Anahy: A Programming Environment for Cluster Computing*

Gerson Geraldo H. Cavalheiro, Luciano Paschoal Gaspary,
Marcelo Augusto Cardozo, and Otávio Corrêa Cordeiro

Universidade do Vale do Rio dos Sinos
Programa Interdisciplinar de Pós-graduação em Computação Aplicada
São Leopoldo – RS – Brazil
{gersonc,paschoal}@unisinos.br, {mcardozo,otaviocc}@turing.unisinos.br

Abstract. This paper presents Anahy, a programming environment for cluster computing. Anahy is presented in terms of its programming interface (API) and its scheduling mechanism. The main features of this environment are the specification of a POSIX thread-based API and the use of dynamic scheduling techniques based on Directed Acyclic Task Graphs (DAG). The main advantage obtained with these features is the dissociation between the description of the concurrency of an application and its parallel execution. The paper examines how Anahy builds a DAG describing the dependencies among tasks at execution time from a multithreaded program and how this DAG is handled by the runtime to apply dynamic scheduling techniques. The paper concludes discussing three case studies of applications developed in the context of Anahy environment.

1 Introduction

New runtime environments have been proposed for cluster computing to assist the development of applications. Some of them are composed by a layered architecture, wherein at the top they propose a high level application programming interface (API) to describe the concurrency of an application as a concurrent program and, at the bottom, a runtime to execute this program. Therefore, an efficient execution depends on a good strategy for scheduling the computational cost generated by the program in execution (computation, data, and communication) over the computational resources available on the hardware (processors, memories, and network). Such scheduling cannot be undertaken in a straightforward manner, since it must consider information related to program behavior.

Overcoming the above difficulty is a challenge that involves both programming model [1] and scheduling. A common approach taken in the development of programming tools and runtime environments (e.g. Athapascan-1 [2], Cilk [3],

* The Anahy project is supported by CNPq/PDPG-TI (55 2196/2002-9), FAPERGS (02/0571.4), UNISINOS (37.00.001/00-0), and was developed in collaboration with HP Brazil R&D.

Pyrros [4], and GrADS [5]) is building an intermediate level between the program in execution and the scheduler describing the program structure in terms of a Directed Acyclic task Graph (DAG). Since the literature on scheduling strategies taking graphs as input is vast (e.g. [4, 6–8]), the interaction between graph and scheduling is well known (e.g. [9]). However, traditional programming tools for cluster computing (such as those based on multithreading and/or message passing) don't offer high level programming resources for creating such graph representation.

This paper addresses the aforementioned problem by proposing Anahy, a programming environment for cluster computing. We present the programming interface (API) proposed for this environment as well as some aspects related to its scheduling mechanism. The main features of this environment are the specification of a POSIX thread-based API and the use of dynamic scheduling techniques based on DAGs. The main advantage obtained with these features is the dissociation between the description of the concurrency of an application and its parallel execution [10]. The paper examines how Anahy builds a DAG describing the dependencies among tasks at execution time from a multithreaded program and how this DAG is handled by the runtime to apply dynamic scheduling techniques.

The remaining of the paper is organized as follows. In the next section, related work is briefly presented. Section 3 presents the Anahy programming interface. Section 4 covers the algorithm employed to schedule concurrent programs. Section 5 presents three case studies, and Section 6 concludes the paper with final remaks and perspectives for future work.

2 Related Work

A DAG is a typical abstraction to model the structure of programs in terms of concurrent activities and data communications [11]. In this abstraction, each concurrent activity defined by the program, named task, is represented by a vertex and a communication between two tasks is represented by an arc connecting two vertices. The use of DAG is very common in static schedulers. Dynamic techniques have been proposed ([12]) in order to avoid inefficiency on blind dynamic scheduling techniques [13] (that is, schedule techniques that don't considering the program structure).

Scheduling DAG is a NP-hard problem [14]. Most of the DAG schedulers are based on list scheduling techniques (e.g. [7] and [6]). Those schedulers handle the tasks generated by the program in priority list. This technique is based on a two step algorithm: in the first step a priority list is built by assigning each task generated by the program a priority; in the second step tasks are mapped to processors respecting their execution priorities.

In 1995 Feitelson has observed in [15] that although lots of researches were being made on DAG scheduling strategies, few efforts were observed in exploiting their use on runtime systems. Nowadays, the research on the area is still popular (e.g. [16–18, 5]). Nevertheless, the number of programming and execution envi-

ronments employing DAG based scheduling is limited, particularly if we consider those that support applications whose DAGs are created at execution time. Cilk [3] and Athapascan-1 [2] are examples of them for SMP and cluster architectures. Both Cilk and Athapascan-1 propose APIs that allow building graph structures at execution time and a runtime able to apply dynamic scheduling techniques based on list strategies.

The Cilk API provides resources for the explicit creation and synchronization of concurrent activities, called threads, and to access a shared memory space. These features allow the programmer to introduce synchronizations among threads in order to control data exchange. The Athapascan-1 API offers special data types in a shared memory space and a primitive to create concurrent activities, called tasks. Tasks are created explicitly but, differently from Cilk, the programmer must identify the input and the output data of each task.

The approach considered by both Cilk and Athapascan-1 takes into account that the scheduler can exploit the structure of the graph *during its construction*. In such way, they can apply a heuristic to explore the program structure in order to achieve an index of performance and avoid inefficiency of blind scheduling techniques. Nevertheless, the graph built in Cilk represents only the precedence among threads, not representing concurrency in a smaller unit such as a task. As a consequence, the Cilk scheduler is able to exploit only serial parallel graphs (nested fork and join operations). On the other hand, the graph built in Athapascan-1 is more complete since it includes the data dependencies among tasks. In this case the scheduler has more information about the program in execution but the cost to build and manage the graph is higher. We propose to mix these two approaches by offering a programming environment able to obtain data dependencies among tasks from a graph describing execution precedence among threads.

3 The Anahy programming interface

The Anahy API offers high level programming resources to handle a large number of concurrent activities and communications in a multithreading style. This API offers a *fork/join*-based model to describe the concurrency in terms of threads. An intermediate level between this API and the runtime is responsible for identifying the concurrency in smaller units, called *tasks*, and creating a DAG representing the data dependencies between tasks.

3.1 Handling tasks with Anahy

The Anahy API provides services to explore a shared memory multiprocessor architecture. These services allow the creation and the synchronization of threads and can be represented by the operations *fork/join/exit*. A *fork* consists in the creation of a new execution flow responsible for executing a function $\mathcal{F}$ defined in the body of the program having a set of data $\mathcal{X}$ as input. The *fork* operator returns an identifier for the newly created thread. Although the thread is

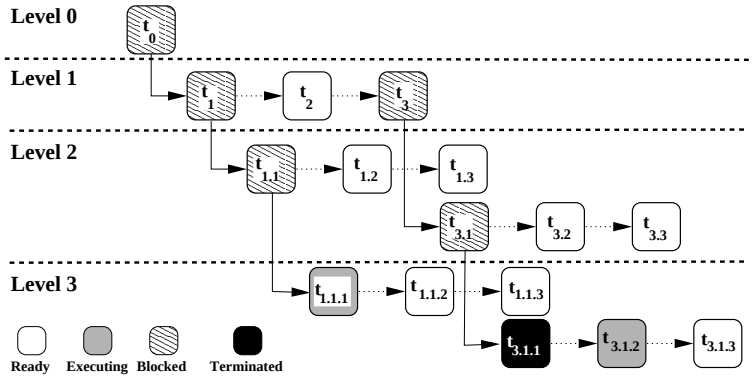

Fig. 1. Graph representing threads creation.

ready to be executed, the programmer cannot predict when this thread will be triggered. The *exit* corresponds to the last operation performed by the thread extinguishing the execution flow. The synchronization upon termination of a thread is performed through a *join*, identifying the flow to be synchronized. This operation allows a thread to be blocked until the termination of another thread, so that it can gather the results $\mathcal{Y}$ produced by $\mathcal{F}(\mathcal{X})$.

Figure 1 shows a snapshot of threads state taken during the execution of an Anahy program. The graph in this figure contains all threads created until the snapshot: notice that they are in different states reflecting their life cycle (executing, ready, blocked, terminated). The threads are grouped in levels, based on their depth in the program: a thread from level i creates threads on level $i+1$. For example, thread t_0 creates threads t_1, t_2 and t_3 (in this order). Arrows in this representation identify the relation of creation; dotted arrows were employed to identify threads on the same level created by the same thread.

3.2 A POSIX-like thread interface

Considering the programming model, both *fork* and *join* operations create new tasks. We have implemented this model in Anahy as a library for C/C++ programs offering a programming interface closer to the POSIX threads standard in order to provide a *multithreading* programming style. Therefore, although *fork* and *join* handle tasks, the API of Anahy offers primitives to create and synchronize (join) Anahy threads.

The body of a thread. The body of a thread is defined as a conventional C function, as follows:

```
void * func( void * in ) {
  /* code */
  return out;
}
```

In this example, `func` corresponds to the function to be executed in a new thread and `in` corresponds to the memory address (in the shared memory) where the input data for the function is located. The return instruction (`return out`) corresponds to the *exit* operation. Notice that when a thread finishes its output is stored in the shared memory at the address specified by `out`.

Synchronization of threads. The `pthread_create` and `pthread_join` services correspond to the creation and join-synchronization of threads in POSIX threads standard. The corresponding syntaxes in Anahy are:

```
int athread_create( athread_t *th, athread_attr_t *attr,
                    void *(*func)(void *), void *in);
int athread_join( athread_t th, void **res);
```

`athread_create` creates a new thread to execute the function defined by `func`; the input data of `func` is stored in the address specified by `in`. The parameter `th` will be updated to get a value to identify the new thread created. The `attr` argument specifies thread attributes to be applied to the new thread (as memory requirements or computational costs). In the operation of `athread_join` the thread on which the synchronization is to be performed is identified by `th` and `res` will be updated to point to a position in the shared memory where the output of the function executed by the thread `th` can be found.

Migration of threads. Although Anahy interface provides a multithreaded programming style, executions can be achieved on distributed memory architectures. Thus, threads can be migrated between nodes. The scheduling mechanism was developed to migrate threads transparently. Nonetheless, the programmer must provide the execution support with information about the data required (parameters) and produced (results) by the threads allowing the data transfers. The mechanism adopted introduces the use of *pack/unpack* functions. The prototypes of pack/unpack functions for a given thread are the following:

```
int packInFunc( void *in, char **buff );
int unpackInFunc( void *in, char **buff );
int packOutFunc( void *res, char **buff );
int unpackOutFunc( void *res, char **buff );
```

The first parameter of each pack/unpack function represents the data to be sent (`in`) or produced (`res`) to/by a thread. The second parameter (`buff`) represents the buffer where the input data for a thread must be *packed* – in thread creation – or from where data must be read to be *unpacked* – in thread launching –. Each function must return the size (in bytes) of data packed/unpacked. The programmer associates specific pack/unpack functions to threads in the thread attributes (`athread_attr_t`):

```
int athread_attr_setpackinput(    athread_attr_t *attr,
                                  int (*func)(void *in, char **buff) );
int athread_attr_setunpackinput(  athread_attr_t *attr,
                                  int (*func)(void *in, char **buff) );
int athread_attr_setpackoutput(   athread_attr_t *attr,
                                  int (*func)(void *res, char **buff) );
int athread_attr_setunpackoutput( athread_attr_t *attr,
                                  int (*func)(void *res, char **buff) );
```

The default value (`NULL`) allows the thread to execute only in the node where it was created.

To illustrate the use of Anahy, the program presented in Figure 2 implements the code able to generate the graph in Figure 1. Due to space limitations the code not related to Anahy and the one describing pack/unpack operations are not presented.

```
void *foo( void *depth ) {
  athread_t child[3];
  int mydepth, *childdepth, *ret, *res = new int(0);
  mydepth = (int *) i*depth;
  if( mydepth > 3 )
     *res = computeSomething( mydepth );
  else {
     *childdepth = new int( mydepth+1 );
     for( int i = 0 ; i < 3 ; i++ )
        athread_create(&child[i], NULL, foo, childdepth);
     for( int i = 0 ; i < 3 ; i++ ) {
         athread_join( child[i], (void **)&ret );
         *res += computeSomething( *ret );
         delete( *ret );
     }
     delete( childdepth );
  }
  return res;
}
int main() {
  int complexity, *result;
  result = foo( (void *) &complexity )
  free(result);
  return 0;
}
```

Fig. 2. An example of Anahy program.

4 The Anahy scheduler

While the API of Anahy provides a multithreaded abstraction to describe the concurrency of applications, the scheduling strategy deals with tasks. The interface between the API and the scheduling builds a DAG considering accesses to the shared memory. These tasks are implicitly defined when the program executes calls to `athread_create` and `athread_join`.

A call to a `athread_create` implies the creation of two tasks: the first one is defined in the context of the new thread spawned. This task has as input data the arguments of the thread itself. The second task is created in the original thread, having as input the data present in the local memory of this thread and the identifier of the new thread created. A call to a `athread_join` implies the creation of one new task: the thread terminates the execution of the current task and creates a new one starting in the instruction that follows (in lexicographical order) the *join*. This new task has as input data the local memory of the current

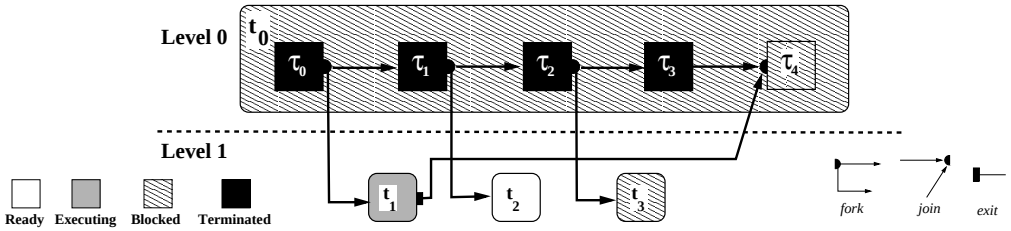

Fig. 3. Zoomed section of the DAG representing the tasks of thread t_0 in Figure 1.

thread and the output produced by the last task executed in the synchronized thread.

The set of tasks created in the context of thread t_0 (Figure 1) is represented in Figure 3. In this figure arrows represent data dependencies among tasks. Down arrows represent *fork* operations while the up arrow represents a *join*.

4.1 Scheduling algorithm

The scheduling algorithm assumes shared memory architecture. A task is defined as the unit of scheduling manipulation as well as is assumed to be executable in a finite time. Tasks finish by executing *fork*, *join* or *exit* operations. The basic algorithm generates three lists: the **ready**, containing the tasks with no restrictions to be launched; the **terminated** list, containing the tasks that have finished; and the **blocked** list containing the tasks waiting for synchronization.

First we introduce the scheduling algorithm considering a mono-processor architecture. The processor is initially idle, then it takes the first task, τ_0, from the list of ready tasks and starts its execution. The instructions of τ_0 are processed sequentially until τ_0 finishes by executing one operation involving the scheduling process (*fork*/*join*/*exit*). The execution of a *fork* produces the creation of two tasks τ_1 and τ_2. Task τ_2 is the one explicitly created by the *fork* and task τ_1 is the one created to be the continuation of τ_0 just after the *fork*. τ_2 is stored in the list of ready tasks while τ_1 is launched. When a *join* is executed, for example $\tau_1.join(\tau_2)$, τ_1 terminates and a new task τ_3 is created: the code of task τ_3 starts at the instruction that follows the *join*; the initial state of τ_3 is blocked. The next scheduling action executes τ_2: the task τ_2 is taken from the ready list of and launched. At the termination of τ_2, τ_3 is unblocked (becomes ready) and started. Notice that τ_3 has as input both the data produced by τ_2 and τ_1. This process can be recursively applied.

In the case of a parallel architecture, there are two or more processors executing the algorithm described above. So, when a task τ_i requests a *join* with τ_j, two new situations may arise: either τ_j has already terminated or τ_j is being executed at that moment. In both cases τ_i finishes and a new task τ_{i+1} is created. In the former case (τ_j has terminated), the procedure consists of recovering the data produced by τ_j, allowing the processor to continue with the execution of τ_{i+1} (τ_j is removed from the list of terminated tasks). In the latter case (τ_j is

being executed), τ_{i+1} remains blocked and the processor looks for a new activity on the list of ready tasks. τ_{i+1} will become ready when τ_j terminates.

The Anahy scheduler was conceived to exploit list scheduling strategies. Thus, its implementation was guided by the existence of a critical path defining the largest sequence of tasks in the program. Considering this critical path, the best performance can be achieved if the scheduler guarantees that during the execution of a program, at least one of the processors is executing a task from this path. Since Anahy focuses dynamic execution of programs, the real critical path is unknown during the execution of the program. Therefore, considering that the concurrent execution of a program must give the same result that a sequential one, the scheduling assumes that the first and the last tasks of the critical path are, respectively, the first and the last task created in the context of t_0 (the first thread launched). The algorithm was implemented in order to guarantee that a processor will be dedicated to execute the tasks of t_0 or the tasks defined in the context of the threads synchronized by t_0. The optimization obtained by the Anahy implementation exploits the recursive nature of the scheduling: while a processor is dedicated to execute the path starting on t_0, a second processor is dedicated to execute the path starting at t_1, another to the path starting at t_2 and so on.

4.2 Multilevel scheduling

To execute an Anahy program, the user must inform a description of the real architecture that will be explored. Like in MPI or PVM, it is necessary to inform the number of nodes of a cluster involved in the execution as well as, for each node, the number of virtual processors (VPs) desired ([19]). The Anahy *virtual machine* is loaded as a runtime kernel when the program is launched on each node. This runtime executes cooperatively and supports the implementation of the scheduling algorithm. This implementation was conceived in three layers. The lowest is handled by the operating system. In this level the VPs are scheduled as system threads over the real processors on each node of the cluster. There is no migration of VPs between nodes.

The second level refers to the allocation of tasks to VPs considering task status (ready, terminated etc.). This level was implemented to consider the locality of tasks. The list of tasks is implemented as a tree where each node represents an Anahy thread (Figure 1) an Anahy thread is a sequence of tasks. This tree has as root the first thread executed by the program, whereas the threads created by the root thread compose the second level. The threads in the second level are the roots of new *sub-trees* of threads and so on – as shown in figures 1 and 3. So, a VP handles only a section of the tree where there are tasks involved in the execution of the current thread. When a VP has no more threads to be executed in its local section, it tries to steal one from a different VP. If so, the VP will choose one thread ready to execute from the highest level of the tree. Such thread is expected to have a larger amount of work than those in lower levels. Another key aspect of this strategy is related to the locality of tasks in-

side threads. Since each thread defines a sequence of tasks, the data transfers between them are accomplished without accesses to the shared memory.

Finally, the third level of scheduling is demanded by the distribution of computational load among the nodes of the cluster. The algorithm is an extension of the second level, taking into account the (communication) costs involved in thread migration between nodes. The implemented load distribution strategy considers the depth of threads in the graph and the size of the data to be sent between nodes. Other factors, including the computational and the physical location of the data, can be added to this basically strategy. Notice that this scheme doesn't consider the migration of running threads.

5 Case Study

In this paper, we discuss the use of Anahy to support the description of applications describing DAGs. A general performance assessment of Anahy can be found in [20] and the performance of a specific application developed in the context of the Anahy project is presented in [21].

To illustrate the use of Anahy we present a synthetic program in Figure 4. This program implements a recursive algorithm able to construct a binary tree structure with a great number of concurrent activities. More details can be found in [20]. The main input of the program is the one defining the number of recursive interactions to be accomplished. Figure 5 presents the graph generated by running this program. In this figure, we also highlight the dependencies between tasks (continuation dependency), between a task and a thread (creation dependency), and between a thread and a task (join dependency). The final structure of the graph (a binary tree) reflects the locality of references of data (inputs and outputs of threads). Those dependencies are exploited at execution time by the Anahy scheduler in order to optimize the execution of tasks in the critical path. Notice that all threads execute the same amount of work.

Figure 5 presents the graph generated by the program in Figure 4 and Figure 6 presents the execution trace of the same program. For the trace, the Anahy runtime was configured with 4 VPs. In the figure, each line segment represents a thread executed by a VP. Each different line style represents a different VP responsible for executing the corresponding thread. Thus, it is possible to observe that the scheduling strategy confer different priorities to threads according to VPs – each VP is give a high priority to execute the tasks in the path . In this figure we have highlighted the threads executed by the VP 1 to exemplify the scheduling behavior.

In the context of the Anahy project we are working on the development of real applications, among them we name a dynamic programming based sequence alignment algorithm and a fluid dynamics simulation. The DAGs for these applications are represented in Figure 7 – circles represent tasks and boxes the data exchanged between them. In [21] it is presented an evaluation of the performance obtained with the dynamic programming application.

```
#include <athread.h>
int main(int argc, char **argv){
    athread_t thr;
    void *dta, *res;
    dta = malloc(...); *dta = foo(...);
    athread_create(&thr, NULL, tree, &dta);
    athread_join(thr, &res);
    free(dta);
    doSomething(*res);
    free(res);
    return 0;
}

void *tree(void *argVoid){
    void *arg0, *arg1, *res, *aux0, *aux1;
    athread_t thr0, thr1;

    if( notFinish(*argVoid) ) {
      arg0 = malloc(...); *arg0 = foo(*argVoid);
      arg1 = malloc(...); *arg1 = bar(*argVoid);
      athread_create(&thr0, NULL, tree, &arg0);
      athread_create(&thr1, NULL, tree, &arg1);
      *res = doSomething(arg0, arg1);
      athread_join(thr0, &aux0);
      athread_join(thr1, &aux1);
      *res += doSomething(*aux0, *aux1);
      free(aux0); free(aux1);
   }
   else res = NULL;

   return res;
}
```

Fig. 4. Synthetic program executing a recursive algorithm.

The dynamic programming application describes a regular DAG (Figure 7.a). In this application, a recursive algorithm fills in a matrix representing the comparison of two sequences. The value of each cell of the matrix corresponds to the similarity between the elements of these sequences. The matrix is filled in from top to bottom and from left to right, with element $M_{i,j}$ requiring three values that were previously calculated according to the concurrency relation: $M(i,j) = \mathcal{F}(M_{i-1,j-1}, M_{i-1,j}, M_{i,j-1})$. As shown in Figure 7.a, data locality can be predicted by the scheduler considering the regular structure of communications.

On other hand, fluid dynamics simulation is an irregular application, since it presents an unpredictable program structure (Figure 7.b). The proposed implementation divides the physical space into triangles. A thread is generated for

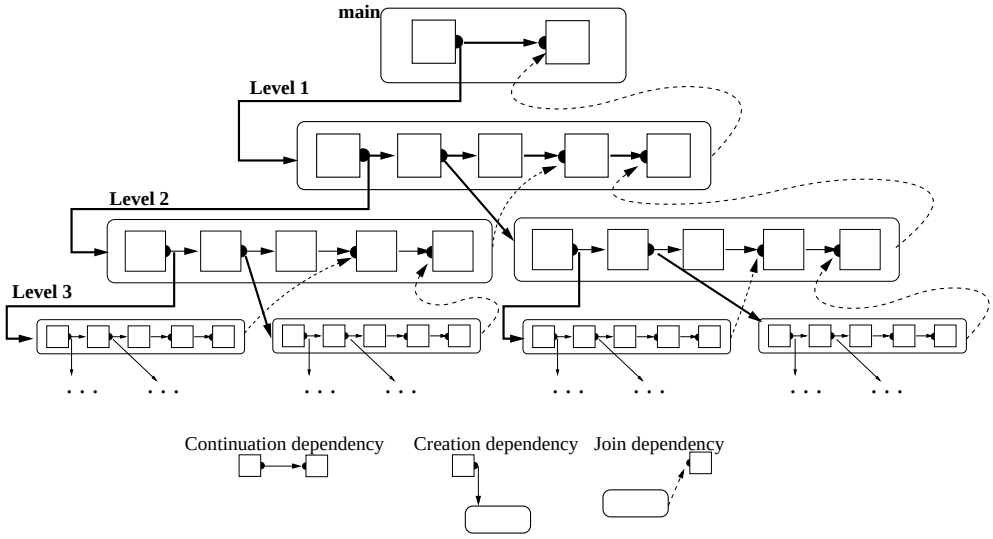

Fig. 5. DAG generated by the Anahy runtime for the recursive program presented in Figure 4.

each triangle to compute the fluid velocity using Euler equation. Once a thread finishes computing the equation, new threads can be generated to give sequence to the simulation. Although the DAG is irregular, the scheduler can apply a load balancing strategy considering the depth of threads in the graph: the closer to the top a thread is, the higher is the probability of this thread accumulating a large amount of work.

6 Conclusion

This paper presented Anahy, an environment for exploring high performance processing in cluster architectures. Anahy was presented in terms of its API and the principles adopted for introducing an intermediate level responsible for building a DAG at execution time. This DAG is exploited by the Anahy runtime to avoid inefficacity of blind dynamic scheduling strategies in the execution of tasks. Another key contribution of this work is the adoption of an API based on the POSIX threads standard allowing the development of programs to distributed memory architectures without dealing with issues related to message exchange mechanisms.

The next steps of this work include the development of load balancing strategies and the extension of the API to include all POSIX-defined synchronization mechanisms for thread execution control (as critical sections and condition variables). Even though the use of such mechanisms is not recommended in the Anahy programming model, due to potential performance loss, they will be included to increase compatibility with legacy code.

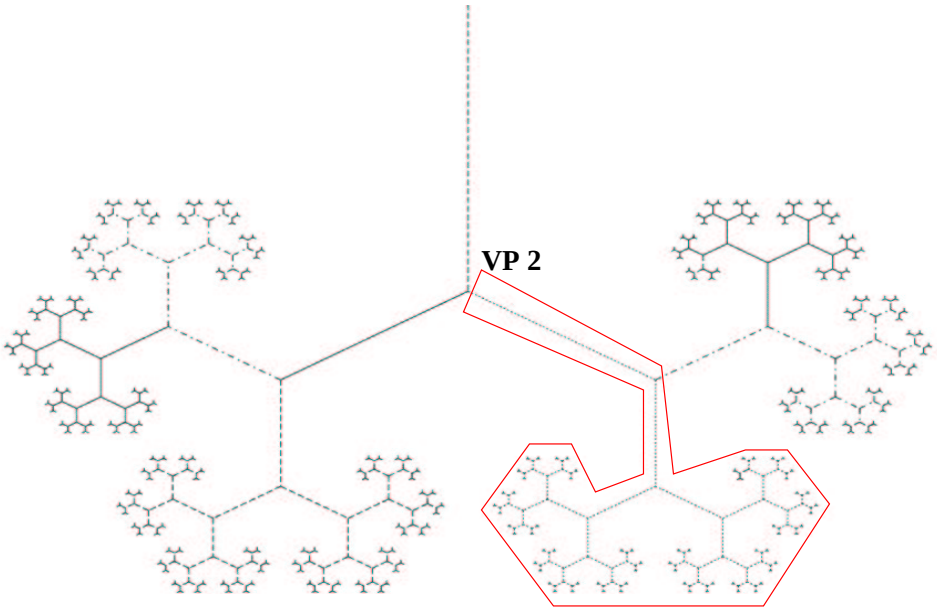

Fig. 6. Trace representing the execution of the DAG presented in Figure 5.

References

1. Alverson, G.A., Griswold, W., Lin, C., Snyder, L.: Abstractions for portable, scalable parallel programming. IEEE Trans. on Parallel and Distributed Systems **9**(1) (1998) 71–86
2. Galilée, F., Cavalheiro, G.G.H., Roch, J.L., Doreille, M.: Athapascan-1: on-line building data flow graph in a parallel language. In: PACT'98, Paris (1998)
3. Blumofe, R., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., K. H. Randall, Y.Z.: Cilk: an efficient multithreaded runtime system. Journal of Parallel and Distributed Computing **37**(1) (1996) 55–69
4. Yang, T., Gerasoulis, A.: DSC: Scheduling parallel tasks on an unbounded number of processors. IEEE Transactions on Parallel and Distributed Systems **5**(9) (1994) 283–297
5. Berman, F., Casanova, H., Chien, A., Cooper, K., Dail, H., Dasgupta, A., Deng, W., Dongarra, J., Johnsson, L., Kennedy, K., Koelbel, C., Liu, B., Liu, X., Mandal, A., Marin, G., Mazina, M., Mellor-Crummey, J., Mendes, C., Olugbile, A., Patel, M., Reed, D., Shi, Z., Sievert, O., Xia, H., YarKhan, A.: New grid scheduling and rescheduling methods in the grads project. International Journal of Parallel Programming **33**(2–3) (2005) 209–229
6. Coffman, E., Graham, R.: Optimal scheduling for two-processor systems. Acta Informatica **1** (1972) 200–213
7. Hu, T.: Parallel sequencing and assembly line problems. Operations Research **19**(6) (1961) 841–848
8. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. Parallel and Distributed Computing **59**(3) (1999) 381–422

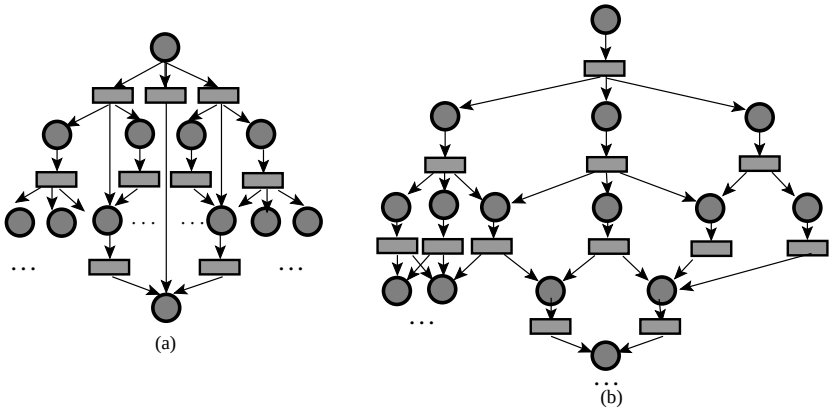

Fig. 7. Sections of DAGs generated by a regular (a) and an irregular (b) application.

9. Cavalheiro, G.: A general scheduling framework for parallel execution environments. In: Proc. of the SLAB'01, Brisbane (2001)
10. Black, D.L.: Scheduling support for concurrency and parallelism in the mach operating system. IEEE Computer **23**(5) (1990) 35–43
11. Xiao, Z., Li, W., Jenq, J.: On unit task linear-nonlinear two-cluster scheduling problem. In: Proc. of the SAC'05, Santa Fe (2005)
12. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. **31**(4) (1999) 406–471
13. Culler, D., Arvind: Resource requirements of dataflow programs, Honolulu (1988)
14. Garey, M., Johnson, D.: Computers and intractability: a guide to the theory of NP-Completeness. (1979)
15. Feitelson, D., Rudolph, L.: Parallel job scheduling: issues and approaches. In Feitelson, D., Rudolph, L., eds.: Proc. of the IPPS'95. Volume 949., Springer (1995) 1–18
16. Iverson, M.A., Özgüner, F.: Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In: Heterogeneous Computing Workshop. (1998)
17. Sinnen, O., Sousa, L.: List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Computing (V. 30:1. 2004)
18. Sakellariou, R., Zhao, H.: A hybrid heuristic for DAG scheduling on heterogeneous systems. Proc. of the Heterogeneous Computing Workshop (2004)
19. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM (V. 33:8. 1990)
20. Cordeiro, O., Peranconi, D., Villa Real, L., Dall'Agnol, E., Cavalheiro, G.: Exploiting multithreaded programming on cluster architectures. In: HPCS 2005, Guelph (2005)
21. Peranconi, D.S., Cavalheiro, G.G.H.: Using Active Messages to explore high performance in cluster of computers. In: SCCC 2005. (2005)