Athapascan-0
User Manual
Version 2.4.13 - Mar 10, 1998

Jacques Briat
Ilan Ginzburg
Marcelo Pasin

17th March 1998

Abstract

This document explains how to use Athapascan-0, a multi-threaded, portable, parallel programming runtime system, designed to solve efficiently large irregular problems. It is not intended to be a research paper, but to show how to program, compile and execute programs using Athapascan-0. Nevertheless, this introduction gives some explanation about on what Athapascan-0 is focused, which can lead to better understanding of it.

To exact function syntax, refer to the Reference Manual.

Please, send all your comments about this manual to the authors, so perhaps one day it would read well.

Contents

1 Introduction

The currently traditional parallel programming approaches, as communicating processes and data parallelism, can easily exploit hardware parallelism when the computation can be split into smaller equally weighted computation processes, with good data locality. This type of parallel computation is called **regular**. Simple mapping and scheduling algorithms can be used to exploit satisfactorily the hardware parallelism. The same does not happen if the computations do not have equivalent sizes (or they are not previously known) or do not have good data locality properties. Parallel computations like that are called **irregular**.

Traditional parallel programming approaches, such as communicating processes (PVM, MPI) and data partitioning (HPF), face the same problem differently. The former offers low-level programming tools, allowing to write a large class of algorithms and to handle efficiently irregular problems. Nevertheless, a considerable programming effort is necessary, because the programmer must predict all possible cases of unbalance, in order to adapt the strategy of data and computation distribution. The latter approach is not currently adapted to process irregular problems, and leads, in this case, to non-efficient programs.

The programming model of Athapascan-0 exploits two levels of parallelism. The first level is the parallelism between computers, implemented with system **processes**. It is a coarse grain parallelism, due to the operating system overhead, the lack of physically shared memory and the low speed of the computer network if compared to the memory bus. The second level of parallelism appears inside a computer, between a computation processor and its attached communication processor or between processors in a symmetric multiprocessor machine.

To achieve better performance, the fine granularity must be implemented by inexpensive mechanisms. We propose the use of **threads** as the execution support of a task, with local and remote thread handling functions. With this approach, an irregular algorithm is expressed as a **dynamic network of communicating threads**.

Parallel machines are considered as generic networks of heterogeneous symmetric multiprocessors, with remote thread creation and message exchange. Athapascan-0 can dynamically map fine granularity computations on such machines and overlap communication delays with computation. These features allow to perform better load sharing and therefore to achieve better performance.

2 Programming with Athapascan-0

An Athapascan-0 **parallel machine** is a set of **nodes**. A node is implemented by a process and a physical computer can execute many of them. All executable modules of the nodes' processes can be the same or be different from each other.

The indication of the number of nodes necessary to a given parallel processor, their allocation to processes and the loading of the executable modules are not part of Athapascan-0. All that must be provided by the underlying process communication environment. The current implementation uses MPI, which imposes an SPMD model (Single Program Multiple Data), where all nodes execute the same module.

2.1 Athapascan-0 Objects

The execution of an Athapascan-0 program involves many objects implementing the aforementioned concepts. Some of these objects exist at startup time, others must be explicitly created and destroyed. Some objects also require an uniform naming across all nodes. The Athapascan-0 objects are:

- **Nodes**: the heavyweight processes. The set of nodes is fixed at startup time and cannot be modified. They are numbered from 0 to the total number of nodes minus one.
- **Threads**: the lightweight processes. An initial thread begins to execute the `main()` function in every node. Supplementary threads can be directly cre-

ated locally on a node or indirectly created remotely on other nodes, by starting a remote service.

- **Services**: the identifiers of functions executed by starting remote service. The services executed remotely must be registered on the remote node. The registration procedure is identical on all the nodes, so the identification of a service has a system-wide meaning.
- **Ports**: the addresses to communicate. Messages are sent to a given port of a remote node. Only threads executing on the addressed node can receive messages from the addressed port, but the addressed port may have been created in any node. The port creation algorithm can control whether the ports created by a node will be different or not from the ports created by other nodes.
- **Requests**: the state of a communication. Any function call involving communication (message send and receive, remote thread creations, etc.) have an associated request. Those functions can be called by blocking or not the calling thread until they end. The request is used to wait or test the termination of the communication in the case they are demanded non-blocking. The same request can also be used to verify the correct termination of the communication.

2.2 Main Module and Startup

The executable module of every node starts executing the `main()` function of its C program. The first Athapascan-0 function call must be `a0Init()`, followed by some other Athapascan-0 function calls, and then `a0InitCommit()`. This two-phase initialization procedure allows the programmer to safely initialize critic objects. This procedure guarantees across all nodes a coherent naming scheme of the objects.

All the services must be declared between `a0Init()` and `a0InitCommit()`, in such a way they are all performed in the same order in all nodes. The ports that are supposed to be created identical in all nodes must also be declared there.

```
#include <ath0.b>

/* a service and its function */
int Serv1;
a0tError FuncServ1(void *data)
{  ...
}

/* another service and its function */
int Serv2;
a0tError FuncServ2(void *data)
{  ...
}

int main(int argc, char *argv[])
{
  a0Init(&argc, &argv);
  a0NewService(&Serv1, FuncServ1, ...);
  a0NewService(&Serv2, FuncServ2, ...);
  a0InitCommit();
  ...
  a0Terminate();
}
```

Athapascan-0 is not usable before executing `a0Init()`.

After executing `a0Init()`, all nodes are ready to initialize the static Athapascan-0 objects. In this phase, all library module initialization functions can be called (see next sections). Declaration of service handlers and port creations are valid Athapascan-0 operators in this step. Global variables of the module can be initialized and synchronization objects like semaphores or mutexes can be created either during or after the initialization phase. Thread creation or communication are not allowed before `a0InitCommit()`.

When the execution starts, only one thread is running on each module. This thread is called **the main thread**, as it executes the `main()` function. When the main thread execution reaches `a0InitCommit()`, the executing modules can use the full Athapascan-0 operator set.

After finishing all computations, at the end of the program, all the nodes must call `a0Terminate()`. When all nodes reach `a0Terminate()`, they stop execution and therefore the whole parallel program terminates. It is strongly encouraged that `a0Terminate()` be executed only by the main thread, because `exit()` is called after the return of its function, which would abnormally terminate the Athapascan-0 program. Though, at any point of execution, the parallel program may be canceled by a `a0Abort()`, after an user-detected irrecoverable error.

The variables `a0SelfNode` and `a0NodeCount` contain the local node identification number and the total number of nodes available in the parallel machine, respectively. This variables can be used to modify the behavior of the program, depending on the current node and on different quantities of nodes.

2.3 Program Arguments Parsing

The address of the arguments `argc` and `argv` of the `main()` function must be given to the `a0Init()` function. Athapascan-0 extracts from the program's arguments its own parameters and those of the underlying layers. This extraction is done on a reserved keyword basis and all options that are interpreted by Athapascan-0 or the underlying layers cannot be seen by the user. Although all the Athapascan-0 options start with an `-a0` prefix, no assumption is made about the underlying layers' options.

```
a0run myprogram -n 4 -a0stack 10000 test -mypar 54
```

In the above example command line, the arguments `-n 4` will be extracted by the underlying layers, in order to run four copies of the program in the parallel machine. The arguments `-a0stack 10000` will be extracted by Athapascan-0 and every copy of the program `myprogram` will have `test`, `-mypar` and `54` as arguments.

2.4 Descriptor Allocation

All descriptors defined in an Athapascan-0 program (for threads, ports, requests, semaphores, etc.) have to be previously allocated in the memory of the executing module. All the functions that define new objects expect to receive a pointer to the object to be defined, which is just "filled in".

```
a0tBuffer *create_buffer(long size)
{
  a0tBuffer *new;

  new = (a0tBuffer *) malloc(sizeof(a0tBuffer));
  a0NewBuffer(new, A0SendBufferType, size);
  return new;
}
```

The use of automatic[1] variables within a function, should be used as a descriptor if, and only if, they are created, used and destroyed in the same function, because its allocation space disappears after the end of the function.

```
void send(...)
{
  a0tBuffer buff;
```

[1] defined with `auto` in C

```
  a0NewBuffer(&buff, ...);
  a0Pack(&buff, ...);
  ...
  a0SendBuffer(..., &buff);
  a0DisposeBuffer(&buff);
}
```

2.5 Library Modules

The implementation of an initialization function is encouraged, when developing Athapascan-0 user libraries. Each module can provide different services and global variables should be associated to them. All the services provided by a module of a library should have their declaration in the initialization function.

```
/**** module fourier.c, a library module ****/
#include <ath0.b>

/* exported services */
a0tService  ServFourier1;
a0tService  ServFourier2;

/* service functions */
static a0tError FuncFourier1()
{  ...
}
static a0tError FuncFourier2()
{  ...
}

/* module initialization */
void InitFourier()
{
     a0NewService(&ServFourier1, FuncFourier1, ...);
     a0NewService(&ServFourier2, FuncFourier2, ...);
}
```

Using an initialization function, a program that uses services of some library modules can execute all the initialization functions of all library modules in the same order. This guarantees that all services and ports created by those modules will receive the same identifiers.

```
/***** program who uses fourier library *****/
#include <ath0.b>
#include <fourier.h>

int main(int argc, char *argv[])
{
  a0Init(&argc, &argv);
  InitFourier();
  ...
  a0InitCommit();
  ...
}
```

3 Communication

Athapascan-0 provides various functions to transfer data between threads of two different nodes. They allow the user to specify the origin or destination of the data, as well as what kind of data are to be transferred. There are blocking and non-blocking (also called asynchronous) communication functions, to take profit of the architectures that can compute and communicate at the same time.

The basic communication functions operate with typed data. All C language scalar basic **data types are supplied to the user. A higher-level library, called Athapascan-0 Formats is available to express complex data, pipeline communications and remote memory accesses.**

3.1 Requests

In both synchronous and asynchronous communication, a function call have a request descriptor between its arguments. In a synchronous call, it is used by the kernel to hold the context of the blocked thread until its communication is terminated. In an asynchronous call, it is used to store the current status of the communication (whether it is completed or not).

The completion of a send is achieved when the data to send from the user space were buffered or sent to the destination node by the underlying communication library. The sent data cannot be de-allocated or modified before the completion of the request. The completion of a receive is achieved when the data to receive were already stored in the user space. The received data cannot be read before the completion of the request.

After the completion of a communication, its associated request contains no useful information for Athapascan-0. It can be reused right after the return of the synchronous communication function that required it. Nevertheless, after the return of a receive call, the request contains the number of elements actually received and the source of the message. They can be obtained from the request by calling the function `a0GetRequestData()`**.**

When the user has no intention to inspect the contents of the request after the completion of the communication, a `NULL` **value can be used. The synchronous Athapascan-0 communication functions will use an automatic variable to control the request instead of requiring the user to pass his/her own's. The asynchronous functions called with a** `NULL` **request is allowed, but they cannot have their completion tested. The programmer must have other means to identify the end of the request (a reply message, for example).**

3.2 Ports

The identification of the origin and the destination of a message is done using a port, a node number and a tag. In the process of sending a message, the port and the tag are collated to the data into a message and sent to the destination node. In the destination node, when a receive is performed, it will be completed at the arrival of a message whose port, tag and source node are the same as specified. For a given port, node and tag, the messages arrive in the same exact order in which they were sent.

Sender and receiver have to agree about which port and which tag to use. It is up to the programmer to choose the correct values, but Athapascan-0 provides some functionality to allow the programmer to use the same ports in different nodes. Because Athapascan-0 identifier generators are deterministic, all objects get the same identifier on all nodes if the initialization sequence is the same on all the nodes. It allows to create the same ports in different nodes at the same time.

There are two ways of creating ports: global or local creation. Note that it is the creation that is either local or global. Ports are always local to a given node. The global creation must be done between the calls of `a0Init()` **and** `a0InitCommit()`**. The local creation can be performed anywhere in the program. The function** `a0NewPort()` **is used to perform both local or global creation.**

If the parameter `type` **is** `A0GlobalCreation`**, all the nodes must call it simultaneously, and the returned value will be the same to all nodes. Remember that the creation can be global only if** `a0NewPort()` **is called between** `a0Init()` **and**

a0InitCommit(). In that place, as all nodes are expected to execute the same exact code, and the port generation is deterministic.

```
int main(int argc, char *argv[])
{
  ...
  a0Init(&argc, &argv);
  a0NewPort(&Port1, 1, A0GlobalCreation); /* all create same port */
  a0InitCommit();

  /* node 0 sends an integer to node 1 */
  if (a0SelfNode == 0)
    a0Send(&Port1, /*node*/ 1, ...);

  /* node 1 receives an integer from node 0 */
  if (a0SelfNode == 1)
    a0Receive(&Port1, /*node*/ 0, ...);
}
```

If `type` is `A0LocalCreation` when calling `a0NewPort()`, the port created will never be returned by any other call of `a0NewPort()`, either in the same node or not. As ports are just identifiers, whose can be referenced by any thread of any node, a port can be created in a node and used in another one. Ports can then be passed inside a message or in the parameters to a remote thread creation.

A common mean of communication between two threads, when one creates the another, is to pass a port in the arguments of the created thread. A port would be created by parent thread and stored in a buffer, that being used to create a child thread remotely. This port is then known by both parent and child thread, whose can use it to communicate to each other.

```
a0tError my_service(a0tBuffer *input)
{
  ...
  a0Unpack(input, ..., &port);
  a0Receive(&port, /*node*/ 0, ...);
  ...
}

  ...
  a0NewPort(&port, 1, A0LocalCreation);
  a0Pack(buff, ..., &port);
  a0CreateRemoteThread(/*my_service at node 1*/, buff);
  a0Send(&port, /*node*/ 1, ...);
```

A thread usually communicates with its neighbors, whom it is convenient to designate by a single index. For this reason the communication functions have a port and a tag associated. In the allocation process, the program specifies the maximum number of tags to be used with the new port. The kernel creates as many consecutive ports as the number of tags asked and returns the number of the first port. All the communications identify their destinations using a base port number and a tag, the latter being added to the former to find the actual port number.

3.3 Synchronous Communication

The functions `a0Send()` and `a0Receive()` are the basic thread blocking communication operators. These operators allow communication between any threads in the parallel machine. By the means of these functions, any data can be exchanged between remote threads. Additional arguments to a remote thread creation can be passed and results can be received.

When a thread calls a synchronous communication function, it stays blocked until the communication is completed. This means that the user data can be freely accessed right after the return of a blocking call. All the other threads keep running without any interference when the communicating thread is blocked.

```
...
/* node 0 sends an integer to node 1 */
if (a0SelfNode == 0)
  a0Send(port, /*node*/ 1, 0, &req, a0_INT, &data, 1);

/* node 1 receives an integer from node 0 */
if (a0SelfNode == 1)
  a0Receive(port, /*node*/ 0, 0, &req, a0_INT, &data, 1);
...
```

3.4 Asynchronous Communication

Athapascan-0 provides non-blocking send and receive in order to exploit the ability of the underlying communication kernel to advance independent communications as soon as it is possible. Every call to an asynchronous communication function return immediately and the associated request holds the current status of the communication. The completion of such communication can be tested with the function `a0TestRequest()`. Similarly, the function `a0WaitRequest()` blocks the calling thread until the communication request is completed.

In the following example, a thread sends data asynchronously to several others, do some processing, and then wait for the completion of the communications to reuse the sent variables. Note that the function `a0ISend()`[2] is used instead of `a0Send()`.

```
float Data[SIZE*NUMBER];
...
a0tRequest Request[NUMBER];
for(i=0; i<NUMBER ; i++)       /* asynchronous arguments spreading */
  a0ISend(&Port[i], Node[i], TAG, &Request[i],
          a0_FLOAT, &Data[SIZE*i], SIZE);
... compute ...
/* wait for completion before reusing Data[] array */
for(i=0; i<NUMBER ; i++)
  a0WaitRequest(&Request[i]);
```

The same construction is possible with `a0IReceive()`. In this case, the `a0WaitRequest()` must be done before using the received data. The main advantage of the asynchronous communication is to provide an easy way to deal with race-free, deadlock-free and buffer-size-independent communications.

```
int Neighbor[4];               /* Four neighbors */
a0tRequest Request[8];         /* 8 communications */
...
for(i=0;i<4;i++) {             /* initialize the exchange */
  a0ISend(&Port, Neighbor[i], TAG1, &Request[i],
          &SendType, &SendData[i], count);
  a0IReceive(&Port, Neighbor[i], TAG2, &Request[i+4],
          &ReceiveType, &ReceiveData[i], count);
}
...
for(i=0; i<8; i++)             /* wait completion */
    a0WaitRequest(&Request[i]);
```

[2]I stands for immediate return.

3.5 Buffers

When the data to send are stored in memory regions with variable offsets, that cannot be defined by a simple data type, the user can pack them into a buffer and then send the buffer. On the other side, the user receives data into a buffer and unpacks them. However, the user cannot send some data using a normal `a0Send()` and receive using a buffer neither vice-versa. It is also up to the user to certify that the unpacked data from the buffer will have the same type of the data packed in it.

A buffer must be defined before using it, with the function `a0NewBuffer()`. The function `a0Pack()` is used to pack data into a buffer and `a0Unpack()` to the unpack them. The functions to send and receive a buffer are respectively `a0SendBuffer()` and `a0ReceiveBuffer()`. There are also non-blocking functions, called `a0ISendBuffer()` and `a0IReceiveBuffer()`.

```
a0tBuffer Buf;
float F[10];
int I[20];
...
a0NewBuffer(&Buf, A0SendBufferType, BUFSIZE);
...
a0Pack(&Buf, a0_FLOAT, &F[0], 10);
a0Pack(&Buf, a0_INT,   &I[0], 20);
a0SendBuffer(&Port, Dest, Tag, &Buf);
```

4 Multi-threading

Threads may be created locally or remotely. By using remote thread creation, a service is executed on a given node. The mapping of a thread onto a node must be explicit. All threads of a node cooperate via shared data and synchronizations objects such as mutexes and semaphores. Threads placed on different nodes cooperate by message exchanges.

There are several forms of thread creation in Athapascan-0. Threads can be created in the local node or in a remote node. The threads created locally share the memory with the creator. Remotely created threads, which don't share memory with the creator, can receive a block of data at their startup.

4.1 Local Threads

Threads created locally are called slave threads. They can be created by any existing thread, even another slave thread. At the moment of the creation of any thread, the function that it executes must be specified (the main thread runs the `main()` function). The function to be executed by a slave has a special prototype, but no service must be declared to start it.

```
a0tError slave1(void *arg)
{
    ... compute ...
}

  a0tThread ThrSlave;
  ...
  a0NewSlave(&ThrSlave, ..., slave1, arg);
```

A thread can create a slave when it needs to wait for some event and some computing can be done at the same time. Creating a slave and later waiting for its termination to get its result is a limited form of a parallel function call. Though, it is necessary to explicitly synchronize them if they have some level of dependency.

A daemon thread can also be created, when some background task must be done while the other threads execute. A daemon thread is said to be detached from the creator thread (`a0DetachSlave()`) and its termination cannot be waited.

Any thread can terminate itself by returning its main function or by calling the function `a0ExitThread()`. In the latter case, a result can be passed to the thread who will wait its termination. The result has type `void *`, and a little integer could be easily passed inside of it with a cast. To return bigger structures, note that data must not have been automatically allocated, because their storage will disappear as soon as the thread returns.

```
/* slave function definition*/
a0tError Func(void *argument)
{
  /* malloc to return 10 floats in it. */
  float *MyResult = malloc(10 * sizeof(float));
  ...
  a0ExitThread(MyResult);
}
...
float Data[100];
float Result;
a0tThread Slave;
...
a0NewSlave(&Slave, SchedulingRule, Priority, StackSize,
           Func, (void *)&Data);
...
a0JoinSlave(&Slave, &Result);
```

4.2 Synchronization

Any thread may create others slave threads with the function `a0NewSlave()`, to assist to compute some values or to communicate with other threads. The operators to synchronize local threads provided by Athapascan-0 are mutexes,semaphores, and condition variables.

Mutexes are used to build critical sections in which at most one thread executes. Semaphores can be seen as a bag of execute rights in which threads insert a ticket (V function) or try to get a ticket P function). If a thread does a P when there are no tickets in the bag, it waits until some other thread inserts one in it.

Conditions variables are used in conjuction with mutexes to allow a thread to wait until an arbitrary condition has occurred. There are two basic operations on a condition variable: signalling and waiting for it to be signalled. First, one or more thread wait on a condition variable. Then, when a condition variable is signalled, one or all of the threads (as specified by the signaller) waiting for the the condition variable are allowed to proceed. A singal on a condition that does not have any waiting threads is not remembered. The next thread that waits ont the condition variable will block until the condition variable is again signalled. in other words, condition variables are ”stateless”.

In the example below, a mutex is created when there is only one thread running. Later on, several threads need to store values at the end of a table. The incrementation of the table counter and the store of the new element at the end of the table must be done atomically. This is guaranteed by the mutex, which is locked before and unlocked after modifying the table.

```
a0tMutex CounterMutex;
int Table[MAX];
int Count = 0;
...
/* only one thread running, create mutex */
a0NewMutex(&CounterMutex);
```

```
...
/* several threads run, lock and unlock mutex */
a0LockMutex(&CounterMutex);
Table[Count++] = NewValue;
a0UnlockMutex(&CounterMutex);
```

4.3 Scheduling Rules and Priority

The processor allocation to ready threads is controlled by the scheduling rules inherited from POSIX. The scheduling rules are the following, in decreasing order of priority:

- First-In-First-Out (FIFO): by this rule, one of the threads of the highest priority run to completion or blocking (on a mutex, a semaphore, a communication, etc.). Threads may starve if a higher priority thread runs continuously.
- Round-robin: by this rule, the threads of the highest priority run in a time-sharing basis, if there are no FIFO threads to execute. There is no starvation in a priority level but threads starve if a thread on a higher priority level or rule runs continuously.
- Other: It can be anything. Most of the POSIX threads libraries used to compile Athapascan-0 have an *other* scheduling rule, in which the implementors are free to do whatever they want. In general, that rule means round-robin with no priorities, i.e., all threads run in a time-sharing basis.

The scheduling rule and priority of a thread are established at its creation time. They can be given as default values or be inherited from the creating thread. They can also be changed during the existence of the thread, by the thread itself or by any other thread.

Not all scheduling rules neither different priorities are supported over all implementations of Athapascan-0. Though, all the rules and the priorities range provided by the POSIX threads used are given to the user. The POSIX norm requires the implementors to provide at least the *other* scheduling, so does Athapascan-0.

4.4 Remote Threads

Prior to execute one thread remotely, a service must be declared with the function `a0NewService()`. The service identifier is a number which denotes a function, even local or a remote. All the service declarations must be done in the initialization phase, in order to guarantee their deterministic execution order. The declaration of a new service must give a scheduling rule, a priority and a stack size. A section below in the text explains how scheduling works.

```
int Service;

/* service function definition */
a0tError ServiceFunction(a0tBuffer *Input)
{
  a0Unpack(Input, ... );
  ...
}
...
a0Init(&argc, &argv);
a0NewService(&Service, ServiceFunction,
             SchedulingRule, Priority, StackSize);
a0InitCommit();
```

To create a thread in a remote node, the function `a0StartRemoteThread()` must be used, giving a service number, a scheduling rule, a priority and a stack size. The remotely started thread executes the procedure declared with the given service number with the given scheduling rule, priority and stack size. To use

the scheduling rule, priority and a stack size specified in the service declaration, use `A0ServiceScheduling` , `A0ServicePriority` and `A0ServiceStack` respectively.

```
a0tBuffer Buffer;
a0tRequest Request;
...
a0Pack(&Buf, ... );
a0StartRemoteThread(Node, Service,
                    SchedulingRule, Priority, StackSize,
                    &Request, &Buffer);
```

Active messages are also implemented, as a slightly modified remote service call, named urgent services. The declaration of an urgent service handler is done with `a0NewService()` and its activation is done with `a0StartRemoteUrgent()`. Instead of creating a new thread in the destination node, a procedure is executed directly by the urgent daemon thread of Athapascan-0. The scheduling, priority and stack information passed to `a0NewService()` are ignored, since no new thread is started. It allows the program to perform some quick operation remotely, but the service cannot use any blocking function.

5 Compiling, Linking, Executing

5.1 Environment

A single directory (which has subdirectories) contains all the files related to Athapascan-0. Normally this directory is `/usr/local` or `/usr/local/ath0b`, which must be stored in the shell environment variable `A0HOMEDIR`. A file `a0setup.csh`, contained in the `bin` subdirectory of `A0HOMEDIR`, sets all the shell environment variables necessaries to compile, link and run programs with Athapascan-0, as well as Athapascan-0 itself.

5.2 Making Athapascan-0

The currently systems supported by Athapascan-0 are:

- `ibm.aix4`**: IBM SP/2 running AIX 4.x, with AIX POSIX threads and IBM MPI.**
- `mpif.aix3`**: IBM SP/2 running AIX 3.2, with AIX DCE threads and IBM MPI-F.**
- `lam.hpux10`**: A network of workstations HP-9000 running HP-UX 10.x, with HP-UX DCE threads and LAM/MPI.**
- `lam.osf4`**: A network of workstations DEC Alpha running OSF/1 4.x, with OSF/1 POSIX threads and LAM/MPI.**
- `lam.sgi6`**: A network of workstations Silicon Graphics running Irix 6.x, with Irix POSIX threads and LAM/MPI.**
- `lam.sol2`**: A network of workstations (Sparc or Intel) running Solaris 2.5, with Solaris POSIX threads and LAM/MPI.**
- `lam.mit`**: A network of workstations Intel running Linux 2.x with MIT POSIX threads and LAM/MPI.**

Be sure to have installed in your system the threads and MPI packages necessary to compile Athapascan-0. After expanding the *tar*file, a directory called `ath0b` is created with the following contents:

- **file `Makefile`: Athapascan-0 main makefile**
- **file `README`: You should have read that already**
- **file `INSTALL`: basic installation instructions**

- **directory `src`: Athapascan-0 sources**
- **directory `doc`: Athapascan-0 manuals**
- **directory `examples`: (should have) some Athapascan-0 examples**
- **directory `include`: directory that holds include files (empty before Athapascan-0 compilation)**
- **directory `lib`: directory that holds library files (empty before Athapascan-0 compilation)**
- **directory `bin`: directory that holds binary utility files.**

To compile Athapascan-0 itself, the user must execute `make` being inside the directory `ath0b`. It generates the `bin/a0setup.csh` file and fills the directories `include`, `lib` and `bin`, with the installation files, which are the include files, the libraries generated by the compilation and the binary utilities.

To install the libraries in your system, execute `make install`. It will copy the contets of the `include`, `lib` and `bin` directories to the installation directory. Nevertheless, Athapascan-0 can be used uninstalled, in the same directory it was compiled.

In the doc directory you can find all the documents and manuals (like this, for example). They are supplied in LaTeX source. Postscript versions can be obtained by executing `make` in the `doc` directory. There is also a program which converts the LaTeX files into html, to be read with you preferred navigator.

5.3 Installing Athapascan-0

Usually, after having compiled all libraries of Athapascan-0, one can make a copy of it to make available to all user in the system. Though, there is no need to copy all the sources of Athapascan-0 for that. Athapascan-0 comes with an installation procedure, activated by `make install` in the main directory. It copies only the needed files generated in `include`, `lib` and `bin` directories to the destination directory (chosen with `configure`).

5.4 Compiling/Linking the Examples

The `examples` directory contains some example programs for Athapascan-0. To compile them, use `make`. Don't forget to run the `bin/a0setup.csh` file before.

There are also subdirectories for every Athapascan-0 layer (look at `src` directory), with little test programs. They can be compiled with `make all`.

5.5 Compiling/Linking an Athapascan-0 Program

To use Athapascan-0, at least two variables must be previously set: `A0HOMEDIR` and `A0TYPE`. Files `a0setup.csh` (for C-shell) and `a0setup.sh` (for Bourne-shell) set those variables and the `PATH` variable to find the MPI and Athapascan-0executables. To set all the variables needed, execute:

```
prompt% source ..../ath0b/bin/a0setup.csh
```

To compile an Athapascan-0 program, the user must execute the C compiler, passing all the needed options. There is a makefile in the file pointed by the variable `A0MAKEFILE` which can be used to compile and link any simple Athapascan-0 program. For example, to compile a program called `toto.c`:

```
prompt% make -f ${A0MAKEFILE} toto
```

Athapascan-0 supplies two front-end scripts, to compile and link Athapascan-0 programs, namely `a0cc` and `a0ld`. Their usage is the same as the regular MPI compiler of your system (`mpcc` with IBM MPI, `hcc` with LAM/MPI, `mpicc` with MPI-CH). They add all needed options and execute the system configured compiler. The option `-v` can be used to inspect the commands issued by those scripts.

```
prompt% a0cc -BLABLA -c -o file1.o file1.c
prompt% a0cc -FOOBAR -c -o file2.o file2.c
prompt% a0ld -OPTION -o myprog file1.o file2.o file3.o
```

As `a0cc` and `a0ld` are just front-ends to a C compiler and linker, all the options passed to both are passed to the back-end compiler and linker. For example, to see which compiler is executed and its options, one can use the `-v` option, as follows:

```
prompt% a0cc -c -v emamp.c
================a0cc output================
hcc -c -I/users/navajo3/pasin/a0/sparc.lam.sol2/include -DA0_TSOL2
-DA0_LAM -c -v examp.c
================system output================
"examp.c", line 45: warning: Function has no return statement : examp
```

There are some options interpreted by `a0cc` and `a0ld` themselves, not passed to the back-end. They are:

- `-a0comp` *compiler*: defines the back-end compiler to be used.
- `-a0link` *linker*: defines the back-end linker to be used.
- `-a0kernel`: tells the front-end to use the predefined options for compiling the a0kernel library.
- `-a0`: tells the front-end to use the predefined options for compiling the a0 library.
- `-a0format`: tells the front-end to use the predefined options for compiling the a0format library.

To compile and link complex programs, which require several objects or libraries, the user must build a makefile. Suppose you have 3 source files, which must be compiled with different compiler options, and linked together at the end. File `file1.c` needs options `-BLABLA`, file `file2.c` needs `-FOOBAR` and file `file3.c` needs no options. The option `-OPTION` mut be passed to the linker. The lines that follow are a makefile for such 3-file example.

```
include $(A0MAKEFILE)

file1.o: file1.c
$(CC) $(CCFLAGS) -BLABLA -o file1.o file1.c

file2.o: file2.c
$(CC) $(CCFLAGS) -FOOBAR -o file2.o file2.c

# make knows how to make file3.o, since it has no special options

myprog: file1.o file2.o file3.o
$(LD) $(LDFLAGS) -OPTION -o myprog file1.o file2.o file3.o
```

5.6 Executing Athapascan-0 Programs

In the `bin` directory, the user can find `a0run`, which executes an Athapascan-0 program using the configured MPI. The `a0run` command understands the following options:

- `-a0n` *number*: executes the program in a given number of nodes.
- `-a0verbose`: executes the program in the verbose mode. All the commands issued will be echoed on the screen.

For example, to run program `fubar` in three nodes, with parameters `myfile` and `34` passed to the program's `argv[1]` and `argv[2]`, use:

```
prompt% a0run fubar -a0n 3 myfile 34
```

An Athapascan-0 program can be also started as any MPI program in the system. For more information, refer to the MPI manual of your system. On SP2 systems, check `poe`, on LAM/MPI or MPI-CH systems, check `mpirun`.

The Athapascan-0 runtime library extracts itself some of the parameters passed to the executable program. All the parameters not extracted by Athapascan-0 will be passed in the program `main()` function arguments `argv[]`. The currently recognized Athapascan-0 arguments are:

- `-a0stack` *value*: defines the default stack size of the created threads.
- `-a0verbose`: verbose mode, echoes on the screen the nodes startup and termination (`a0run` with `-v` passes `-a0verbose` to the application).
- `-a0account`: accounting mode. Created, blocked, running threads are counted.
- `-a0stop`: stops for debugging. All nodes stop execution at function `a0Init()`. Execution is resumed with `a0cont` command (see next section on debugging).
- `-a0divert`: diverts `stdin`, `stdout` and `stderr` files on each node to the files `stdin-`*N*, `stdout-`*N* and `stderr-`*N*, where *N* is the node number.
- `-a0divertstdin`: diverts `stdin` file on each node to the file `stdin-`*N*, where *N* is the node number.
- `-a0divertstdout`: diverts `stdout` file on each node to the file `stdout-`*N*, where *N* is the node number.
- `-a0divertstderr`: diverts `stderr` file on each node to the file `stderr-`*N*, where *N* is the node number.
- `-a0stdin` *file*: diverts `stdin` file on each node to the file *file-N*, where *N* is the node number.
- `-a0stdout` *file*: diverts `stdout` file on each node to the file *file-N*, where *N* is the node number.
- `-a0stderr` *file*: diverts `stderr` file on each node to the file *file-N*, where *N* is the node number.

5.7 Debugging

Very few systems for parallel and threaded programs debugging exist. Athapascan-0 programs can be degugged with regular debuggers, using a special startup. The debugger to be used should at least be capable to attach to a running process.

To run an Athapascan-0 program with a debugger, start the program with the `-a0stop` option. It will cause the program to stop in the call of `a0Init()` on all nodes. One debugger should be started for every node to be debugged.

Some debuggers allow the user to debug many Unix processes at the same time. If all nodes can be run in the same physical computer, one single debugger can be started. See your MPI manual to learn how to run all nodes in a single computer.

Some MPI implementations run every node in a different physical machine, so, the debugger must be remotely started manually on that machines too. A hostfiles, with a list of machines in which the parallel program must run, can be useful in that moment. See your MPI manual to learn how to use hostfiles.

After starting all the necessary debuggers, attach every one to the Unix processes corresponding to the nodes. Starting Athapascan-0 with the option `-a0verbose`, the physical machines and process identifiers of all nodes are echoed. Having attached all the processes, they will be blocked in the `a0Init()` function call.

To continue the execution, the user must run `a0cont` in the same directory where the node 0 is running. All nodes will return from the call of `a0Init()` and all debuggers can take control of it.

References

[1] Digital E.C. POSIX 1003.4a (pthread) Reference, part of the *Guide to DECthreads*. Digital Equipment Corporation, Maynard (Mass.), March 1993.

[2] MPI Forum. MPI: A Message-Passing Interface Standard. University of Tenessee at Knoxville, June 1995.

[3] Something about some version of Athapascan.

[4] Pasin, M., Ginsburg, I. Athapascan Kernel