access can be locally performed. An object of such a data type can thus only be passed as an argument to another procedure.

7.3.1 Conversion Rules

When forking a task `t` with a shared object `x` as an effective parameter, the access right required by the task `t` has to be owned by the caller. More precisely, the Figure **??** page **??** enumerates the compatibility, at task creation, between a reference on a shared object type and the formal type required by the task procedure declaration. Note that this is available only for task creation, and not for standard function calls where the C++ standard rules have to be applied.

type of *formal parameter*	required type for the *effective parameter*
a1::Shared_r[p] < T >	a1::Shared_r[p] < T > a1::Shared_rp_wp < T > a1::Shared< T >
a1::Shared_w[p] < T >	a1::Shared_w[p] < T > a1::Shared_rp_wp < T > a1::Shared< T >
a1::Shared_cw[p]< F,T >	a1::Shared_cw[p] < F, T > a1::Shared_rp_wp < T > a1::Shared< T >
a1::Shared_rp_wp < T >	a1::Shared_rp_wp < T > a1::Shared< T >
a1::Shared_r_w < T >	a1::Shared_rp_wp < T > a1::Shared< T >

Figure 3: Compatibility rules to pass a reference on some shared data as a parameter to a task.

7.3.2 `Shared` Type Summary

Figure **??** page **??** summarizes the basic properties of references on shared data.

Reference type	decl.	formal P	effectif P	read	write	cumul	modif	concurrent
`a1::Shared_r      < T >`		●	●	●				●
`a1::Shared_rp     < T >`		●	●	○				●
`a1::Shared_w      < T >`		●	●		●			
`a1::Shared_wp     < T >`		●	●		○			
`a1::Shared_cw     < F,T >`		●	●			●		●
`a1::Shared_cwp    < F,T >`		●	●			○		●
`a1::Shared_r_w    < T >`		●		●	●		●	
`a1::Shared_rp_wp  < T >`		●	●	○	○		○	
`a1::Shared        < T >`	●		●	○	○		○	

Figure 4: Figure 6.3: Available types (and possible usages) for references on shared data. A● stands for a direct property and a ○ for a postponed one. *formal P* denotes formal parameters (type given at task declaration) and *effective P* denotes effective ones (type of object given at the task creation). *concurrent* means that more than one task may refer to the same shared data version.

7.4 Example: A Class Embedding ATHAPASCAN Code

A good way to write ATHAPASCAN applications is to hide ATHAPASCAN code in the data structures. Proceeding that way will allow you to keep your main program free from parallel instructions (making it easier to write and understand). In Chapter 5.3.3 we wrote a communicable class implementing a resizable, communicable array called myArray. We are now going to write a shared data structure on top of this class.

```
#include "athapascan-1.h"
#include "resizeArray.h"

/**
   class shared_array is a class hiding Athapascan code so that
   the main code of the application could be written as if it was
   sequential. It is based upon the resizable array class called
   myArray
*/

//resize the shared array
template<class T>
struct resize_shared_array {
   void operator() (a1::Shared_r_w<myArray<T > > tab, unsigned int size) {
      tab.access().resize(size);
   }
};

//affect a local myArray to a shared_array
template<class T>
struct equal_shared_array {
      //NB: we use a read/write access because we need the size
      void operator() (a1::Shared_r_w<myArray<T > > shtab, myArray<T> tab) {
      myArray<T>* t = new myArray<T>(tab);
      t−>resize(shtab.access().size);
      shtab.access() = *t;
   }
};

//append a shared array to a shared array
template<class T>
struct append_shared_array {
   void operator() (a1::Shared_r_w<myArray<T > > t1, a1::Shared_r<myArray<T > > t2) {
      int i = t1.access().size;
      int k = i + t2.read().size;

      t1.access().resize( k );
      for (int j(i); j<k; j++) {
         t1.access().elts[j] = t2.read().elts[j−i];
      }
   }
};

//swap two elements of a shared array
template<class T>
struct swap_shared_array {
   void operator() (a1::Shared_r_w<myArray<T > > tab, int i1, int i2) {
      T temp;

      temp = tab.access().elts[i1];
      tab.access().elts[i1] = tab.access().elts[i2];
      tab.access().elts[i2] = temp;
   }
};

//print the data of a shared array to standard output
template<class T>
struct ostream_shared_array {
   void operator() (a1::Shared_r<myArray<T > > tab) {
      unsigned int size = tab.read().size;
      for (int i(0); i<size; i++) cout << tab.read().elts[i] << " ";
   }
};

template<class T>
class shared_array : public a1::Shared<myArray<T > > {
public:
   //constructors
   shared_array() : a1::Shared<myArray<T > >(new myArray<T>()) {}
   shared_array(unsigned int size) : a1::Shared<myArray<T > >(new myArray<T>(size)) {}

   void resize(unsigned int newSize) {
      a1::Fork<resize_shared_array<T> >() ((a1::Shared<myArray<T> >&) *this, newSize);
   }

   void operator= (const myArray<T> &a) {
      a1::Fork<equal_shared_array<T> >() ((a1::Shared<myArray<T> >&) *this, a );
   }

   void append(shared_array &t2) {
      a1::Fork<append_shared_array<T> > () ((a1::Shared<myArray<T> >&) *this, (a1::Shared<myArray<T> >&) t2);
   }

   void swap(int i1, int i2) {
      a1::Fork<swap_shared_array<T> > () ((a1::Shared<myArray<T> >&) *this, i1, i2);
   }
};

// ostream operator
template<class T >
ostream& operator<<( ostream& out, const shared_array<T>& z ) {
   a1::Fork<ostream_shared_array<T> > () ((a1::Shared<myArray<T> >) z);
   return out;
}
```

The following main file tests the shared class. As you can see, there is no more reference to specific parallel code.

```
#include "athapascan-1.h"
#include "sharedArray.h"

#define SIZE 100

int doit( int argc, char** argv )
{
      shared_array<int> t1(10), t2(20);
      myArray<int> tab(SIZE);

      //fill the array
      for (int i(0); i< SIZE; i++) {
```

```
      tab.elts[i] = i;
    }

    //resize the shared array to test the methods
    t1.resize(SIZE);

    //move the data to the shared array
    t1 = tab;

    //try to swap a data
    t1.swap(2, 27);

    //append another shared array
    t2 = tab;

    t1.append(t2);
    cout << t1 << endl;
  }

  return 0;
}

int main( int argc, char** argv )
{
  //MAIN METHOD AS PREVIOUSLY DEFINED

  return 0;
}
```

8 Other Global Memory Paradigm

Access to shared data involve task synchronization: tasks are unable to perform side effects.

In some applications like Branch&Bound, it's conveniant to share a variable with all other tasks, data that can be read and written by anybody. This variable usually contains the value of a reached minimum or maximum. No information with respect to another task's activity is associated with this variable.
We are currently in the process of finishing the implementation of global variables for Athapascan-1 (variables that can be both read and written on the collection of nodes in a community without the constraints of data dependancy that exist for Shareds). Please bare with us as this project is still in development.

8.1 Memory Consistencies

The system offers three different consistencies on this memory:

- A Causal-Consistency where the data consistency is maintained along the paths of the precedency graph. That is to say that if the task T_1 preceeds the task T_2 in the precedency graph, then the modification on the memory made by T_1 will be seen by T_2.
- A Processor Consistency where the data consistency is maintained among the virtual processors of the system. That is to say that the order of modification of the memory on a virtual processor P_1 is the same that the order of modification seen on an other virtual processor P_2.
- An Asynchronous Consistency, where the data consistency is maintained on the system in its globality. That is to say that each modification made on one virtual processor will eventually be seen on other virtual processors.

8.2 Declaration

The declaration of a global data has the following prototype:

- `a1_mem_cc< T > x( T* pval );`
  for causal consistency
- `a1_mem_pc< T > x( T* pval );`
  for processor consistency
- `a1_mem_ac< T > x( T* pval );`
  for asynchronous consistency

The type `T` must be communicable and `pval` assigns a pointer to the initial value assumed by the object. This pointer can be null and is entirely managed by the system. That is to say, the pointer must be considered as lost by the programmer. This declaration is permitted anywhere in the code. An object of type `a1_mem` can be used as a parameter of a task or declared globally. If recieved as a parameter, the scope of the vaiable is limited to the procedure's body whereas it has the scope of the entire code if it is declared globally.

8.3 Registration

Due to some implementation characteristics, the global data have to be registered, effectively linking all the representatives (one per processor) to the same global data. If the object is used in the task parameters, this registration is made automatically. Otherwise if the object is declared globally the registration must be manually performed. Manually registering global data is done by invoking the `register( pval )` method on each object during the initialization phase, between the `a1_system::init()` and `a1_system::init_commit()` invocation. The `pval` parameter assigns a pointer to the initial value taken by the object. This pointer can be null and is entirely managed by the system, that is to say must be considered as lost by the programmer. The order of invocation must be the same [5] on all virtual processors.

[5] The result of registration is to associate an unique identifier to the object. This identifier result of an incrementation of a variable locale to the virtual processor. So two object are considered as identicall if their identifier are equal, that is to say if they have been registered at the same rank.

8.4 Usage

Three operations are allowed on an `a1_mem` object `x` representing a data of type `T`:

- The call `x.read()` returns a constant reference on the data located in `x`
- The call `x.write( pval )` writes the value pointed to by `x`. The pointer `T* val` must be considered as lost by the programmer.
- The call `x.fetch_and_op( int (*f)( T& a, const T& b ), val)`, where the object `val` is of type `const T&` and `f` designates a pointer to the funtion to be performed. The first parameter of this function will be the data stored in `x` and the second will be stored in `val`. The result of this function should be 1 if the data have been modified, otherwise it should be 0.

8.5 Destruction

The destruction of `a1_mem` objects is managed by the system and occurs:

- When no task possess it, for objects used as parameter
- At the end of the program execution, for objects that have been registered.

8.6 Example

The following example, Figure **??** shows the basic usage of `a1_mem` objects. The `complex` type is communicable and has been previously defined in Chapter **??** page **??**.

```
a1_mem_cc< int > x( 0 );
a1_mem_pc< complex > z( 0 );

int min( int& a, const int& b ) {
   if( a<b )
      return 0;
   a= b;
   return 1;
}

task T1( a1_mem_ac< double > f ) {
   f.fetch_and_op( &min, 0.01 );
}

task T2( ) {
   if( x.read() > 5 )
      z.write( new complex( 1.2, 2.5 ) );
}

int a1_main( int argc, char* argv[] )
{
   a1_system::init();
      x.register( new int( 1 ) );
      y.register( new complex );
   a1_system::init_commit();

   ...

   a1_system::terminate();
   return 0;
}
```

Figure 5: Basic usage of `a1_mem` objects.

8.7 Consideration on Global Data

Global data permits side effects to occur, therefore, the guarantee of a sequential execution is not maintained if these data are used.

9 Examples

In this chapter, we preset several complete examples of Athapascan programs. These examples are simple enough to be extensively presented within the confines of this chapter and complex enough to demonstrate the implementation of Athapascan in the context of real-world applications.

All these examples come with the library distribution.

9.1 Simple Parallel Quicksort

The aim of this implementation is to sort an array of data on two processors using an algorithm based upon a pivot. This implementation uses the class myArray (a resizable array as defined in **??** page **??**) as well as the `shared_array` class (as defined in **??** page **??**).

What we wish to show here is how to embed parallel instructions in the classes representing a user's data structures. Programming this way makes the main application code a lot more simple to understand and to write.

The idea of the algorithm is:

1. to split the array in two parts;
2. to sort each array in parallel (using qsort);
3. to split those arrays in two parts: elements <pivot and elements > pivot;
4. to merge the arrays containing data<pivot (and data>pivot);
5. to append the second array to the first one;

mySharedArray.h

```
#include "sharedArray.h"

//qsort the array
template<class T>
struct qsort_my_shared_array {
   void operator() (a1::Shared_r_w<myArray<T > > t) {
      t.access().qsort();
   }
};

//find a pivot
template<class T>
struct fPiv_my_shared_array {
   void operator() (a1::Shared_r<myArray<T > > t, a1::Shared_w<T> pivot) {
      int middle = t.read().size / 2;
      T *result = new T((t.read().elts[middle] + t.read().elts[middle+1])/2);
      pivot.write(result);
   }
};

//copy part of a myArray to a my_shared_array
template<class T>
struct copy_my_shared_array {
   void operator() (a1::Shared_w<myArray<T > > t,
                    myArray<T> from,
                    int start,
                    int size)
   {
      int sizeMyArray = from.size;
      //error control
      //(is size of shared big enough to receive data from array?)
      if (size > sizeMyArray) {
         cerr << "ERROR copy: size of shared_array < size of myArray" ;
         cerr << endl;
         exit(1);
      }
      myArray<T> *temp = new myArray<T>(size);
      for (int i(0); i< size ; i++) temp−>elts[i] = from.elts[i+start];

      t.write(temp);
   }
};
//copy part of a my_shared_array to a my_shared_array
template<class T>
struct copy2_my_shared_array {
   void operator() ( a1::Shared_r_w<myArray<T > > t,
                     a1::Shared_r<myArray<T> > from,
                     int start,
                     int size)
   {
      int sizeMyArray = from.read().size;
      //error control (is size of shared big enough to receive data?)
      if (size > sizeMyArray) {
         cerr << "ERROR copy: size of dest < size of source" << endl;
         exit(1);
      }
      for (int i(0); i< size ; i++) t.access().elts[i] = from.read().elts[i+start];
   }
};

template<class T>
class my_shared_array : public shared_array<T> {
   public:
      //constructor
      my_shared_array() : shared_array<T>() {}
      my_shared_array(unsigned int size) : shared_array<T>(size) {}

      //sort the array
      void qsort() {
         a1::Fork<qsort_my_shared_array<T> >()((a1::Shared<myArray<T> >&) *this);
      }

      //find a pivot and split
      void findPivot(a1::Shared<T> p) {
         a1::Fork<fPiv_my_shared_array<T> >() ((a1::Shared<myArray<T> >&) *this, p);
      }

      void operator= (const myArray<T> &a) {
         shared_array<T>::operator=(a);
      }

      //copy
      void copy(const myArray<T> &a, int start, int size) {
         a1::Fork<copy_my_shared_array<T> >() (
                        (a1::Shared<myArray<T> >&) *this,
                        a,
                        start, size);
      }
      void copy(const my_shared_array<T> &a, int start, int size) {
         a1::Fork<copy2_my_shared_array<T> >() (
                        (a1::Shared<myArray<T> >&) *this,
                        (a1::Shared<myArray<T> >&) a,
                        start,
                        size);
      }

      //merge arrays < to a certain value, so that they are sorted (Not optimized)
      void merge(my_shared_array<T> &a) {
        this->append(a);
        this->qsort();
    }
};
```

mainQSORT.cpp

```
#include "mySharedArray.h"

void usage() {
  cerr << "usage: qsort <size_of_array>" << endl;
  exit(1);
}

template<class T>
void myQsort(myArray<T> *t) {
  int size = t->size;
  int half1 = abs(size/2);

  //we split the array in 2 smaller my_shared_arrays
  my_shared_array<int> *t1, *t2;
  t1 = new my_shared_array<int> (half1);
  t2 = new my_shared_array<int> (size − half1);
  *t1 = *t;
  t2->copy(*t, half1, size−half1);

  //we sort the 2 shared_array using the standard C qsort()
  t1->qsort();
  t2->qsort();

  //we search for a pivot
  a1::Shared<T> pivot = new T();
  t1->findPivot(pivot);

  //we split each shared array in 2 pieces: <pivot and >pivot
  //<pivot stay in initial shared array
  //>pivot goes to a new array
  int half12 = abs(half1/2);
  my_shared_array<int> *t3, *t4;
  t3 = new my_shared_array<int> (half12);
  t4 = new my_shared_array<int> (/*size - half1 - */half12);
  t3->copy(*t1, half1 − half12, half12);
  t4->copy(*t2, (size − half1) − half12, half12);
  t1->resize(half1 − half12);
  t2->resize(size − half1 − half12);

  //we merge arrays< (>) pivot together
  t1->merge(*t2);
  t3->merge(*t4);

  //we append the second array to the first one and that's it
  t1->append(*t3);

  //print the result to stdout
  cout << endl << "qsort=>" << *t1 << endl;
}

int doit( int argc, char** argv ) {

    //set the scheduling to work_stealing
    a1_base_group* sch;
    sch = new a1_work_steal::basic;
    a1_set_default_group(*sch);

    //first, for testing purpose, we fill an array with randomized data
    int size =atoi(argv[1]);
    myArray<int> *t = new myArray<int> (size);
    for(int i(0); i < size; i++) t->elts[i] = rand()%10;

    //then we sort the array
    myQsort<int>(t);
  }

  return 0;
}

int main( int argc, char** argv )
{
  //MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

9.2 Adaptative Quadrature: Integration by Newton-Cotes

The aim of this very simple divide and conquer strategy is to compute the integration of the function f on the interval $[a, b]$ according ot the Newton-Cotes method:

$$\int_a^b f dx = \int_a^{\frac{(a+b)}{2}} f dx + \int_{\frac{(a+b)}{2}}^b f dx$$

$$\forall |b - a| < h, g(a, b) = \int_a^b f dx$$

NC.C

```
#include <athapascan-1.h>

double g(double a, double b)
{
  return (a*a+b*b)*(b-a)/2.0;
}

struct sum {
  void operator()(Shared_r<double> a, Shared_r<double> b, Shared_w<double> c){
    c.write(new double(a.read() + b.read()));
  }
} ;

struct compute {
  void operator()(double a, double b, Shared_r<double> h, Shared_w<double> res){
    if(b-a < h.read()) {
      res.write(new double(g(a,b)));
    } else {
      Shared<double> res1(new double);
      Shared<double> res2(new double);
      res1.graph_name("res1"); res2.graph_name("res2");

      Fork<compute>() (a, (a+b)/2, h, res1);
      Fork<compute>() ((a+b)/2, b, h, res2);

      Fork<sum>() (res1, res2, res);
    }
  }
} ;

struct print_res {
  void operator() (Shared_r_w<double> res) {
    cout << "res = " << res.access() << endl;
  }
} ;

int doit( int argc, char** argv)
{
  double a, b, tmp;

  cout << "Give me a, b and the step h : ";
  cin >> a >> b >> tmp;

  Shared<double> res(new double);
  Shared<double> h(new double(tmp));
  res.graph_name("res"); h.graph_name("h");

  cout << "OK, start the computation..." << endl;
  Fork<compute>() (a, b, h, res);
  Fork<print_res>() (res);
  cout << "OK, that's done." << endl;

  return 0;
}

int main( int argc, char**argv )
{
  // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

Makefile

```
include ${A1_MAKEFILE}

all: NC

NC : NC.o

clean:
rm *.o NC
```

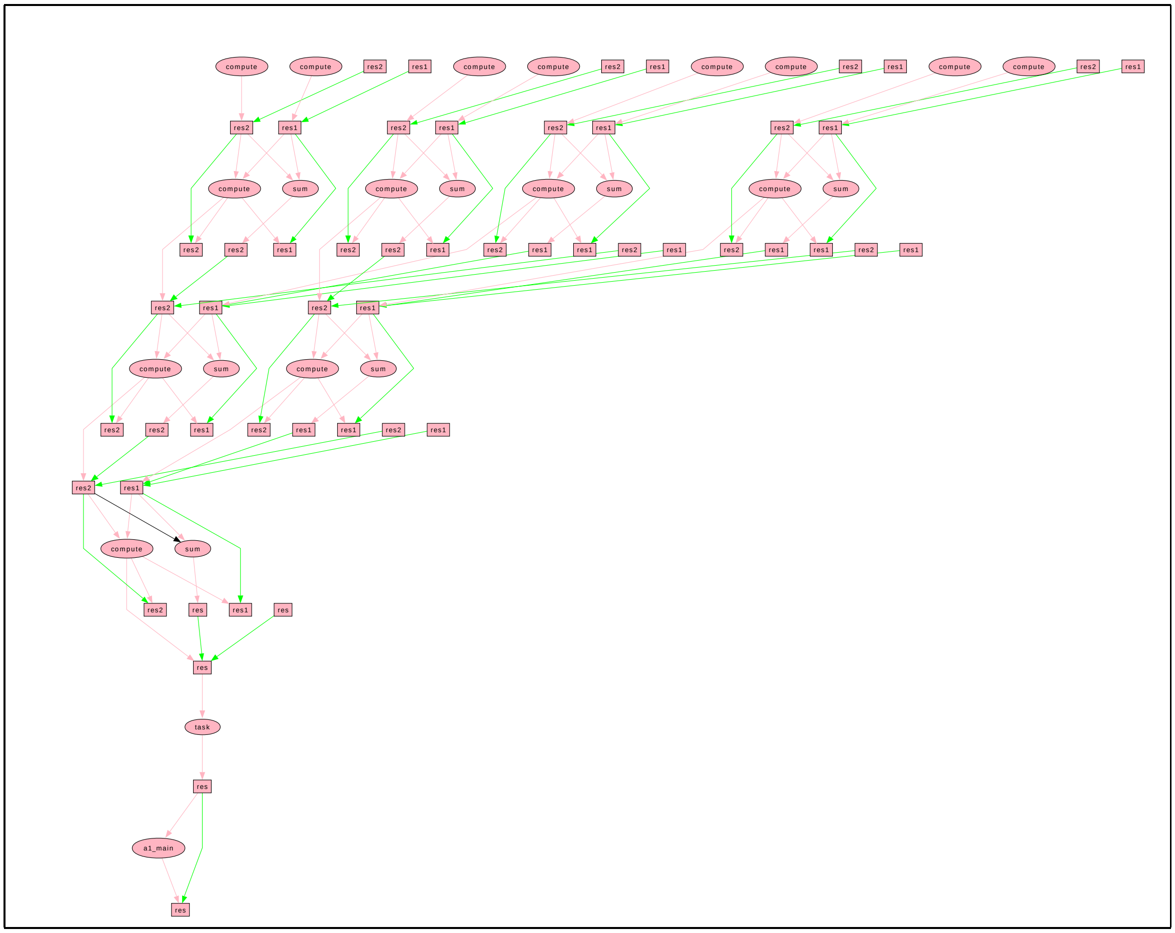

Figure 6: Execution graph corresponding to $a = 0$, $b = 1$ and $h = \frac{1}{8}$.

9.3 Scalar Product

This example shows the use of a cumulative shared and of an array of parameters. Parameter arrays are recursively split until their sizes are 1. Then the result is accumulated in a shared data.

Makefile

```
include ${A1_MAKEFILE}

all: pscal

pscal : pscal.o

clean:
rm *.o pscal
```

pscal.C

```
#include "athapascan-1.h"
#include <stdio.h>
#include <stdlib.h>

class add {
public:
   void operator()( int& a, const int b ) const {
      a += b;
   }
};

struct pscal {
public:
   const char* graph_name() const
   {
      return "pscal";
   }
   void operator()(  Param_array< Shared_r< int > > x,
                     Param_array< Shared_r< int > > y,
                    Shared_cw< add, int > res )
   {
      cout << "debut pscal" << endl ;
      cout << "on " << a1_system::self_node() << ":" << x.size() << endl;
      int n= x.size();
      if( n == 1 ) {
        res.cumul( x[0].read() * y[0].read() );
      } else {
         Param_array< Shared_r< int > > x1( n/2 );
         Param_array< Shared_r< int > > x2( n/2 + n%2 );
         Param_array< Shared_r< int > > y1( n/2 );
         Param_array< Shared_r< int > > y2( n/2 + n%2 );
         for(int i=0; i<n/2; i++) {
            x1[i] = x[i];
            x2[i] = x[n/2 + i];
            y1[i] = y[i];
            y2[i] = y[n/2 + i];
         }
         if( n%2 == 1 ) {
            x2[n/2]= x[ n-1 ];
            y2[n/2]= y[ n-1 ];
         }

         Fork<pscal>() (x1, y1, res );
         Fork<pscal>() (x2, y2, res );
      }
   }
};

struct verif {
public:
   const char* graph_name() const
   {
      return "verif";
   }
   void operator()( int val, Shared_r< int > x )
   {
      cout << "debut verif" << endl ;
      cout << "on " << a1_system::self_node() << ": Task verif execution: "
           << val << " ?=? " << x.read() << endl;
      if( val != x.read() )
         cout << "A1_TEST_ERROR: bad result: "
              << val << " != " << x.read() << endl;
   }
};

int doit( int argc, char** argv )
{
  cout << "in main" << endl;

  Shared< int > res( new int( 0 ) );
  res.graph_name( "res" );
  Param_array< Shared< int > > x( atoi( argv[1] ) );
  Param_array< Shared< int > > y( atoi( argv[1] ) );

  int val = 0;

  for( int i=0; i<x.size(); i++ ) {
    x[i] = Shared< int > ( new int( i ) );
    y[i] = Shared< int > ( new int( 2*i ) );
    char name[10];
    sprintf( name, "x[%d]", i );
    x[i].graph_name( name );
    printf( name, "y[%d]", i );
    y[i].graph_name( name );
    val += 2*i*i;
  }

  cerr << "avant fork" << endl ;
  Fork<pscal> ()(x, y,  res );
  cerr << "apres fork" << endl ;
  Fork<verif> ()(val, res );

  cout << "out main" << endl;

  return 0;
}

int main( int argc, char**argv )
{
  // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

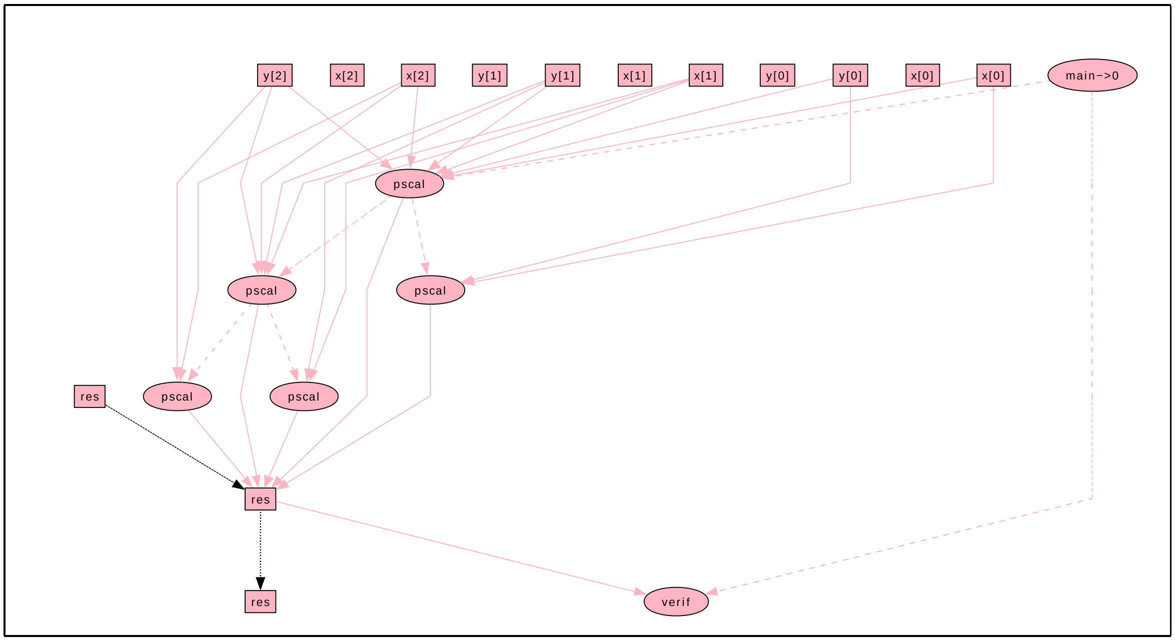

Figure 7: Execution graph corresponding to `pscal 3` execution.

9.4 Mandelbrot Set

This example intends to show the possible interaction between an Athapascan program and a X server. The following code results in a visualization of the Mandelbrot set on a X window. The algorithm is standard: the size of the image is split in four until a given threshold has been reached. The visualization is made during the computation, so that visualization threads have to execute on the X server site: a special scheduling policy is used for these threads.

Makefile

```
include ${A1_MAKEFILE}

X11_INCLUDES = /usr/openwin/include
X11_LIB = /usr/openwin/lib
CXXFLAGS+= -g -I$(X11_INCLUDES)
LDFLAGS+= -L$(X11_LIB) -lX11 -lm $(OBJ)

SRC= $(wildcard *.C)
OBJ= $(patsubst %.C,%.o,$(SRC))

all: mandel

mandel: $(OBJ)

clean:
rm -rf Templates.DB mandel mandel.o $(OBJ) tempinc/* core *.o
ptclean

cleanDB:
rm -rf Templates.DB mandel mandel.o


auto_test:
```

types.h

```
#ifndef TYPES_H
#define TYPES_H

#include <stream.h>
#include "athapascan-1.h"

class zone {
// (_xi,_yi): top left, (_xf,_yf): bottom right _xi<_xf _yf<_yi
public:
   zone();
   zone( double xi, double yi, double xf, double yf, int w, int h, int it,
         int thr, int pow );
   int empty() const;
   double scale_x() const;
   double scale_y() const;

   double _xi, _yi, _xf, _yf;
   int _w, _h, _it, _thr, _pow;

   friend ostream& operator<<( ostream& out, const zone& z );
   friend a1_ostream& operator<<( a1_ostream& out, const zone& z);
   friend a1_istream& operator>>( a1_istream& in, zone& z);
};


class complex {
public:
   complex();
   complex( double re, double im );

   double re() const;
   double im() const;
   double abs2() const;

   complex operator+( const complex& ) const;
   complex operator-( const complex& ) const;
   complex operator*( const complex& ) const;
   complex pow( int n ) const;

private:
   double _r;
   double _i;
};

#endif
```

win_mand.h

```
#ifndef WIN_MAND_H
#define WIN_MAND_H

#include "win_zoom.h"
#include "win_key.h"

class win_mand: public win_zoom, public win_key {
public:
   int init( zone z0, int caption, int nb_col );

   zone new_zone();
   void help();
protected:
   virtual int key_pressed( const char key );
   virtual int zoom( const region& r );

   virtual int x_resize( int width, int height );

   virtual XColor col2XColor( int c );

private:
int _quit;
   zone _z0;
   zone _old_zone;
   zone _new_zone;
};

#endif
```

win_proc_mand.h

```
 #ifndef WIN_PROC_MAND_H
#define WIN_PROC_MAND_H

#include "win_proc.h"
#include "types.h"

class win_proc_mand: public win_proc {
public:
   int init( zone z0, int caption, int nb_proc, int nb_threads );

protected:
   virtual int x_resize( int width, int height );
};

#endif
```

mandel.C

```
#include <stream.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include <athapascan-1.h>

#include "win_gest.h"
#include "win.h"
#include "win_mand.h"
#include "win_proc_mand.h"
#include "types.h"


static win_mand w_mand;
static win_proc_mand w_proc;

  class my_sch : public a1_mapping::group {
  public:
        int priority() { return 10; }
  };

int xy2color( double x, double y, int n, int it, int nb_col )
{
   int color= 0;
   complex z( 0, 0 ), c( x, y );

   for( int i = 0 ; i < it ; i++ ) {
      z= c + z.pow( n );

      if( z.abs2() >= 4 ) {
         color = 1+ (int)floor( double( i )/it * nb_col );
         break;
      }
   }
   return( color );
}

void compute_region( win::region r, zone z, Param_array< Shared<int> >& col, int nb_col )
{
   int i, j, k;

   for( j = r._y, k=0 ; j < r._y + r._h ; j++ )
      for( i = r._x ; i < r._x + r._w ; i++, k++ )
         col[k]= Shared<int> (new int(xy2color( z._xi + z.scale_x()*i, z._yi + z.scale_y()*j,
               z._pow, z._it, nb_col )));
}

struct display_region {
   const char* graph_name() const { return "display"; }

   void operator()( int node, int thread,
         win::region r, Param_array< Shared_r<int> > col )
   {
      int* _col= new int[ col.size() ];
      for( int i=0; i<col.size(); i++ )
         _col[i]= col[i].read();
      win_gest::Xenter();
      w_mand.draw( r, _col );
      w_proc.draw( r, node, thread );
      win_gest::Xleave();
      delete[] _col;
   }
};

struct mandel {
   const char* graph_name() const { return "mandel"; }
   mandel() {}

   void operator()( win::region r, zone z, int nb_col )
   {

      if( r._w <= z._thr ) {
         Param_array< Shared<int> > col( r._w * r._h );

         compute_region( r, z, col, nb_col );
         Fork<display_region>(SchedAttributes(-1,0,10,-1),a1_mapping::fixed())( a1_system::self_node(),
            a1_system::self_thread(), r, col );
      } else {
         int i, j;
         for( j = 0 ; j < 2 ; j++ )
            for( i = 0 ; i < 2 ; i++ ) {
               win::region rij( r._x + i*r._w/2,
                     r._y + j*r._h/2,
                     r._w/2 + i*(r._w%2),
                     r._h/2 + j*(r._h%2) );
               if( rij._w * rij._h > 0 )
                  Fork<mandel>()( rij, z, nb_col );
            }
      }
   }
};

int doit( int argc, char** argv )
{
  int i_size= atoi(argv[1]);
  int thr= atoi(argv[2]);
  int it= atoi(argv[3]);
  int nb_col= atoi(argv[4]);

  zone z0( -2.1, 2.1, 2.1, -2.1, i_size, i_size, it, thr, 2 );
  win_gest::init();

  int ok= w_mand.init( z0, 20, nb_col );
  ok|= w_proc.init( z0, 20, a1_system::node_count,
  a1_system::thread_count() );

  if( ok == 0 ) {
    cerr << "not enough colors..." << endl;
  } else {
      zone z= z0;

      w_proc.clean();
      w_mand.clean();
      w_mand.help();

      while( !z.empty() ) {
        cerr.precision( 15 );
        cerr << endl << "Mandelbrot: z=c+z^"<< z._pow << endl
             << "  zone            : " << z << endl
             << "  max iteration   : " << z._it << endl
             << "  image size      : " << z._w << " x " << z._h
             << endl
             << "  threshold       : " << z._thr << endl;

//               a1_work_steal::group my_grp;
//               a1_set_default_group(my_grp);

        win::region r( 0, 0, z._w, z._h );
        Fork<mandel>(SchedAttributes(-1,-1,-1,-1),a1_work_steal::basic())( r,  z, nb_col );

        win_gest::treatXEvents();
        z= w_mand.new_zone();
        w_proc.win::resize( w_mand._reg );
        w_proc.clean();
      }
    }
    win_gest::terminate();

  fflush(stdout);

  return 0;
}


int main( int argc, char**argv )
{
  // MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER

  return 0;
}
```

types.C

```
#include "types.h"

//------
// zone
//------
zone::zone()
   : _xi( 0 ), _yi( 0 ), _xf( 0 ), _yf( 0 ), _w( 0 ), _h( 0 ),
      _it( 0 ), _thr( 0 ), _pow( 0 )
{}

zone::zone( double xi, double yi, double xf, double yf, int w, int h, int it,
      int thr, int pow )
   : _xi( xi ), _yi( yi ), _xf( xf ), _yf( yf ), _w( w ), _h( h ), _it( it ),
      _thr( thr ), _pow( pow )
{}

int zone::empty() const
{
   return (_xf-_xi)*(_yf-_yi) == 0;
}

double zone::scale_x() const
{
   return (_xf-_xi)/(_w-1);
}

double zone::scale_y() const
{
   return (_yf-_yi)/(_h-1);
}

ostream& operator<<( ostream& out, const zone& z )
{
   out << "(" << z._xi << "," << z._yi << ")x(" << z._xf << "," << z._yf
       << ") ";
   return out;
}
a1_ostream& operator<<( a1_ostream& out, const zone& z)
{
   out << z._xi << z._yi << z._xf << z._yf << z._w << z._h << z._it << z._thr
       << z._pow;
   return out;
}
a1_istream& operator>>( a1_istream& in, zone& z)
{
   in >> z._xi >> z._yi >> z._xf >> z._yf >> z._w >> z._h >> z._it >> z._thr
      >> z._pow;
   return in;
}


//---------
// complex
//---------
complex::complex()
   : _r( 0 ), _i( 0 )
{}

complex::complex( double re, double im )
   : _r( re ), _i( im )
{}

double complex::re() const
{
   return _r;
}

double complex::im() const
{
   return _i;
}

double complex::abs2() const
{
   return( _r*_r + _i*_i );
}

complex complex::operator+( const complex& c ) const
{
   return complex( _r+c._r, _i+c._i);
}

complex complex::operator-( const complex& c ) const
{
   return complex( _r-c._r, _i-c._i);
}

complex complex::operator*( const complex& c ) const
{
   return complex( _r*c._r - _i*c._i, _r*c._i + _i*c._r);
}

complex complex::pow( int n ) const
{
   complex c( 1, 0 );
   for( int i=0; i<n; i++ )
      c= c*(*this);
   return c;
}
```

win_mand.C

```
#include <math.h>

#include "win_mand.h"

// -------------
// Constructors
// -------------
int win_mand::init( zone z0, int caption, int nb_col )
{
   _old_zone= z0;
   _z0 = z0;

   char title[100];
   sprintf( title, "Mandelbrot set (z=c+z^%d)", _old_zone._pow );
   region reg( 0, 0, z0._w, z0._h );

   int res;
   res= win_key::init( title, reg, caption, nb_col );
   res|= win_zoom::init( title, reg, caption, nb_col );

   return res;
}

// --------------
// Other: public
// --------------
void win_mand::help()
{
   cerr << endl
        << "Key binding in Mandelbrot window:" << endl
        << "   u: iterations + 10" << endl
        << "   U: iterations + 100" << endl
        << "   d: iterations - 10" << endl
        << "   D: iterations - 100" << endl
        << "   m: power of Mandelbrot - 1" << endl
        << "   M: power of Mandelbrot + 1" << endl
        << "   t: threshold - 10" << endl
        << "   T: threshold + 10" << endl
        << "   r: redraw picture" << endl
        << "   s: restart on default zone" << endl
        << "   +: redraw on five times smaller zone" << endl
        << "   -: redraw on five times bigger zone" << endl
        << "   h: print this help" << endl
        << "   q: quit" << endl
        << "Mouse binding in Mandelbrot window:" << endl
        << "   Button 1: define a zoomed rectangular zone" << endl
        << "   Button 2: define a zoomed square zone" << endl
        << "   Button 3: define a zoomed square zone centered" << endl
        << endl;
}

zone win_mand::new_zone()
{
   if( _quit==1 )
      return zone();

   _old_zone= _new_zone;
   _new_zone= zone();

   char title[100];
   sprintf( title, "Mandelbrot set (z=c+z^%d)", _old_zone._pow );
   set_title( title );

   return _old_zone;
}

static double pic( const double p, const double d, const double f, double x )
{
   x= x- floor( (x-p)/f )*f;
   if( ( x<p ) || ( x>p+d ) )
      return 0;
   else {
      x= (x-p-d/2)/(d/2);
      x= x>0 ? x : -x;
      double res= 1- exp(2*log(x));
      return res;
   }
}

// -------------------------
// Other: virtual protected
// -------------------------
XColor win_mand::col2XColor( int c )
{
   int max_col= 65535;
   double x= double(c)/_nb_col;

   XColor col;
   col.red=   (int) ceil( pic( 1.0/2, 2.0/3, 2.0, x )*max_col );
   col.green= (int) ceil( pic( 1.0/6, 2.0/3, 2.0, x )*max_col );
   col.blue=  (int) ceil( pic( -1.0/6, 2.0/3, 2.0, x )*max_col );

   return col;
}

int win_mand::key_pressed( const char key )
{
   int cont= 1;
   _quit= 0;

   switch( key ) {
   case ’h’: {
      help();
      break;
   }
   case ’q’: {
      _quit= 1;
      cont= 0;
      break;
   }
   case ’r’: {
      _new_zone= _old_zone;
      cont= 0;
      break;
   }
   case ’s’: {
      resize( region( 0, 0, _z0._w, _z0._h ) );
      clean();
      _new_zone= _z0;
      cont= 0;
      break;
   }
   case ’u’: case ’U’: case ’d’: case ’D’: {
      switch( key ) {
      case ’u’ : { _old_zone._it += 10; break; }
      case ’U’ : { _old_zone._it += 100; break; }
      case ’d’ : { _old_zone._it -= 10; break; }
      case ’D’ : { _old_zone._it -= 100; break; }
      }
      if (_old_zone._it < 0) _old_zone._it= 0;
      char title[100];
      sprintf( title, "Next draw iterations: %d", _old_zone._it );
      set_title( title );
      break;
   }
   case ’m’: case ’M’: {
      if( key==’m’ )
         _old_zone._pow -= 1;
      else
         _old_zone._pow += 1;
      if (_old_zone._pow < 1) _old_zone._pow= 1;
      char title[100];
      sprintf( title, "Next draw Mandelbrot: %d", _old_zone._pow );
      set_title( title );
      break;
   }
   case ’t’: case ’T’: {
      if( key==’t’ )
         _old_zone._thr -= 10;
      else
         _old_zone._thr += 10;
      int nb;
      if (_old_zone._thr < 1) {
         _old_zone._thr+= 10;
      }
      nb= (int) ceil( log( double(_reg._w)/_old_zone._thr ) / log( 2.0 ) );
      nb= (int) ceil( exp( nb * log( 2.0 ) ) );
      char title[100];
      sprintf( title, "Next draw threshold: %d => %d blocks",
            _old_zone._thr, nb*nb );
      set_title( title );
      break;
   }
   case ’+’: {
      region r( (2*_reg._w)/5, (2*_reg._h)/5, _reg._w/5, _reg._h/5 );
      _new_zone= _old_zone;
      _new_zone._xi= _old_zone._xi+ _old_zone.scale_x()*r._x;
      _new_zone._xf= _new_zone._xi+ _old_zone.scale_x()*r._w;
      _new_zone._yi= _old_zone._yi+ _old_zone.scale_y()*r._y;
      _new_zone._yf= _new_zone._yi+ _old_zone.scale_y()*r._h;
      cont= 0;
      Pixmap buff= XCreatePixmap( _dpy, _wroot, _reg._w, _reg._h, _depth );
      copy( _buff, r, buff, _reg );
      clean();
      XCopyArea( _dpy, buff, _buff, _gc, 0, 0, _reg._w, _reg._h, 0, 0 );
      XCopyArea( _dpy, _buff, _xwin, _gc, 0, 0, _reg._w, _reg._h+_caption, 0,    0);
      XFreePixmap( _dpy, buff );
      break;
   }
   case ’-’: {
      double dx= 2*(_old_zone._xf - _old_zone._xi);
      double dy= 2*(_old_zone._yf - _old_zone._yi);
      _new_zone= _old_zone;
      _new_zone._xi= _old_zone._xi- dx;
      _new_zone._xf= _old_zone._xf+ dx;
      _new_zone._yi= _old_zone._yi- dy;
      _new_zone._yf= _old_zone._yf+ dy;
      region r( 0, 0, _reg._w/5, _reg._h/5);
      Pixmap buff= XCreatePixmap( _dpy, _wroot, r._w, r._h, _depth );
      copy( _buff, _reg, buff, r );
      clean();
      XCopyArea( _dpy, buff, _buff, _gc, 0, 0, _reg._w, _reg._h,
            (2*_reg._w)/5, (2*_reg._h)/5 );
      XCopyArea( _dpy, _buff, _xwin, _gc, 0, 0, _reg._w, _reg._h+_caption, 0,    0);
      XFreePixmap( _dpy, buff );
      cont= 0;
      break;
   }
   }
   return cont;
}

int win_mand::zoom( const region& r )
{
   region new_reg= region( 0, 0, -1, _reg._h );
   new_reg._w= (int) ceil( double( r._w * new_reg._h ) / r._h );
   if( new_reg._w > _reg._w ) {
      new_reg._w= _reg._w;
      new_reg._h= (int) ceil( double( r._h * new_reg._w ) / r._w );
   }
   Pixmap buff= XCreatePixmap( _dpy, _wroot, new_reg._w, new_reg._h,
         _depth );
   copy( _buff, r, buff, new_reg );
   resize( new_reg );
   XCopyArea( _dpy, buff, _buff, _gc, 0, 0, _reg._w, _reg._h, 0, 0);
   XCopyArea( _dpy, _buff, _xwin, _gc, 0, 0, _reg._w, _reg._h+_caption, 0, 0);
   XFreePixmap( _dpy, buff );
   _new_zone= _old_zone;
   _new_zone._xi= _old_zone._xi + _old_zone.scale_x() * r._x;
   _new_zone._yi= _old_zone._yi + _old_zone.scale_y() * r._y;
   _new_zone._xf= _old_zone._xi + _old_zone.scale_x() * ( r._x + r._w-1 );
   _new_zone._yf= _old_zone._yi + _old_zone.scale_y() * ( r._y + r._h-1 );
   _new_zone._w= _reg._w;
   _new_zone._h= _reg._h;

   return 0;
}

static int min( int a, int b )
{
   return (a<b) ? a : b;
}

int win_mand::x_resize( int width, int height )
{
   region old_reg= _reg;

   Pixmap buff= XCreatePixmap( _dpy, _wroot, old_reg._w, old_reg._h, _depth );
   XCopyArea( _dpy, _buff, buff, _gc, 0, 0, old_reg._w, old_reg._h, 0, 0);
   setup_buffers( region( 0, 0, width, height-_caption ) );
   clean();
   XCopyArea( _dpy, buff, _buff, _gc, 0, 0, min( old_reg._w, _reg._w ),
         min( old_reg._h, _reg._h ), 0, 0);
   XCopyArea( _dpy, _buff, _xwin, _gc, 0, 0, _reg._w, _reg._h+_caption, 0, 0);
   XFreePixmap( _dpy, buff );

   _new_zone= _old_zone;
   _new_zone._xf= _old_zone._xi + _old_zone.scale_x() * ( _reg._w-1 );
   _new_zone._yf= _old_zone._yi + _old_zone.scale_y() * ( _reg._h-1 );
   _new_zone._w= _reg._w;
   _new_zone._h= _reg._h;


   return 0;
}
```

win_proc_mand.C

```
#include "win_proc_mand.h"

// -------------
// Constructors
// -------------
int win_proc_mand::init( zone z0, int caption, int nb_proc, int nb_threads )
{
   return win_proc::init( region( 0, 0, z0._w, z0._h ),
         caption, nb_proc, nb_threads );
}

// -------------------------
// Other: virtual protected
// -------------------------
int win_proc_mand::x_resize( int width, int height )
{
   resize( _reg );
   return 1;
}
```

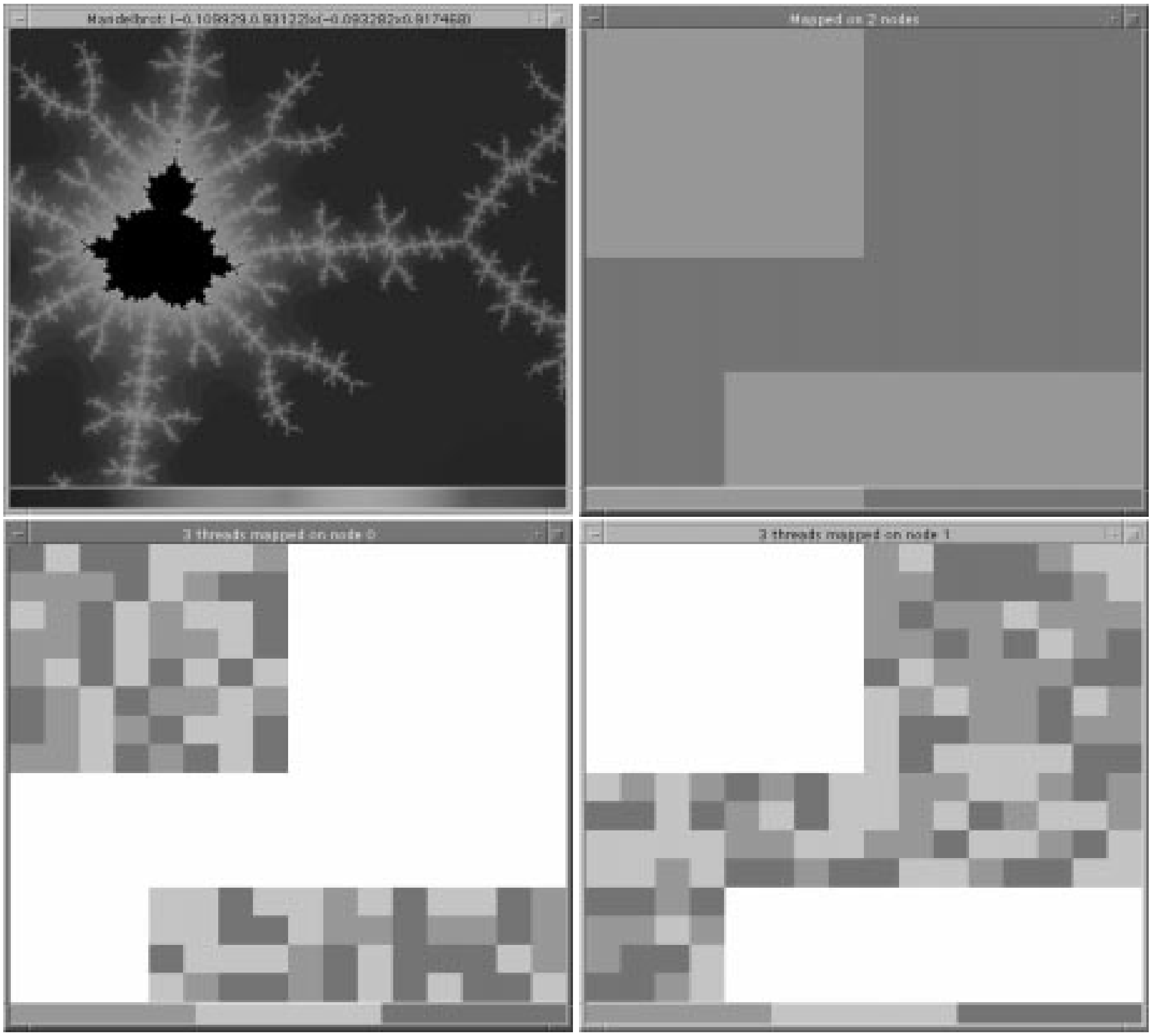
Mandelbrot: [-0.109929,0.931372]x[-0.093282x0.917468]
Mapped on 2 nodes
3 threads mapped on node 0
3 threads mapped on node 1

Figure 8: Mandelbrot Set visualization: main and mapping windows. The execution was on two nodes each having three virtual processors.

9.5 Matrix Multiplication

This example shows the use of ATHAPASCAN for implementing a parallel application on matrix operations. Matrix product and addition are implemented by classical bi-dimensional block parallel algorithms.

Makefile

```
include ${A1_MAKEFILE}

all: matrix

matrix : matrix.o

clean:
rm *.o Matrix
```

matrix.C

```
#include <athapascan-1.h>

/* A basic two dimensional matrix type */
struct LocalMat{
  double tab[50][50] ;

  LocalMat() {} ;
  LocalMat(const LocalMat& A) {
    for (int i = 0 ; i < 50 ; i++ )
      for (int j = 0 ; j < 50 ; j++ )
        tab[i][j] = A.tab[i][j] ;
  } ;

} ;



a1_ostream& operator<<(a1_ostream& ostr, const LocalMat& A ) {
   for (int i = 0 ; i < 50 ; i++ )
     for (int j = 0 ; j < 50 ; j++ )
       ostr << A.tab[i][j] ;
   return ostr ;
}

a1_istream& operator>>(a1_istream& istr, LocalMat& A) {
   for (int i = 0 ; i < 50 ; i++ )
     for (int j = 0 ; j < 50 ; j++ )
       istr >> A.tab[i][j] ;
   return istr ;
}


/* Sequential Computation of C = A + B */
struct seqmatrixadd {
  void operator()(Shared_r<LocalMat> A, Shared_r<LocalMat> B, Shared_w<LocalMat> C){
    LocalMat Temp ;
    const LocalMat& my_A = A.read() ;
    const LocalMat& my_B = B.read() ;

    for (int i = 0 ; i < 50 ; i++)
      for (int j = 0 ; j < 50 ; j++)
        Temp.tab[i][j] = my_A.tab[i][j] + my_B.tab[i][j] ;

    C.write(new LocalMat(Temp)) ;
  }
} ;

/* Sequential Computation of C = A * B */
struct seqmatrixmultiply {
  void operator()(Shared_r<LocalMat> A, Shared_r<LocalMat> B, Shared_w<LocalMat> C){
    LocalMat Temp ;
    const LocalMat& my_A = A.read() ;
    const LocalMat& my_B = B.read() ;

    for (int i = 0 ; i < 50 ; i++)
      for (int j = 0 ; j < 50 ; j++)
        {
Temp.tab[i][j] = 0 ;
  for (int k = 0 ; k < 50 ; k++)
    Temp.tab[i][j] += my_A.tab[i][k] * my_B.tab[k][j] ;
      }

    C.write(new LocalMat(Temp)) ;
  }
} ;


/* Parallel Computation of a matrix product: R = R + A*A */
void ParSquareMat( Shared< LocalMat >** R,
                       Shared< LocalMat >** A, int dim ) {

for( int i = 0 ; i < dim ; i ++ )
  for( int j = 0 ; j < dim ; j++ )
    for( int k = 0 ; k < dim ; k++ ) {
      Shared< LocalMat > tmp(new LocalMat());
      Fork<seqmatrixmultiply> () ( A[i][k], A[k][j], tmp );
      Fork<seqmatrixadd> ()( R[i][j], tmp, R[i][j] );
    }
};

//a1_work_steal::basic()

// The main function
int doit( int argc, char** argv)
{
  Shared< LocalMat >* A[10];
  Shared< LocalMat >* R[10];

  for ( int i = 0 ; i < 10 ; i++){
    A[i] = new Shared< LocalMat >[10];
    R[i] = new Shared< LocalMat >[10];
    for ( int j = 0 ; j < 10 ; j++)
    {
      A[i][j] = Shared< LocalMat >(new LocalMat()) ;
      R[i][j] = Shared< LocalMat >(new LocalMat()) ;
    }
  }
  ParSquareMat (R, A, 10) ;

  return 0;
}


int main( int argc, char**argv )
{
```

```
// MAIN METHOD AS PREVIOUSLY DEFINED IN API CHAPTER
```

```
  return 0;
}
```

10 Culminating Example: lifegame.cpp

The lifegame program was developed to provide a visualization of the asynchronous task execution performed by Athapascan. This program serves as an example for most of the concepts covered in this manual: passing and declariation of Shared data, Forking user-defined structures, the communicability of user-defined classes, and the internal scheduling of tasks by Athapascan.

The program as a whole can be divided into two parts: the simulation (lifegame.cpp, Message.cpp, Message.h) and the visualization (SappeJuggler.cpp, SappeJuggler.h, NJSocket.cpp NJSocket.h, GOLApp.cpp, GOLApp.h). The simulation, in particular lifegame.cpp, uses Athapascan to parallelize the code. The Message class defined by the other two files sends the information needed for the visualization through the sockets it creates. The visualization portion of this project contains no parallel code, and is only used for recieving the messages sent by Message.cpp and generating a graphic output with OpenGL from the information received.

Lifegame.cpp creates a matrix of cells caracterized by a boolean state, an integer time, and two interger coordinates: x, y (as defined in the cell_state class) Given the state of the current cell and the current state of the cells surrounding it, the program calculates a new state for the current cell, updates the time variable, and sends this inforlation as a message through a socket to visualization. The visualization recieves this message and displays the matrix of cells. Each cell is displayed with a color corresponding to the time at which the cell was updated and the information sent.

When running this program in Athapascan's different modes (except sequential) the asynchronous task execution is clearly displayed by the color variance from cell to cell.

cell_state.cpp

```
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#ifndef _CELL_STATE
#define _CELL_STATE
const bool alive = true ;
const bool dead = false ;


class cell_state {
  public:
    bool state;
    int time;
    int my_x;
    int my_y;

    cell_state(){if(rand() > (RAND_MAX/2)) state = alive; else state = dead; time=0;my_x=0;my_y=0;}
    ~cell_state(){}
};

#endif
```

lifegame.cpp

```
#include <athapascan-1>
#include <iostream>
#include <math.h>
#include "cell_state.cpp"
#define NUM_SOCKET_VRJUGGLER 60125

namespace AC = ACOM_NET_NAMESPACE;


// Simple life game
//

class cell;
class force;
a1::OStream& operator << (a1::OStream& ostr, const force& A ) ;
a1::IStream& operator >> (a1::IStream& istr, force& A) ;
a1::OStream& operator << (a1::OStream& ostr, const cell& A ) ;
a1::IStream& operator >> (a1::IStream& istr, cell& A) ;


//////////////////////////////////////// BEGIN FORCE STRUCT ///////////////////////////////////////////

class force {
  // The intensity of the force brought by a ceil c has intensity 1 is c is alive, 0 else.
  // The intensity of the force brought by a set of ceil is the integration (sum) of
  // the intesities of the forces of each cell;

public:
  int intensity ;

  force() : intensity(0) {} ;

  force( const cell contributor ) ;

  void reset() { intensity=0 ; } ; // Annulation of the force

  struct integration_law {
    // To provide a cumulative function class that performs forces integration
   void operator() ( force& res, force a ) { res.intensity += a.intensity; } ;
  };

  // Encapsulation of integration into a task
  struct integration_task {
    void operator() ( a1::Shared_cw<integration_law, force > f, a1::Shared_r< cell > c ) ;
  } ;
} ;

//////////////////////////////////////// END FORCE STRUCT ///////////////////////////////////////////


//////////////////////////////////////// BEGIN CELL STRUCT ///////////////////////////////////////////
//const bool alive = true ;
//const bool dead = false ;

class cell {
  public:
// ORIGINAL   char state ;
    cell_state info;

// ORIGINAL   cell ( char init_state = dead) :  state(init_state) {} ;
    inline cell (){}
    inline cell(int x, int y){info.my_x=x; info.my_y=y;}

    bool is_alive() const { return info.state == alive ; } ;

    void evolution( const force i ) {
      if ((info.state == dead) && (i.intensity >= 3)) { info.state = alive ; }
      else if  ((info.state == alive)  && (
 (i.intensity >= 4) || (i.intensity <=1) )) { info.state = dead ; } ;
    } ;

    // Encapsulation of evolution into a function class for Athapascan

    struct evolution_cell {
      void operator()( a1::Shared_r_w< cell > this_cell,  a1::Shared_r_w< force > f, int current_time, int x, int
    };

    struct evolution_task {
      void operator()( a1::Shared_r_w< cell > this_cell, a1::Shared_r_w< force > f );
    };
};

////////////////////////// END CELL HEAD //////////////////////////////


void cell::evolution_task::operator()( a1::Shared_r_w< cell > this_cell, a1::Shared_r_w< force > f
{
        this_cell.access().evolution( f.access() ) ;
f.access().reset() ;
}
```

```
void cell::evolution_cell::operator()( a1::Shared_r_w< cell > this_cell,  a1::Shared_r_w< force > f, int current_time, int x, int y)
{
        this_cell.access().info.time = current_time;
this_cell.access().info.my_x = x;
this_cell.access().info.my_y = y;
this_cell.access().evolution( f.access() );
f.access().reset() ;
}

force::force( const cell contributor ) {
    intensity = (contributor.is_alive())  ? 1 : 0 ;
  } ;

void force::integration_task::operator()
     ( a1::Shared_cw<integration_law, force > f, a1::Shared_r< cell > c )
{
      f.cumul( force( c.read() ) ) ;
} ;

//////////////////////////////////////////// IO STREAMS FOR ATHAPASCAN ///////////////////////////

#ifndef _FORCE_IO
#define  _FORCE_IO
a1::OStream& operator << (a1::OStream& ostr, const force& A ) {
   ostr << A.intensity ;
   return ostr ;

}

a1::IStream& operator >> (a1::IStream& istr, force& A) {
   istr >> A.intensity ;
   return istr ;
}
#endif

#ifndef _CELL_IO
#define _CELL_IO
a1::OStream& operator << (a1::OStream& ostr, const cell& A ) {
   ostr << A.info.state << A.info.time << A.info.my_x << A.info.my_y ;
   return ostr ;
}

a1::IStream& operator >> (a1::IStream& istr, cell& A) {
   istr >> A.info.state >> A.info.time >> A.info.my_x >> A.info.my_y ;
   return istr ;
}
#endif

//////////////////////////////////////////////// END IO STREAMS FOR ATHAPASCAAN /////////////////////////


// For visualization only
#include "Message.h"

Message* output ;
/*
struct output_to_buffer {
// ORIGINAL  void operator() ( a1::Shared_r_w<vector< char > > b, a1::Shared_r<cell> x, int k) {
  void operator() ( a1::Shared_r_w<vector< int > > b, a1::Shared_r<cell> x, int k) {
    // std::cout << "-- etat-indice: " << x.read().state << "-" << k << std::endl ;
```

```
    b.access()[k] = x.read().state ;
  }
};
*/
struct OutputInit {
//  void operator() ( int nx, int ny, a1::Shared_r_w<vector< char > > buffer) {
  void operator() ( int nx, int ny) {
    std::cout << "OutputInit..." << std::endl ;
    output = new Message ;
    output->SocketEcoute( NUM_SOCKET_VRJUGGLER ) ;
    output->EnvoiMsgInit( nx, ny ) ;
    std::cout << "...OutputInit" << std::endl ;
  } ;
} ;

struct  OuptutPrintCell {
  void operator()(a1::Shared_r_w<cell> this_cell){
//    std::cout<<"OuptutPrintCell .... "<<std::endl;
    output->Envoyer(&this_cell.access().info, sizeof(cell_state));
    cell_state c[1];
    output->Recevoir(c, sizeof(cell_state));
//    std::cout<<" .... OuptutPrintCell"<<std::endl;
  };
};
/* Main of the program
*/

int doit(a1::Community com, int argc, char** argv)
{
  int ny  = atoi(argv[1]);
  int nx = atoi(argv[2]);
  std::cout << "GameLife with ny=" << ny << "  nx=" << nx << std::endl;
  int display_frequency = atoi(argv[3]);
  if (display_frequency < 1) display_frequency=1 ;
  int global_synchronization_frequency = atoi(argv[4]);
  if (global_synchronization_frequency < 1) global_synchronization_frequency=1 ;


  // Initial state of the cells
  a1::Shared<cell>** board = new a1::Shared<cell>* [ny];
  a1::Shared<force>** forces = new a1::Shared<force>* [ny];
  for (int i=0; i<ny; ++i) {
    board[i] = new a1::Shared<cell> [nx];
    forces[i] = new a1::Shared<force> [nx];
    for (int j=0; j<nx; ++j) {
      board[i][j] = a1::Shared<cell>(cell(i,j));
// ORIGINAL     board[i][j] = a1::Shared<cell>( (rand() > (RAND_MAX/2)) ? alive : dead ) ;
      forces[i][j] = a1::Shared<force>( force() ) ;
    };
  } ;

  a1::Fork< OutputInit > ( a1::SetSite(0) ) (nx, ny) ;

  std::cout << "Avant sync..." << std::endl ;
    com.sync() ;
  std::cout << "Apres sync..." << std::endl ;

  for (int t=0 ; true ; t++ ) { // Time loop
  //std::cout << "Avant sync..." << std::endl ;
  //com.sync() ;
  //std::cout << "Apres sync..." << std::endl ;

    // 1. Computation of all the forces for the board
    std::cout << "Time=" << t << std::endl ;
    for (int i=0; i<ny; ++i)
      for (int j=0; j<nx; ++j) {
        // Computation of the force from each neighbour of the cell board[i][j]
        // Loop on the neighbours of each cell
if (i!=0)
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i-1][j] ) ;
if ((i!=0) && (j!=0) )
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i-1][j-1] ) ;
if (j!=0)
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i][j-1] ) ;
if ((i!=ny-1) && (j!=0))
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i+1][j-1] ) ;
if (i!=ny-1)
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i+1][j] ) ;
if ((i!=ny-1)&&(j!=nx-1))
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i+1][j+1] ) ;
if (j!=nx-1)
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i][j+1] ) ;
if ((i!=0) && (j!=nx-1))
a1::Fork<force::integration_task >(OCR(board[i][j]))(forces[i][j], board[i-1][j+1] ) ;
      } ;

    // 2. Application of the force to each cell
// Application of the force
    for (int i=0; i<ny; ++i)
{
      for (int j=0; j<nx; ++j)
{
        a1::Fork< cell::evolution_cell >(OCR(board[i][j])) ( board[i][j], forces[i][j],t, i, j);
if (t % display_frequency == 0) a1::Fork< OuputPrintCell >() ( board[i][j] ) ;

}
}



    if (t % global_synchronization_frequency == 0) {
     std::cout << "Before sync... t=" << t << std::endl ;
     com.sync() ;
     std::cout << "After  sync... t=" << t << std::endl ;
    }

  } ;


  return 0;
}




/* main entry point : Athapascan initialization
*/
int main(int argc, char** argv)
{
  try {
    a1::Community com = a1::System::create_initial_community( argc, argv );
    std::cout << "count argc" << argc << std::endl;
    if (argc != 5)  {
      for (int i=0; i<argc; ++i)
        std::cout << argv[i] << std::endl;
    std::cerr << "Usage: " << argv[0] << " nb_lines nb_cols display_frequency global_synchronization_frquency"
    << std::endl ;
      return 0;
    }

    doit( com, argc, argv );
    com.leave();
  }
  catch (const a1::InvalidArgument& E) {
    std::cout << "Catch invalid arg" << std::endl;
  }
  catch (const a1::BadAlloc& E) {
    std::cout << "Catch bad alloc" << std::endl;
  }
  catch (const a1::Exception& E) {
    std::cout << "Catch : "; E.print(std::cout); std::cout << std::endl;
  }
  catch (...) {
    std::cout << "Catch unknown exception: " << std::endl;
  }

  return 0;
}
```

Message.cpp

```
// ==========================================================================
// (c) projet SAPPE
// Author : F. Zara
//
// ==========================================================================

/** \file Message.C
    Definition des messages envoyes entre le programme lut et le programme Athapascan-1.
*/

#include <stdio.h>
#include <iostream>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/uio.h>

#include <netinet/in.h>
#include <netdb.h>

#include <strings.h>

#include <math.h>
```

```
#include <stdlib.h>

#include "Message.h"


using namespace std;

/**
 * Fonction permettant d'eliminer les processus de service quand ils se terminent :
 il suffit que le serveur ignore le signal SIGCLD.
*/
void Message::sigchld()
{
  /** wait3(p_status, options, p_usage) **/
  while(wait3(NULL, WNOHANG, NULL) > 0 );
}


/**
 * Fonction de creation d'une socket : le parametre est le numero du port souhaite,
 le numero de port sera envoye en resultat.
*/
int Message::CreerSock(int *port, int type)
{
  /* Adresse de la socket */
  struct sockaddr_in nom;

  /* Longueur de la socket */
  unsigned int longueur;


  /** Creation de la socket : socket(domaine, type, protocole) **/
  if ((_desc = socket(AF_INET, type, 0)) == -1)
    {
      perror("Creation de la socket impossible");
      exit(2);
    }

  /** Initialisation a 0 de la zone memoire d'adresse et de taille donnees **/
  bzero((char *)&nom, sizeof(nom));
//  bzero((char *)&nom, sizeof(nom));


  /** Numero du port **/
  nom.sin_port = htons(*port);

  /** Adresse Internet **/
  nom.sin_addr.s_addr = INADDR_ANY;

  /** Famille de l'adresse : AF_INET **/
  nom.sin_family = AF_INET;

  /** Nommage de la socket : bind(sock, p_adresse, lg) **/
  if (bind(_desc, (struct sockaddr *)&nom, sizeof(nom))!=0)
    {
      perror( "Nomage de socket impossible");
      exit(3);
    }

  /** Longueur de la socket **/
  longueur = sizeof(nom);

  /** Recuperation de l'adresse : getsockname(sock, p_adr, p_lg) **/
  if (getsockname(_desc, (struct sockaddr *)&nom, &longueur)!=0)
    {
      perror("Obtention du nom de la socket impossible");
      exit(4);
    }

  /** Passage de la representation reseau d'un entier a sa representation locale **/
  *port = ntohs(nom.sin_port);

  /** Renvoi du descripteur **/
  return _desc;
}


/**
 * Creation de la socket d ecoute.
 */
void Message::SocketEcoute(int port)
{
  /* Adresse de la socket */
  struct sockaddr_in adr;

  /* Taille de l adresse de la socket */
  unsigned int lgadr = sizeof(adr);


  /** Creation de la socket d'ecoute : creersock(int *port, type) **/
  /** et attachement au port du service d'une socket **/
  if ((_sock_ecoute = CreerSock(&port, SOCK_STREAM)) == -1)
    {
      fprintf(stderr, "Creation/liaison de socket impossible \n");
      exit(2);
    }


  /** Creation de la file de connexions pendantes **/
  /** Signale au systeme que le serveur accepte les demandes de connexion : **/
  /** listen(sock, nb) avec nb le nombre max de connexion pendantes **/
  listen(_sock_ecoute, 10);


  /** Taille de l adresse de la socket **/
  lgadr = sizeof(adr);


  /** Extraction d'une connexion pendante dans la file associee a la socket : **/
  /** accept(sock, p_adr, p_lgadr) **/
  _sock_service = accept(_sock_ecoute, (struct sockaddr *)&adr, &lgadr);

  _desc = _sock_service;
  /** Fermeture de la socket d'ecoute **/
  close(_sock_ecoute);
}


/**
 * Fonction recevoir : lit sur la socket fd size octets
 et retourne 0 si coupure de la connexion ou size.
*/
int Message::Recevoir(cell_state *buf, int size) //BAK (int *buf ... // ORIGINAL (char *buf, int size)
{
  int fd = _desc;
  cell_state *buf_inter;
  int NbRead = 0;
  int NbToRead = size;

  buf_inter = buf;

  /* Boucle de reception */
  while(NbToRead != 0)
    {
      buf_inter += NbRead;
      NbRead = recv(fd, buf_inter, NbToRead, 0);
      NbToRead -= NbRead;

      if(NbRead <= 0)
break;
    }//while

  /* Verification que tout a ete recu */
  if (NbRead <= 0 && NbToRead !=0)
    {
      // La connexion est rompue
      return 0;
    }
  else
    // Envoi de la taille des donnees
    return size;
}


/**
 * Fonction envoyer : envoi sur la socket fd size octets
 et retourne 0 si coupure de la connexion ou size.
*/
int Message::Envoyer(cell_state *buf, int size)// BAK (int *buf ... // ORIGINAL (char *buf, int size)
{
  int fd=_desc;
  cell_state *buf_inter;
  int NbSent = 0;
  int NbToSend = size;

  buf_inter = buf;
  /* Boucle d envoi */
  while(NbToSend != 0)
    {
      buf_inter += NbSent;
      NbSent = send(fd, buf_inter, NbToSend, 0);
      NbToSend -= NbSent;

      if(NbSent <= 0)
break;
    }//while

  /* Verification que tout a ete envoye */
  if (NbSent <= 0 && NbToSend !=0)
    {
      // La connexion est rompue
      return 0;
    }
  else
    // Envoi de la taille des donnees
    return size;
}


/**
 * Envoi des caracteristiques des blocs au programme Glut.
 */
void Message::EnvoiMsgInit(int nx, int ny)
{
  /* Message envoye au programme Glut (taille_bloc) */
  cell_state MsgGlut;

  /** Preparation du message pour le programme de visualisation **/
  MsgGlut.my_x = nx;
  MsgGlut.my_y = ny;
  MsgGlut.time = 0;
  MsgGlut.state = 0;

  /** Envoi du message au programme de visualisation **/
    if (Envoyer((cell_state*)&MsgGlut, sizeof(cell_state)) == 0)
// ORIGINAL  if (Envoyer((char*)&MsgGlut, sizeof(MsgGlut)) == 0)
    {
      fprintf(stderr, "Envoi des parametres incorrect\n");
      exit(2);
    }
}
```

Message.cpp

```
// ========================================================================
// (c) projet SAPPE
// Author : F. Zara
//
// ========================================================================

/** \file Message.h
    Definition des messages envoyes entre le programme lut et le programme Athapascan-1.
*/

#ifndef _MESSAGE_H
#define _MESSAGE_H

/** Librairies de base **/
#include <strings.h>
#include <stdio.h>


/** Pour l emploi des sockets **/
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
```

```
#include <sys/ioctl.h>
#include <sys/types.h>
#include <pthread.h>
#include "cell_state.cpp"
/**
 * Message envoye par le programme A1 vers le programme de visualisation.
 */
typedef struct
{
  int nx,ny;
} msg_init;


/**
 * Structure de donnees relatives aux messages transmis entre le programme A1 et de visualisation.
 */
class Message
{
 public:

  /*! Constructeur vide */
  inline Message() {}

  /*! Fonction permettant d'eliminer les processus de service quand ils se terminent :
    il suffit que le serveur ignore le signal SIGCLD */
  void sigchld();

  /*! Fonction de creation d'une socket :
    le parametre est le numero du port souhaite,
    le numero de port sera envoye en resultat */
  int CreerSock(int *port, int type);

  /*! Creation de la socket d ecoute */
  void SocketEcoute(int port);

  /*! Fonction recevoir : lit sur la socket fd size octets
    et retourne 0 si coupure de la connexion ou size */
/*  int Recevoir(char *buf, int size); */
/* int Recevoir(int *buf, int size); */
int Recevoir(cell_state *buf, int size);

  /*! Fonction envoyer : envoi sur la socket fd size octets
    et retourne 0 si coupure de la connexion ou size */
/*  int Envoyer(char *buf, int size);*/
/* int Envoyer(int *buf, int size); */
int Envoyer(cell_state *buf, int size);

  /*! Envoi des caracteristiques des blocs du programme A1 */
  void EnvoiMsgInit(int nx, int ny);

  /*! Destructeur de la class Message */
  inline ~Message() {}

 private:

  // Descripteur de la socket creee
  int _desc;

  // Connexion pendante associee a la socket
  int _sock_service;
```

```
  // Socket d ecoute
  int _sock_ecoute;

};



#endif
```

SappJuggler.cpp

NJSocket.cpp

```
// ==============================================================================
// (c) projet SAPPE
// Author : F. Zara
// modified by: Joe Hecker for use in lifgame
// ==============================================================================

/** \file Message.C
    Definition des messages envoyes entre le programme lut et le programme Athapascan-1.
*/

#define NOMPI

#include <stdio.h>
#include <unistd.h>
#include <iostream>

#include <sys/socket.h>

#include <netinet/in.h>
#include <netdb.h>

#include <strings.h>
#include <string.h>
#include <math.h>

#ifndef NOMPI
#include "mpi.h"
#endif

#include "NJSocket.h"

NJSocket::NJSocket()
{
  masterNode = 0;
  sock = 0;
}

int NJSocket::connect(char* serveur, int port)
{
  printf("%c-connect(%s,%d)\n",isMaster()?'M':'S',serveur,port);
  int code =  0;
  if (isMaster()) {
    /* Adresse de la socket */
    struct sockaddr_in nom;
    /* Adresse internet du serveur */
    struct hostent *hp;
    /** Creation de la socket : socket(domaine, type, protocole) **/
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
      fprintf(stderr,"Creation de la socket impossible");
      code=-2;
    }
    /** Recherche de l'adresse internet du serveur **/
    else if ((hp = gethostbyname(serveur)) == NULL)
    {
      fprintf(stderr,"%s : site inconnu \n", serveur);
      code=-3;
    }
    else {
      /** Preparation de l'adresse de la socket destinataire **/
      bcopy(hp->h_addr, &nom.sin_addr, hp->h_length);
      nom.sin_family = AF_INET;
      nom.sin_port = htons(port);

      /** Demande de connexion : connect(sock, p_adr, lgadr) **/
      if (::connect(sock, (struct sockaddr *)&nom, sizeof(nom)) == -1)
      {
fprintf(stderr,"Erreur dans la connexion");
code=-4;
      }
    }
  }
  else { // by default the sock is valid
    sock=1;
  }
  printf("%c-connect(%s,%d):BCAST\n",isMaster()?'M':'S',serveur,port);
#ifndef NOMPI
  if (MPI_Bcast(&code, 1, MPI_INT, masterNode, MPI_COMM_WORLD)!=MPI_SUCCESS)
    return -1;
#endif
  if (code<0) sock=0; // invalid socket
  printf("%c-connect(%s,%d):END%d\n",isMaster()?'M':'S',serveur,port,code);
  return code;
}


int NJSocket::recv(cell_state* buf, int size)  //(char *buf, int size)
{
  if (sock==0) return -1; // invalid socket
  if (isMaster()) {
    cell_state* buf_inter;
    int NbRead = 0;
    int NbToRead = size;

    buf_inter = buf;

    /* Boucle de reception */
    while(NbToRead != 0)
    {
      buf_inter += NbRead;
      NbRead = ::recv(sock, buf_inter, NbToRead, 0);
      NbToRead -= NbRead;

      if(NbRead <= 0)
break;
    }//while
```

```
    /* Verification que tout a ete recu
    if (NbRead <= 0 && NbToRead !=0)
    {
      // La connexion est rompue
      return -2;
    }
    */
  }

#ifndef NOMPI
  MPI_Bcast(buf, size, MPI_BYTE, masterNode, MPI_COMM_WORLD);
#endif

  return 0;

}


/**
 * Fonction envoyer : envoi sur la socket fd size octets
 et retourne 0 si coupure de la connexion ou size.
*/
int NJSocket::send(cell_state *buf, int size) //(char *buf, int size)
{
  if (sock==0) return -1; // invalid socket
  if (isMaster()) {
    cell_state* buf_inter;
    int NbSent = 0;
    int NbToSend = size;

    buf_inter = buf;

    /* Boucle d envoi */
    while(NbToSend != 0)
    {
      buf_inter += NbSent;
      NbSent = ::send(sock, buf_inter, NbToSend, 0);
      NbToSend -= NbSent;

      if(NbSent <= 0)
break;
    }//while
  }
  return 0;
}

int NJSocket::close()
{
  if (sock==0) return -1; // invalid socket
  if (isMaster()) {
    ::close(sock);
  }
  sock=0;
  return 0;
}

bool NJSocket::isMaster()
{
#ifndef NOMPI
  int node;
  MPI_Comm_rank(MPI_COMM_WORLD,&node);
  return node==masterNode;
#else
  return 0==0;
#endif
}
```

NJSocket.h

```
#ifndef NJSOCKET_H
#define NJSOCKET_H

#include <stdio.h>
#include <iostream>
#include <strings.h>
#include <math.h>
#include <stdlib.h>
//#include "GOLApp.h"
#include "cell_state.cpp"

/* Classe permettant de recevoir des donnees d'un serveur dans un programme netjuggler
 * S'utilise comme une socket TCP. Sur le noeud maitre c'est effectivement une socket mais les noeuds esclaves recoivent les donnees du maitre par broadcast
 * Note: les fonctions renvoyent 0 si succes, code d'erreur negatif sinon.
 */
/*
typedef struct cell_state {
  char state;
  int time;
  int my_x;
  int my_y;
};
*/
class NJSocket {
 public:

  NJSocket();

  bool isMaster();

  /* connecte au serveur */
  int connect(char *serveur, int port);

  /* envois des donnees au serveur. Note: ceci n'a d'effet que sur le maitre */
/*  int send(char *buffer, int size);*/
  int send(cell_state *buffer, int size);

  /* recois des donnees */
/*  int recv(char *buffer, int size);*/
  int recv(cell_state *buffer, int size);

  int close();

 protected:
  int masterNode;
  int sock;
};

#endif
```

GOLApp.cpp

```
// ============================================================================
// (c) projet SAPPE
// Author : F. Zara
// modified by: Joe Hecker for use in lifgame
// ============================================================================

/** \file GOLApp.C
    Application permettant la visualisation des particules.
*/



/** Librairies de base **/
#include <iostream>
#include <math.h>
#include <Math/vjQuat.h>
#include <stdio.h>

/** std::vector **/
#include <vector>

/** Pour l'emploi des sockets **/
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#include <strings.h>
#include <fcntl.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/uio.h>
#include <netinet/in.h>

#include "GOLApp.h"


/** Namespace de la std **/
using namespace std;

struct msg_init
{
  int nx,ny;
};


/**
 * Execute les initialisations necessaires avant le lancement de l API.
 * Initialisation des services.
 */
void GOLApp::init()
{
  vjDEBUG(vjDBG_ALL,0) << "---------- Particules:App:init() ---------------"
                       << endl << vjDEBUG_FLUSH;
  /*** Preparation de la connexion vers le programme A1 ***/
  /** Message recu **/
  sock = NULL;
  sock = new NJSocket();
  if (sock!=NULL) {
    /** Nom du serveur **/
//    if (sock->connect("algonquin",60125)<0)
//   if (sock->connect("oglala",60125)<0)
    if (sock->connect("koguis",60125)<0)
    //if (sock->connect("node2.cluster",4245)<0)
    //if (sock->connect("node3.cluster",4245)<0)
    //if (sock->connect("node4.cluster",4245)<0)
      std::exit(1);

    cell_state msgInit;

// ORIGINAL   if (sock->recv((char*)&msgInit,sizeof(msgInit))<0) std::exit(1);
    if (sock->recv((cell_state*)&msgInit,sizeof(msgInit))<0) std::exit(1);
    nx = msgInit.my_x;
    ny = msgInit.my_y;

    cells = new cell_state();
    next_cells = new cell_state();

// ORIGINAL   if (sock->recv((char*)cells,nx*ny*sizeof(cell_state))<0) std::exit(1);
//    if (sock->recv((cell_state*)cells,sizeof(cell_state))<0) std::exit(1);

std::cout<<"    INITIAL BLOCK OF CELLS HAVE BEEN RECEIVED"<<std::endl;
    cell_state ok;
    ok.state = 1;
    ok.time = 1;
    ok.my_x = 1;
    ok.my_y = 1;
/*    ORIGINAL
    sock->send(&ok, sizeof(char));

    sock->send(&ok, sizeof(char));
    sock->send(&ok, sizeof(char));
    sock->send(&ok, sizeof(char)); // on autorise 3 images d'avance
*/
    sock->send(&ok, sizeof(cell_state));

    sock->send(&ok, sizeof(cell_state));
    sock->send(&ok, sizeof(cell_state));
    sock->send(&ok, sizeof(cell_state)); // on autorise 3 images d'avance
  }
/*  else
  {
    nx = ny = 16;
    cells = new cell_state[nx*ny];
    for (int i=0;i<nx;i++){
      for (int j=0; j<ny;j++){
        cells[i*nx+j].state = 'N';
        cells[i*nx+j].time = 0;
        cells[i*nx+j].my_x = i;
cells[i*nx+j].my_y = j;
      }
    }
  }
*/
```

```
  printf("INIT OK\n");
}


/**
 * Appelee immediatement lors de l ouverture d un contexte OpenGL.
 * Appel fait une fois pour chaque fenetre d affichage ouverte.
 * Ressource OpenGL allouee dans cette fonction.
 */
void GOLApp::contextInit()
{
  glDisable(GL_CULL_FACE);

  /** Definition de la couleur de la fenetre **/
  glClearColor(0.1, 0.2, 0.5, 1); // Bleu
  //glClearColor(1, 1, 1, 1); // Blanc
  glClear(GL_COLOR_BUFFER_BIT);// | GL_DEPTH_BUFFER_BIT);

  glIndexi(0);

  glDisable(GL_LIGHTING);

  /** Depth Buffer **/
  glClearDepth(1.0);
  glDepthFunc(GL_LESS);
  glEnable(GL_DEPTH_TEST);

  /** Choix de la technique du calcul de l ombre **/
  glShadeModel(GL_SMOOTH);
}

// Fonction appelee apres la mise a jour du tracker mais avant le debut du dessin.
// Calculs et modifications des etats faits ici.
void GOLApp::preFrame()
{
}


/**
 * Appelee immediatement lors de la fermeture d un contexte OpenGL.
 * (appele lors de la fermeture d une fenetre d affichage).
 * Ressource OpenGL desallouee dans cette fonction.
 */
void GOLApp::contextClose()
{
}

unsigned char palette[256][4];

/**
 * Dessin de la scene.
 */
void GOLApp::draw()
{
  glClear(GL_DEPTH_BUFFER_BIT);
  //printf(".\n");

  glPushMatrix();
  glTranslatef(-0.25f, 4.f, -1.05f);
```

```
//  glTranslatef(nx,ny,-1);
  glScalef((float)1/(float)nx,(float)1/(float)ny,1);//(float)1/(float)ny);
//     glTranslatef(0,0,-6);
  glTranslatef(-nx,-ny,-1);


  cell_state* cur = cells;

  unsigned char roygbiv[4];
  int time = cur->time % 1536;
//////////////////////////////////////  VISUALIZATION 1  ///////////////////////////////////////////////
/*
//  For visualization of cell state in 2D
//  A cell that is alive (state = true) appears in red, a cell that is dead (state = false) appears in blue
  if (cur->state){roygbiv[0]=255;roygbiv[1]=0;roygbiv[2]=0;roygbiv[3]=0;}
  else { roygbiv[0]=0;roygbiv[1]=0;roygbiv[2]=255;roygbiv[3]=0;}
*/

//////////////////////////////////////  VISUALIZATION 2  ///////////////////////////////////////////////

#define NbrColoursForTime 5
//  For visualization of task execution time 2D
//  Discrete color changes over time (red, orange, yellow, green, blue)
  int modu = time% NbrColoursForTime;
  switch (modu){
    case 0:  {roygbiv[0]=255;roygbiv[1]=0;roygbiv[2]=0;break;}
    case 1:  {roygbiv[0]=255;roygbiv[1]=175;roygbiv[2]=0;break;}
    case 2:  {roygbiv[0]=255;roygbiv[1]=255;roygbiv[2]=0;break;}
    case 3:  {roygbiv[0]=0;roygbiv[1]=255;roygbiv[2]=0;break;}
    case 4:  {roygbiv[0]=0;roygbiv[1]=0;roygbiv[2]=255;break;}
    default :  // should be treated differently
              {roygbiv[0]=255;roygbiv[1]=0;roygbiv[2]=0;break;}
  }


//////////////////////////////////////  VISUALIZATION 3  ///////////////////////////////////////////////
/*
//  For visualization of task execution time in 2D
//  Gradual increment of color through the spectrum over time (black-red-orange-yellow-green-blue-black-...)
      if (time<256){ roygbiv[0]=time;roygbiv[1]=0;roygbiv[2]=0;roygbiv[3]=0;}
      else if ( (time >= 256) && (time<514)){ roygbiv[0]=255;roygbiv[1]=(time%256);roygbiv[2]=0;roygbiv[3]=0;}
      else if ( (time >= 512) && (time<768)){ roygbiv[0]=255-(time % 512);roygbiv[1]=255;roygbiv[2]=0;roygbiv[3]=0;}
      else if ( (time >= 768) && (time<1024)){ roygbiv[0]=0;roygbiv[1]=255;roygbiv[2]=(time%768);roygbiv[3]=0;}
      else if ( (time >= 1024) && (time<1280)){ roygbiv[0]=0;roygbiv[1]=255-(time%1024);roygbiv[2]=255;roygbiv[3]=0;}
      else if ( (time >= 1280) && (time<1536)){ roygbiv[0]=0;roygbiv[1]=0;roygbiv[2]=255-(time%1280);roygbiv[3]=0;}
*/

//////////////////////  DISPLAY OF VISUALIZATIONS ////////////////////////////////////////
  glTranslatef(0,0,-6);
  glBegin(GL_QUADS);
      glColor4ubv(roygbiv);
      glVertex2i(2*cur->my_x,2*cur->my_y);
      glVertex2i(2*cur->my_x+1,2*cur->my_y);
      glVertex2i(2*cur->my_x+1,2*cur->my_y+1);
      glVertex2i(2*cur->my_x,2*cur->my_y+1);

  glEnd();

  glPopMatrix();
}
```

```
/*****************************ORIGINAL***********************************************
void GOLApp::draw()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  //printf(".\n");
  glPushMatrix();

  glTranslatef(-0.25f, 4.f, -1.05f);
  //glTranslatef(0,-3,1);

  glScalef((float)1/(float)nx,(float)1/(float)ny,(float)1/(float)ny);

  glTranslatef(-nx,-ny,0);

  palette['R'][0]=255;
  palette['G'][1]=255;
  palette['B'][2]=255;
  palette['N'][0]=0;
  palette['N'][1]=0;
  palette['N'][2]=0;


  glBegin(GL_QUADS);
  cell_state* cur = cells;
  for (int y=0;y<ny;y++)
  {
    for (int x=0;x<nx;x++)
    {
      glColor4ubv(palette[cur->state]);
      glVertex2i(2*x,2*y);
      glVertex2i(2*x+1,2*y);
      glVertex2i(2*x+1,2*y+1);
      glVertex2i(2*x,2*y+1);
      cur++;
    }
  }
  glEnd();

  glPopMatrix();

********************************************************************************/
void GOLApp::nextFrame()
{
  if (sock!=NULL) {
//    sock->recv((char*)next_cells,nx*ny*sizeof(cell_state));
    sock->recv((cell_state*)next_cells,sizeof(cell_state)); //POSSIBLE ERROR CAUSER
    cell_state ok;
    ok.state = 1;
    ok.time = 1;
    ok.my_x = 1;
    ok.my_y = 1;
    sock->send(&ok, sizeof(cell_state));
//    char ok = 1;
//    sock->send(&ok, sizeof(char));

  }
 }
```

```
void GOLApp::postFrame()
{

  if (sock!=NULL) {
    cell_state* temp = cells;
    cells = next_cells;
    next_cells = temp;
  }
}
```

GOLApp.h

```
// ==========================================================================
// (c) projet SAPPE
// Author : F. Zara
// modified by: Joe Hecker for use in lifgame
// ==========================================================================

/** \file ParticulesApp.h
    Application permettant la visualisation des particules.
*/



#ifndef _PARTICULES_APP_
#define _PARTICULES_APP_

/** Librairies de base **/
#include <stdio.h>
#include <iostream>
#include <math.h>
#include <algorithm>
#include <strings.h>

/** std::vector **/
#include <vector>

/** GLUT **/
#include <GL/gl.h>
#include <GL/glu.h>

/** VR Juggler **/
#include <vjConfig.h>
#include <Kernel/GL/vjGlApp.h>
#include <Kernel/GL/vjGlContextData.h>
#include <Math/vjMatrix.h>
#include <Math/vjVec3.h>
#include <Kernel/vjDebug.h>
#include <Input/InputManager/vjPosInterface.h>
#include <Input/InputManager/vjAnalogInterface.h>
#include <Input/InputManager/vjDigitalInterface.h>
#include <Kernel/vjUser.h>

/** Pour la sauvegarde des images **/
//#include <tiffio.h>

/** Pour l emploi des sockets **/
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/** Pour la visualisation **/
#include "cell_state.cpp"
#include "NJSocket.h"

/*
typedef struct cell_state {
  char state;
  int time;
  int my_x;
  int my_y;
};
*/
/**
 * Application permettant la visualisation des particules.
 */
class GOLApp : public vjGlApp
{
 public:
   /// Ajout d un constructeur
  GOLApp(vjKernel* kern) : vjGlApp(kern) { ; }

  /// Ajout d un destructeur
  virtual ~GOLApp() {}

  // Execute les initialisations necessaires avant le lancement de l API.
  // Initialisation des services.
  virtual void init();

  // Execute les initialisations necessaires apres le lancement de l API
  // mais avant que le drawManager ne commence la boucle d affichage.
  virtual void apiInit()
    {
      vjDEBUG(vjDBG_ALL,0) << "---- ParticulesApp::apiInit() ----\n" << vjDEBUG_FLUSH;
    }

  // Appelee immediatement lors de l ouverture d un contexte OpenGL.
  // Appel fait une fois pour chaque fenetre d affichage ouverte.
  // Ressource OpenGL allouee dans cette fonction.
  virtual void contextInit();

  // Appelee immediatement lors de la fermeture d un contexte OpenGL.
  // (appele lors de la fermeture d une fenetre d affichage).
  // Ressource OpenGL desallouee dans cette fonction.
  virtual void contextClose();

  /**   name Drawing Loop Functions
    *
    *  The drawing loop will look similar to this:
    *
    *  while (drawing)
    *  {
    *        preFrame();
    *       draw();
    *        intraFrame();      // Drawing is happening while here
  *       sync();
  *        postFrame();      // Drawing is now done
  *
  *       UpdateTrackers();
  *  }
  *
  */

// Fonction appelee apres la mise a jour du tracker mais avant le debut du dessin.
// Calculs et modifications des etats faits ici.
virtual void preFrame();

// Fonction de dessin de la scene.
virtual void draw();

// Fonction appelee apres que le dessin soit declenche mais AVANT qu il soit fini.
virtual void intraFrame();
```

```
  // Fonction appelee avant la mise a jour des trackers mais apres que la frame soit dessinee.
  // Calculs effectues ici.
  virtual void postFrame();

 private:

  // Message
  NJSocket* sock;
  int nx, ny;
  cell_state* cells;
  cell_state* next_cells;
};


#endif
```

11 Frequently Asked Questions

This section contains a list of frequently asked questions about Athapascan (and some attempts at answering ;-)). Please feel free to send us any questions that would enable us to enlarge this section.

Q: On which systems do Athapascan run ?
A: Currently Athapascan has been tested on:

- IBM-SPx running aix-4.2 using LAM-MPI or a dedicated switch with
  - `xlC` 4.2 C++ compiler
- Sparc or Intel multiprocessor and network of workstations using LAM-MPI with
  - `CC` 4.2 C++ compiler

Athapascan-0 is currently supported on:

- AIX 3.2.5 and IBM/MPI-F, IBM SP with AIX 4.2 and IBM/MPI
- DEC/Alpha with OSF/1 4.0 and LAM/MPI 6.1
- HP-9000 with HP-UX 10.20 and LAM/MPI 6.1
- SGI/MIPS with IRIX 6.3 and LAM/MPI 6.1, SGI/MIPS with IRIX 6.4 and SGI/MPI-3.1
- Sparc or Intel with Solaris 2.5 and LAM/MPI 6.3
- Intel with Linux 2.0.25, MIT threads 1.60.6 and LAM/MPI 6.3

Q: How do I get a copy of Athapascan?
Q: Where can I comment about Athapascan?
Q: How do I get up-to-date information?
A: There is a web page dealing with Athapascan at `http://www-apache.imag.fr`. The Athapascan distribution, the manual (the document you are reading) and some other related papers are also available from this web page.

Q: The compilation failed: `A1_MAKEFILE` not known! What do I do?
A: Check to see if you properly set up your environment by sourcing the appropriate setup file (the ones for Athapascan-0 and Athapascan).

Q: The option `-a1_trace_file` has no effect at execution! Why could this be?
A: Make sure you are using a program compiled with an appropriate Athapascan library (one compiled to generate dynamic graph visualization information).

Q: An Athapascan internal error occurs at execution! What can I do to correct this error?
A: If you are using MPI-LAM, please clean-up and reboot LAM before executing your Athapascan program. If the problem persists, please follow the instructions on the Athapascan webpage.

Q: The compiler does'nt find a task corresponding to my `a1::Fork` instruction. Why could this be?
A: Make sure that all the shared modes and rights are compatible.
A: Make sure the procedure does not have too many arguments. If so, recompile your library after having increased the authorized number of parameters at configuration (option `nbp` of `configure` script).

Q: I have tried all the previous suggestions and I still have some errors. What shall I do?
A: Send an e-mail to `Jean-Louis.Roch@imag.fr` stating your problem.

[]

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Futurs : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803