Aproximating Static List Schedules in Dynamic Multithreaded Applications

Ccero A. S. Camargo · **Gerson G. H. Cavalheiro** · **Maurcio L. Pilla** ·
Simone A. da Costa · **Luciana Foss**

Abstract List scheduling algorithms are known to be efficient when the application to be executed can be described statically as a directed acyclic graph (DAG) of tasks. Regardless of knowing the entire DAG beforehand, obtaining an optimal scheduling in a parallel machine is a NP-hard problem. Moreover, many programming tools propose the use of scheduling techniques based on list strategies. This paper presents an analysis of scheduling algorithms for multithread programs in a dynamic scenario where threads are created and destroyed during execution. We introduce an algorithm to convert DAGs describing applications as tasks into directed cyclic graphs (DCGs) describing the same application designed in a multithreaded programming interface. Our algorithm cover case studies described in previous works, successfully mapping from the abstract level of graphs to the application environment. These mappings preserve the guarantees offered by the abstract model, providing efficient scheduling of dynamic programs that follow the intended multithread model. We conclude the paper presenting some performance results we obtained by list schedulers in dynamic multithreaded environments and we compare these results with the best scheduling we could obtain with similar static schedulers.

Federal University of Pelotas (Brazil)
Gomes Carneiro, 1
Tel.: +55 53 3921-1327
E-mail: {cadscamargo,gerson.cavalheiro}@inf.ufpel.edu.br
{pilla,simone.costa,lfoss}@inf.ufpel.edu.br

1 Introduction

With the increasing popularization of multicore processors, parallel programming has almost become obligatory in the development of any kind of software. Parallel computers, that were once found only in academic or high-performance computing environments, are now ubiquitous. The rise on the demand for programmers that are capable of understanding the issues of these architectures is a direct consequence of this shift from sequential to parallel programming. However, migration to parallel programming paradigms is not simple, as they require a larger abstraction effort. The predicted increase on the number of cores to tens or hundreds of cores per chip will further increase programming complexity.

One way to decrease programming complexity is by providing an efficient scheduling at runtime, so that the programmer just needs to describe the concurrency of the application regardless of the underlying hardware. Nevertheless, obtaining optimals schedules is a NP-hard problem [7], and such class of scheduling strategies can be applied only to static problems – where the structure of the parallel program, as well as the parallel hardware, are known *a priori*. Even considering these restrictions, the abstraction of scheduling is successfully employed by many comercial programming tools such as OpenMP [10] and Intel® Threading Building Blocks [20], and many academic and research programming tools, such as Cilk [5] and Athreads [8].

In this paper we evaluate the performance of greedy scheduling techniques [17,13] as the runtime support

for dynamic multithreaded programs. Our focus is a scheduling technique that employs a heuristic to determine which thread in a multithreaded program is executing tasks in the *critical path*. This approach has been explored by different academic and research high performance computing runtime environments, such as Cilk [5], and Anahy [8]. Although case studies and empyric evaluation of these environments have shown satisfactory performance, there are no works that compare the performance of static and dynamic schedulings for the same applications.

This work recovers results presented by Graham [15], then map the case studies to an abstraction of program developed in a multithread programming interface, comparing Graham's static schedules with a dynamic schedule for the mutithread abstraction. The main issue that arises is how to compare the schedule of an application described as a DAG of tasks, the required input for list algorithms, against the schedule of the same application described in a cyclic graph of threads (DCG), this later schedule being obtained by the execution of the DCG in a multithreaded dynamic environment.

In this work, Anahy is the chosen multithread programming interface. Thus we present a set of patterns that allow us to transform a DAG abstraction into a graph representing the control flow of an Anahy program. The we present a dynamic scheduling strategy based on List algorithm to schedule a multithreaded program on a hypothetical architecture, and we finally show the performance obtained by applying this strategy to multithread programs. We can then compare, for a given application, the performance obtained by our dynamic scheduling with the performance resulted by applying static strategies. The results indicate that, for the case studies evaluated, a greedy heuristic can be as good in a multithread dynamic environment as List algorithm can in a static scenario for a given application. However, the programmer should be aware of the environment's scheduling strategy and manage the dependencies between threads to take advantage of this.

This work is an extension of the one published in [6]. In this extension we discuss the approximation of static list scheduling algorithms in dynamic multithreaded environments and then present scheduling results obtained in several simulations as well as executions in real multithreaded environments.

The remainder of the paper is organized as follows. List scheduling and critical path heuristic are described in Section 2. Section 3 presents a model for a multithreaded programming interface. The process of a creation of a DCG from a DAG is described in Section 4,

and in Section 5 performance of static and dynamic scheduling are compared. Section 6 presents case studies of applying list scheduling strategies in dynamic multithreaded environments, where one of these cases shows actual execution in real multithreaded environments. Finally, Section 7 presents the concluding remarks.

2 List Scheduling Algorithms

List scheduling algorithms receive a Directed Acyclic Graph (DAG) as input and produce a schedule over m available processors. The input DAG describes a parallel program in terms of tasks (represented by nodes) and their dependencies (represented by directed edges). As an online technique, list scheduling algorithms do not analyze the nodes that follow a task to decide the current scheduling, since these tasks could not be present in the DAG at runtime. Furthermore, heuristics can be applied to determine priority rules for the tasks allowing the scheduler to select the most suitable one to optimize some performance index.

The following subsections define the basic structure of dataflow programs, what is a direct acyclic graph, and the basic greedy scheduling algorithm.

2.1 A Dataflow Program

Programs describing their tasks in terms of data dependences are suitable to be described in a DAG. In the dataflow model [18], a task consists of a sequence of instructions between two synchronization points: the input and the output of data. Any other synchronization mechanism is not allowed. Tasks can be represented as functions $y = f(x)$, x and y representing the input and output set of data. A given task is said to be ready to execute when its input operands become available. When f is finished, its output values may then be taken as input by other tasks.

This execution model is very attractive for several reasons [2], such as: *(i)* tasks use data items defined in a higher level of abstraction without worrying about how they are produced, and *(ii)* programs are highly extensible and reusable because there is no direct task-to-task coupling. Another aspect is its original research motivation in the context of the exploitation of massive parallelism [18].

2.2 Directed Acyclic Graph

A *Directed Acyclic Graph* (DAG) is a representation of an application in terms of tasks and dependencies

between tasks. The nodes in the graph represent the tasks and the directed edges represent the input/output data of these tasks. Nodes may be weighted with their computational costs, while edges may be weighted with communication costs. A more formal description of a DAG follows:

Definition 1 A DAG is a tuple $G = (V, A)$ representing a program P. In this tuple, V corresponds to the set n of nodes and A to the set of edges in G. Each node in V receives a label i, where $1 \leq i \leq n$, in order to represent the task τ_i in the graph: $V = \{\tau_1, \tau_2, \tau_n\}$. A precedence relation between τ_i and τ_j is represented by $\tau_i \prec \tau_j$ and defines a direct edge in the graph: $A = \{(\tau_i \prec \tau_j)|1 \leq i, j \leq n, i \neq j\}$. If the program P describes that τ_i is an immediate predecessor of τ_j, then $(\tau_i \prec \tau_j) \in A$. Such dependence means that τ_j cannot be started before τ_i has been finished, because the output of τ_i must be available to τ_j so it can be started. In this representation, nodes and edges are weighted: $|\tau_i|$ and $|\tau_i \prec \tau_j|$ correspond to the computational cost associated to τ_i and to the data transfer cost associated to $\tau_i \prec \tau_j$, respectively.

From the previous definition we assume that G can have transitive edges: if τ_i, τ_j and τ_k are tasks defined by the application and $\tau_i \prec \tau_j$, $\tau_i \prec \tau_k$ and $\tau_j \prec \tau_k$ are precedence constraints in this application, $(\tau_i, \tau_j, \tau_k) \in V$ as well as $(\tau_i \prec \tau_j)$, $(\tau_i \prec \tau_k)$, $(\tau_j \prec \tau_k) \in A$.

The input tasks and the output tasks can also be identified, that is, the tasks that are ready to execute when the program is launched and the tasks that do not produce data to any other task in the program. Input tasks are the subset V^{in} of tasks τ_i whereas there is not $(\tau_j \prec \tau_i) \in A$. Output tasks are the subset V^{out} of tasks τ_i whereas there is not $(\tau_i \prec \tau_j) \in A$.

2.3 Priority Attributes

Whenever two or more executable tasks contend for a free processor, the selection of the right task is done by their priority of execution, which is set according to some strategy. If there is more than one task ready with the same piority, ties can be broken randomly.

Considering the length of a directed path in the graph as the sum of the weights of all vertices and edges along the path including the initial and final vertices, we can define two major priority attributes found in the literature: the *level* (or *bottom-level*), and the *co-level* (or *top-level*). Both are defined in the same sense as in [11]. The *level* of an output task τ_i in G is $|\tau_i|$, and the *level* of a non-output task τ_i is the longest path from τ_i to an output task. The *co-level* is defined in the

same way, but the references are the input tasks. Thus the *co-level* of an input task τ_i in G is $|\tau_i|$, and the *co-level* of a non-input task τ_i is the longest path from $|\tau_i|$ to an input task.

Several List Scheduling Algorithms like the following are based on These two attributes.

- HLFET (Highest Levels First with Estimated Times): A-Schedules [11] are list schedules which consider the priority of a task as being its *level*;
- HLFNET (Highest Levels First with No Estimated Times): This algorithm considers (possibly incorrectly) that all tasks have the same execution time and their priority attibute is their *level*. HLFNET can be used when HFLET is desirable but we can't know tasks execution times in advance.
- RANDOM: Task priority values are assigned randomly. Random and optimal schedules are common bases to compare other list schedules.
- SCFET (Smallest Co-levels First with Estimated Times): This algorithm considers the priority of a task as the negative of its *co-level*.
- SCFNET (Smallest Co-levels First with Estimated Times): This algorithm works the same as SCFET, but all tasks are assumed to have the same execution time.

There are applications where the DAG is built dynamically as the program evolves. In these cases we can not use an algorithm like HLFET/HLFNET which needs the entire graph structure, from input to output vertices, to calculate the priority attribute for a task. However, algorithms like RANDOM and SCFET/SCFNET can be used for situations when certain tasks can be inserted in the graph asynchronously or one of its predecessors decides to create it. We can also use the arrival time of tasks to assign priorities during execution time. Two major algorithms are built over this idea: LIFO (Last In, First Out) gives higher priority to the most recently created task; FIFO (First In, First Out) gives higher priority to oldest ready task.

2.4 A Greedy Scheduling Strategy

The purpose of a scheduling S is to complete the execution of a given DAG G in a finite time. Scheduling algorithms usualy include strategies to minimize an index of performance, such as the total execution time, as considered in this paper. We note $|S(G)|$ the length of the scheduling, that is, the time required to complete the execution of all tasks in G. The basic list scheduling algorithm runs as follows.

- Input: graph G and a set of m identical processors.
- Output: $|S(G)|$.

1. Set all input tasks in G as $READY$ and the remaining as NOT_READY
2. $L = V$
3. Sort(L)
4. **FOR EACH** processor m **DO**
   (a) $\tau = $ GetReadyTask(L)
   (b) **IF** τ is **NOT** $NULL$
       i. Run τ
       ii. Set τ as $FINISHED$
       iii. Update τ's successors
   (c) **END IF**
5. **UNTIL** τ is $NULL$

This basic strategy makes several assumptions: *(i)* a set of m processors is responsible for executing the *for each* loop concurrently, running the tasks described in G; *ii* Sort(L) is an algorithm like the ones described in Section 2.3, that sorts the list of vertices L in decreasing order of priority; *(iii)* GetReadyTask(L) always returns a valid task, unless the graph's computation is completed; if not NULL, the task's state is atomically set to RUNNING before the function returns; if there are not $READY$ tasks at the moment, the function blocks the applying processor until some task becomes READY; *(iv)* lines labeled as *ii* and *iii* are executed atomically.

This basic strategy was studied in several papers. Graham [16] defines the concept of Critical Path to give the priority to each task and analyzes the worst-case performance. A Critical Path CP is a subset of V that contains the tasks present in the longest sequence of dependences. The concept of a critical path allows the estimation of upper and lower bounds for the time T_m required to schedule a graph G in a machine with m identical processors. Considering that there are no scheduling and communication costs, the constraints for time T_m are defined as [16]:

$$\frac{T_1}{m} \leq T_m \leq \frac{T_1}{m} + T_\infty \tag{1}$$

where:

- $T_1 = \sum_{i=0}^{n} |\tau_i|, \forall \tau_i \in V$, that is, the sum of all computational costs of tasks in G.
- $T_\infty = \sum_{i=0}^{k} |\tau_i|, \forall \tau_i \in CP$, that is, the sum of all computational costs of tasks present in the Critical Path of G.

Since the tasks in a Critical Path must be executed in sequence, the sum of the computational cost in this path represents the minimum execution time expected for the application, independent from the number of available processors. Graham [15] presents several case studies involving the scheduling of DAGs. Further works like [3,19,1,14] extend the results of Graham's.

3 Multithreaded Programming Model

The multithreaded programming model is employed to write parallel programs in terms of concurrent execu-

tion flows sharing a common memory address space. The interface facilities of this program must include primitives to create and synchronize execution flows and to guarantee safe accesses to shared data. We discuss in this section an abstraction of this model to create a Directed Cyclic Graph to represent the control flow among threads.

3.1 Evolving a multithread program

In a multithreaded program, each thread is responsible for executing a specific sequence of instructions. The final result is an arrangement of the partial results computed by each thread. We assume that threads can be dynamically created and destroyed during the execution of a program by invoking a specific service.

Communication among threads may be implemented in two ways, both of them considering that the data itself is present in a shared addressing space. The first method explores the explicit synchronization introduced by invocations to create and join primitives. By monitoring calls to those primitives it is possible to guarantee the control of the execution flow in order to establish a deterministic behavior for the program. A second strategy is to employ semaphores, or other equivalent constructors, to control the execution of threads in a mutual exclusion mode when accessing data in the shared addressing space. A drawback of this strategy is that it is impossible to guarantee an execution order among threads and the execution will be, necessarily, nondeterministic.

3.2 Directed Cyclic Graph

In the present work, a restriction to the communication among threads is imposed: threads may not employ mutual exclusion synchronization mechanisms. Our programming model allows communication among threads at creation or termination time. Hence, these restricted threads become similar to the definition of a task: a thread can be represented as a function $y = F(x)$, since there is no collateral effects in global data. In the same way as in the task model, when F is finished its output values can be received by other threads but differently of the task model, a thread is said to be ready to execute just after its creation, since its input parameters are received during the creation process. Another important difference from tasks is that the code of a thread may include instructions to fork (*create*) new threads or to wait for the end of other threads (*join*). Therefore, a graph generated by a multithreaded execution is

said to be directed with cycles (DCG *Directed Cyclic Graph*).

Definition 2 A DCG is a tuple $C = (N, E)$ representing a program P. In this tuple, N corresponds to the set n of nodes and E to the set of edges in C. Each node in N receives a label i, where $1 \leq i \leq n$, in order to represent the thread Γ_i in the graph: $N = \{\Gamma_1, \Gamma_2, \Gamma_n\}$. A precedence order among instructions of two threads Γ_i and Γ_j is represented by $\Gamma_i \rightarrow \Gamma_j$, in the graph: $E = \{\Gamma_i \rightarrow \Gamma_j | 1 \leq i, j \leq n, i \neq j\}$. There is a directed edge from Γ_i to Γ_j if the program P describes that Γ_i creates Γ_j or that Γ_j synchronizes with the termination of Γ_i. That means that or Γ_j can be started just after it has been created by Γ_i either that Γ_j must remains blocked until to be guaranteed that Γ_i has finished. In this representation nodes and edges are weighted: $|\Gamma_i|$ and $|\Gamma_i \rightarrow \Gamma_j|$ correspond to the computational cost associated to Γ_i and to the data transfer cost associated to $\Gamma_i \rightarrow \Gamma_j$, respectively.

From the definition of a DCG it is allowed to have both $\Gamma_i \rightarrow \Gamma_j$ and $\Gamma_i \rightarrow \Gamma_k$, represented by $\Gamma_i \rightarrow \Gamma_{j,k}$ as well as $\Gamma_i \rightarrow \Gamma_j$ and $\Gamma_k \rightarrow \Gamma_j$, represented by $\Gamma_{i,k} \rightarrow \Gamma_j$. The first case ($\Gamma_i \rightarrow \Gamma_{j,k}$) represents a split (multiple thread creation) operation and the second one a reduce (merge of multiple threads) operation.

Another characteristic inherent to the practice of multithread program is that the execution begins by the execution of one thread (the thread Γ_1) and finishes executing only one thread (the same Γ_1).

3.3 Scheduling threads at application level

There are many strategies to schedule threads at user level, such as the ones implemented in OpenMP [10], Athreads [9], and Cilk [5].

In OpenMP user level threads are created in a nested fork-join structure. The runtime scheduler distributes the work generated among the processors. Three strategies are available to distribute the threads: *cyclic*, *dynamic*, and *guided*. The first one sends equal amounts of work to each processor. The *dynamic* and *guided* strategies can be used to adjust the load distribution in function of the difference of loads among the threads. Such a programming interface is very attractive when the application may be described in terms of data parallelism.

The Cilk programming interface provides two synchronization primitives: *spawn* to create a thread and *sync* to synchronize at once all threads created by the current thread in a nested fork-join structure. There is no explicit data exchage among threads, those synchronization primitives indicate that a set of data are ready to be read from a shared addressing space. The Cilk runtime builds a graph describing the dependences among threads. This graph is handled by the scheduler in order to distribute the load among the processors and to control the execution order. The scheduler algorithm is implemented in two layers known as *work stealing* [4]. At local level, running on each processor, the deeper a thread is in the subsection of the graph handled by the processor, the higher is its priority. At global level, threads can migrate from a processor to another when a processor becomes idle and choses a victim processor to steal work from. In this case, the higher a thread is in the graph, the higher is its priority to be stolen.

Athreads is an implementation of Anahy [8] model. Its programming interface proposes two primitives to handle threads and to communicate data among them: *create* and *join*. These primitives allow the creation or synchronization of $n \geq 1$ threads at once, like split or reduce operations – the runtime implements a kind of garbage collector to check references to finished threads. Addresses of data in a shared dynamic heap are passed to threads at creation time, and also as return values when a thread terminates. Athreads' runtime also generates a graph representing the control flow in reaction to the execution of create/join services but the generated graph is unstructured. From this graph, it is possible to extract a tree representing the creation of threads: each thread represents the root of a subtree, and its children are all threads it has created. When a processor becomes idle, the higher branch of the tree, i. e., the thread containing the oldest ready task not yet executed, is selected by the scheduler. Locally to the processor, the closer a thread is to the root of the subtree, the higher is the priority.

4 Building a DCG from a DAG

List Scheduling strategies are the basis of both Cilk and Athread runtime. More specifically, a List strategy which takes into account the Critical Path. In both environments the heuristic applied is that the Critical Path starts and finishes in the first and last tasks of the first launched thread. Both environments handle threads at application level, and empirical results [5, 8] show that they achieve good performance for some kinds of applications. In the present work we are interested in evaluating how far the performance obtained by dynamic list schedules that take a DCG as input is from static DAG schedules.

4.1 Preliminaries and methodology

The first study of List Scheduling was presented in 1966 by Graham [16]. He has shown that the online greedy algorithm can change the schedule length by a factor of at most $2 - \frac{1}{m}$ depending on the order of task arrival. The model introduced by Graham considers that the scheduling does not incurs in any overhead and that there are no communication costs. In [15], Graham presented a sequence of examples of DAGs being scheduled by List Scheduling, and showed several schedules for them, including the best possible ones. This basic strategy was successfully employed by different multithreaded programming environments, such as Cilk [5] and Athreads [12], although these programming tools employ DCGs instead of DAGs to represent the control flow.

In this paper, we propose the application of list scheduling to a multithreaded program during runtime. For each DAG presented in [15], an equivalent transformation to a DCG is proposed, and the result is submitted to a list scheduling strategy to obtain the scheduling length. Athreads was chosen as the programming interface since it provides resources to build an unstructured graph and both the creation and the synchronizations of multiples threads at once.

The resulting DCGs highlight the control flow among threads but, internally to threads, the tasks of the original DAGs are still represented as well as the original dependencies are still respected.

In the following of this section, we present the translations rules applied to build a DCG from a DAG. Next section presents the case studies and the performance we have obtained.

4.2 Building Rules

The creation of a DCG $C_i = (N_i, E_i)$ representing a DAG $G_i = (V_i, A_i)$ preserves the original structure of G_i. The creation process consists in encapsulating a sequence of $n \leq 1$ tasks of the original G_i in the context of C_i threads. The directed edges between tasks in G_i are preserved in C_i, but only the edges representing dependencies between tasks in different threads are included in E_i. Those edges represent the creation or the synchronization of threads.

From here on we are going to use extensivelly the term "left most", which reflects the way we organize our drawings. By "left most" we mean either the task τ_i with the smallest identifier i, when we refer to tasks at the same level in the graph, or threads that are closer to Γ_i and were created first in a dynamic multithreaded execution.

In Fig. 1 we present a set of nine patterns of relationships among tasks present in a DAG G_i and the rule to be applied to introduce the corresponding section in the DCG C_i. In this figure, circles represent tasks in V_i, and the dotted rectangles threads in N_i. On each pattern, the grey task is the one being evaluated. A hachured task has at least one left most dependence that was already evaluated. An entirely evaluated task is represented in black. Dotted circles are tasks called *conforming tasks* (CT). Those tasks are introduced to guarantee that the tasks present in C_i will finish executing just one synchronization in order to correspond to the task model adopted by Athreads. In this work we consider $|\tau^{CT}| = 0$. A cloud with a + signal involving a thread or a task indicates the presence of at least one of those thread or task. A symbol ? indicates that the element associated can be or not present and the symbols ˆ and $ indicate the begin and the end of G.

4.2.1 Preprocessing G_i

The creation process starts by applying the preprocessing patterns. This first pattern, named `Begin`, identifies the tasks in G_i that have no precedence constraints. A conforming task is created to represent the begining of execution and a multiple thread creation. The result of the application of this pattern is that each task with no precedence constraints will represent the first task to be executed in the context of a thread in C_i. If there is just one task without precedence in V_i, just one thread is added to N_i. Notice that the conforming task is introduced in the left most thread since the first thread launched by the program (that is, the *main* thread) will be represented in the left most side of C_i.

The second pattern, named `End`, identifies tasks without successors in G_i. Similarly to the `Begin`, all tasks become the last task to be executed in the context of a thread and a conforming task is added to the leftmost thread to represent the multiple join. In the case were there is just one exit task in G_i, just one thread is added to C_i.

The last preprocessing pattern is the `Bunch`, which represents that $(\tau_j \prec \tau_k) \in A_i$ but neither $(\tau_x \prec \tau_k), \forall x \neq j \in A_i$ or $(\tau_j \prec \tau_x), \forall x \neq k \in A_i$. The resulting $\tau_{j,k}$ cost is $|\tau_{j,k}| = |\tau_j| + |\tau_k|$.

4.2.2 Iterative processing

After the preparation of G_i, all task in V_i must be visited to identify matching patterns. A task can match one of the following patterns: `Create`, `Sequence`, `Join`, `Spawn`, or `Broadcast`. The tasks in G_i are visited in a breadth-first strategy, taken each level from left to
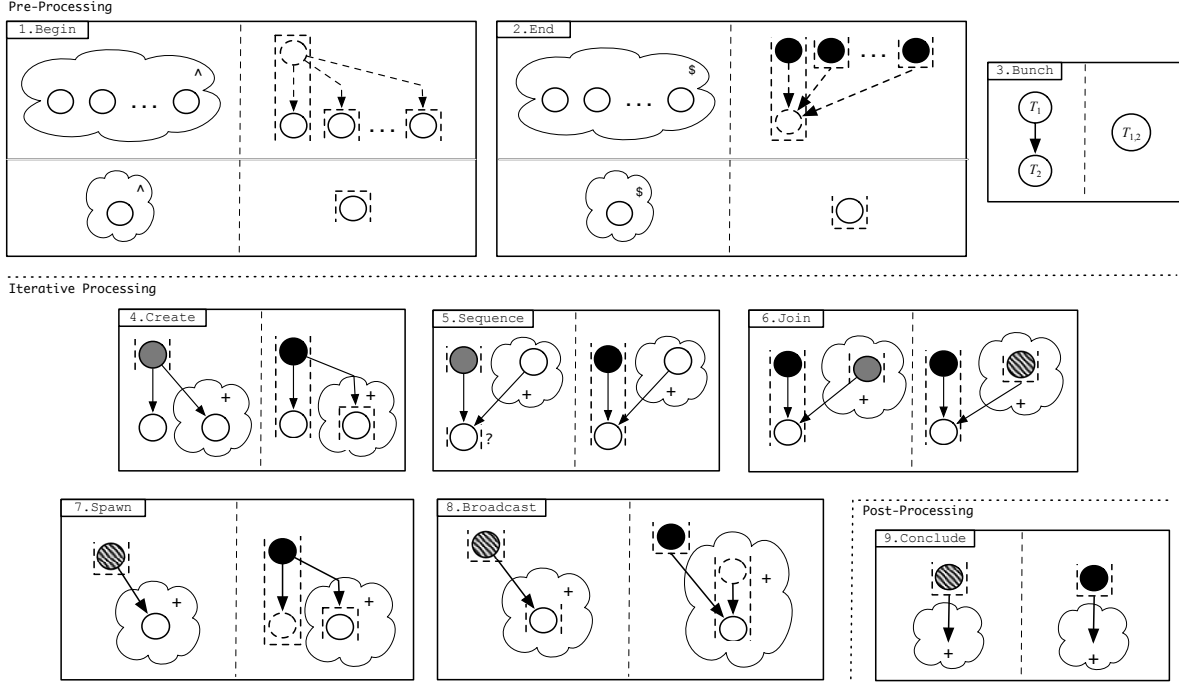
Fig. 1 Patterns in a DAG and the corresponding segment in a DCG.

right. If a task matches more than one pattern, the leftmost dependence must be solved first.

The `Create` and `Join` patterns correspond to the creation or synchronization of threads. At creation, new threads are added to C_i and the task which succeeds the evaluated task becomes part of the same thread. The `Join` pattern identifies the termination of threads and the respective synchronization. As result, the thread containing the evaluated task is closed. The `Sequence` pattern enrolls a task in the context of an existing thread or merges two segments of threads.

Despite the end of a thread being identified, the last task executed by this thread may have other dependences `Spawn` and/or `Broadcast`. A spawn corresponds to creation of new threads; a conforming task is necessary to express the already evaluated dependences. A broadcast corresponds to the synchronization of the end of the current thread with others. In this case, two conforming tasks must be inserted in order to guarantee the existing precedence constraints.

4.2.3 Post-processing C_i

Once all tasks on G_i have been evaluated and introduced in the context of threads on C_i, there are no more dependencies to be evaluated. The pattern `Conclude` identifies the tasks which have matched the pattern `Join` without being followed by any other dependence.

Since the nodes of the graph are visited in a breadth-first strategy and that left dependencies are evaluated first, a larger amount of work is expected to be present in the left side of the DCG. A scheduling heuristic can be applied to explore the presence of the Critical Path in the left most threads.

5 Appling List Scheduling to DCGs

In this section we evaluate the use of a simple scheduling algorithm based in a List Strategy introduced in the runtime of a multithread environment. We present this algorithm and we compare its performance with one presented by the original Graham scheduling.

5.1 Basic List Scheduling Strategy

The proposed scheduling strategy takes as input a DCG $C = (N, E)$ to be executed in a parallel architecture $P = (P_1, P_2, \ldots P_m)$ with m homogeneous processors. The graph is built at execution time: a thread creation introduces new nodes in the DCG and new edges from the node hosting the current thread to the new ones as well as a join adds a new edge from the node(s) representing the thread(s) to be synchronized with the node hosting the thread asking for synchronization. The scheduler can mark nodes in C to identify the current status of the thread.

The scheduler runs as follows in a parallel machine with m identical processors. The first thread on the application (Γ_1) becomes the first node in of the DCG and is annotated as *ready*. In this graph, closer a *ready* thread is from the root node, higher is its priority to be chosen when a processor becomes idle. When a processor P_i becomes idle it takes from the graph a node and marks the corresponding thread as *executing*. Once a thread is finished, the corresponding node is marked as *terminated* and the processor becomes idle. During its execution, a thread may request join operation. In this case, the synchronized threads can be marked as *terminate*, *ready* or *executing*. In the first situation, the current thread can continue its execution. If the synchronized thread is *ready*, the scheduler suspends the execution of the current thread and launch over this processor the synchronized thread. In the last situation, the synchronized thread is *executing*, the scheduler also suspends the execution of the current thread and chose another one to be launched in the portion of the graph having the synchronized thread as root; this processes is repeated as many time as necessary. The execution of a thread is resumed when its synchronization requirement is satisfied.

5.2 Building DCGs

Our case study are the examples 1, 2, 4, 5, 6, 7, 9 and 10 presented in [15]. We name those graphs as $G_1, G_2, \ldots G_8$ and we present in Fig. 2 the corresponding C_i, obtained by applying the proposed process. The number at the interior of each task represents its computational cost this cost can be presented in terms of the number m of available processors or in terms of ε, representing a very small cost, if none, the cost is 1. The conforming tasks have cost 0 (zero).

5.3 Scheduling the Case Studies

The performance of our case study is presented in terms of scheduling length, i.e., the time required to execute the corresponding application. Table 1 presents those execution times. The first two columns represent the corresponding input graph (C_i/G_i) in Fig. 2 and the number m of available processors. The third column present the execution time $|S(C_i)|$ obtained with our scheduling. The fourth colunm presents the best scheduling $|S(G_i)|$ obtained for a given DAG and the fifth column is the scheduling $|S'(G_i)|$ obtained for the same DAG changing the priority list. The last column will be explained later.

Since our scheduling strategies give higher priority to threads in the left side of C_i, supposing that the Critical Path will be present in this side of the DCG, we compare our results with the best ones obtained by Graham. We observe that our scheduling provides to graphs C_1, C_2, C_4 and C_5 the same performance as the best obtained by scheduling the corresponding DAG. In another hand, the performance of our strategy is not optimal for the graphs C_3, C_6, C_7 and C_8.

The algorithm converts a DAG into its equivalent DCG considering the order of the tasks in the input graph as the temporal order of their creation. The mechanism match the patterns in a breadth-first order. This way, the DCG generated does not always take advantage of the scheduling technique used by the execution environment. However, if the programmer is aware of the process of building a DCG and the scheduling algorithm, he can reorganize the DAG without prejudice to the precedence constraints arranging the tasks to fit both. Thus, graphs G_3 and G_6 can be rewritten to result the DCGs showed in Fig. 3 with execution times $|S(C_i')|$ presented in Table 1.

The situation observed in graphs C_7 and C_8 is different from that in C_3 and C_6. In those cases, $|S(C_7)| = |S'(G_7)|$ and $|S(C_8)| = |S'(G_8)|$ since the scheduler S' corresponds to a strategy based in the Critical Path, the same heuristics applied by our scheduling algorithm. Therefore, our scheduling cannot produce better results, as expected by a tipical greedy list scheduling algorithm.
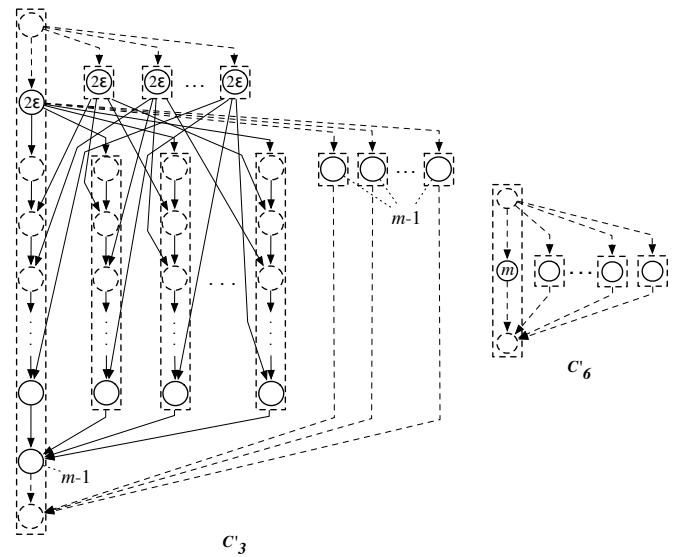

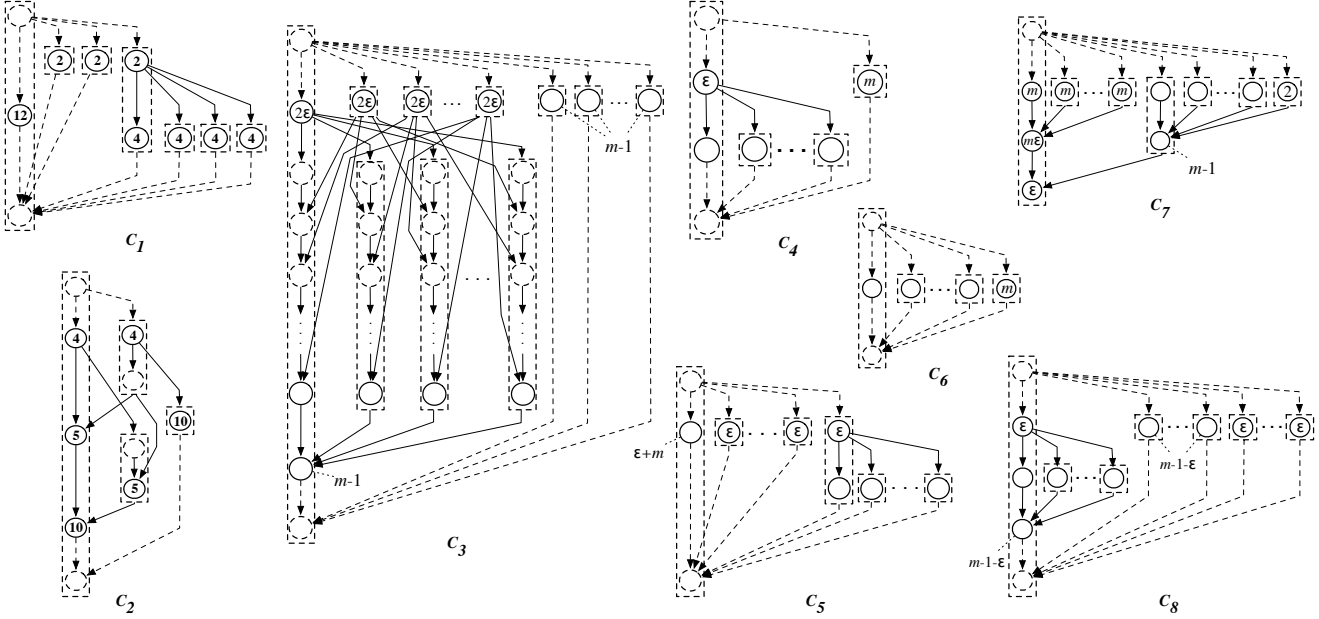**Fig. 3** Transformation over graphs G_3 and G_6 rewritten to fit the scheduling strategy.

Fig. 2 Result of the application of DCG generation.

Table 1 Scheduling of C_i.

DCG/DAG	# m	$|S(C_i)|$	$|S(G_i)|$	$|S'(G_i)|$	$|S(C_i')|$
C_1/G_1	3	12	12	14	–
C_2/G_2	2	19	19	–	–
C_3/G_3	m	$2m - 1 + 2\varepsilon$	$m + 2\varepsilon$	–	$m + 2\varepsilon$
C_4/G_4	m	$m + \varepsilon$	$m + \varepsilon$	–	–
C_5/G_5	m	$m' + 2\varepsilon$	$m' + 2\varepsilon$	–	–
C_6/G_6	m	$2m - 1$	m	$2m - 1$	m
C_7/G_7	m	$2m + \varepsilon$	$(m + 1)(1 + \varepsilon)$	$2m + \varepsilon$	–
C_8/G_8	m	$2m - 1 - 2\varepsilon$	m	$2m - 1 - 2\varepsilon$	–

6 Approximating Static Scheduling

The purpose of this section is to present a study on the performance of different heuristics for prioritizing tasks in dynamic multithreaded environments. This study aims to measure the distance in terms of schedule lenght between dynamic multithread scheduling and static DAG scheduling, both using list algorithms. In the first part of the study we simulated two distinct execution environments, one for programs described in a DAG, the other for programs described in a DCG. We executed several experiments considering five priority policies for each environment. In the second part of the study we implemented the strategy that provided better results in the simulated evironment that cosidered DCGs in the full-fledged multithreaded environment of Anahy. In both parts of the study, we believe that the random strategy for prioritizing tasks is to insert less overhead operations by scheduling no need to select the task / thread to be launched.

6.1 Simulating Dynamic Scheduling Behavior

For this paper we will present two case studies to examine the behavior of list scheduling a program described in a graph in a simulated dynamic environment. In both case studies we consider a synthetic application, whose DAG tasks and dependecies are generated ramdomly in a nested fork/join fashion. These comparisons we will make can only be obtained by simulation because in a real dynamic execution, the DAG is not available *a priori*. The scheduling lenghts are presented in Figures 4 and 5 in terms of time units. In Figure 4 we considered that all tasks cost one time unit to be executed. In Figure 5 each task has a pseudo-random cost inside the interval $[1; 10]$. For each case we run 100 simulations and considered the average of the results (the standard deviation was always below 5%).

The lines in the charts from Figures 4 and 5 represent the best results obtained using heuristics LIFO,

FIFO, RANDOM, SCFET and SCFNET for prioritizing threads using the execution modes:

- Work-First (WF): when a processor executes a *create* operation, it starts instantaneously the execution of the newly created thread and the current thread is made available for other processors to resume it, becoming suspended if there is no idle processor. If a processor executes a *join* over a non-finished thread, the current thread remains blocked until the joined thread is finished, when the previously blocked thread can be migrated to any processor to be resumed.
- Help-First(HF): when a processor executes a *create* operation, the processor continues the execution of the current thread after making the newly created thread available for other processors. During a *join* operation the scenario is the same as in Work-First mode.
- Help-First without Migration: the processor behaves the same way as in HF during thread creation. However, if a thead gets blocked on a *join* operation only the current processor can resume its execution.

For comparison purposes, we present the best static schedule obtained using the heuristics HLFET, HLFNET, RANDOM, SCFET and SCFNET, presented in Section 2.3.

As we can see in the performance charts, the schedule lengths obtained in the dynamic simulations are very close to the static schedules, recalling that in these charts we show only the best schedules considering each execution mode. The greater distance among the dynamic multithreaded simulation results and the static schedules correspond to an execution up to 12% slower than the best static scheduled one, and no multithreaded simulations provided schedule lengths less than or equal to static schedule lengths considering more than 2 processors.

Another result we could extract from the simulations is that the heuristic SCFET provided 75% of the best dynamic schedules, considering all execution modes: Work-First, and Help-First with and without migration.

6.2 Scheduling Threads in a Dynamic Environment

In this section we present the performance achieved by executing the Smith-Waterman algorithm [21] with Anahy and other three multithread parallel environments: Cilk, TBB and OpenMP. The Smith-Waterman is a dynamic algorithm conceived to align sequences of characters. In Figure 6 we present the execution times
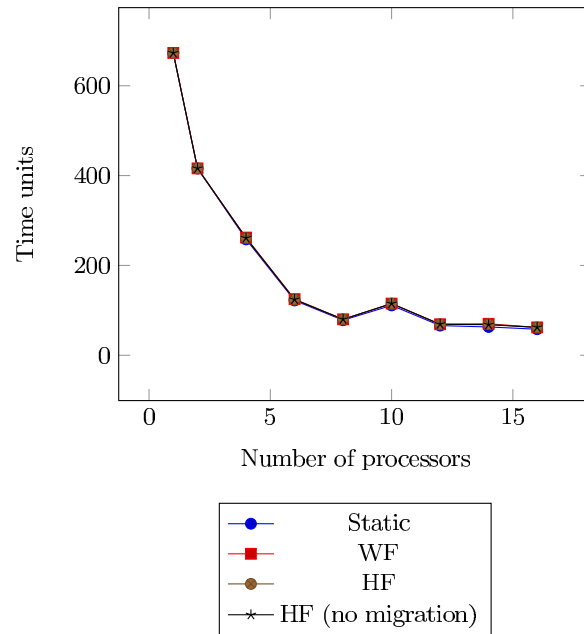

Fig. 4 Simulations considering an irregular graph structure and tasks with processing costs of 1 (one) time unit.
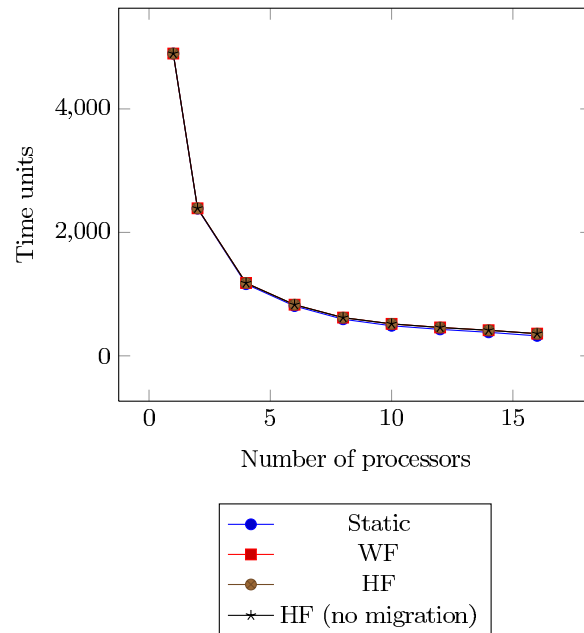

Fig. 5 Simulations considering an irregular graph structure and tasks with pseudo-random processing costs.

(in the Y-axis) we have obtained to align 1,000 strings of 1,000 characters each one. The alignment was achieved considering blocks of 10×10 elements in the algorithm's work matrix. In this graph, the X-axis represents the maximum number of threads allowed to run in parallel. Each point in this graph corresponds to the average of 100 executions (standard deviation smaller than 5%). Each curve in the graph corresponds to the execu-
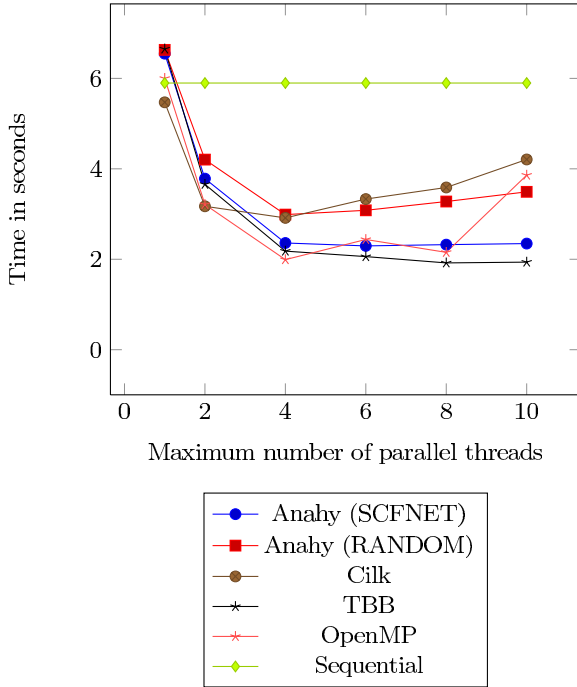
Fig. 6 Execution of the Smith-Waterman algorithm, each thread calculatig a block of 10×10 elements.

tion times obtained by Anahy (considering two different heuristics to set the priority to threads), Cilk, TBB and OpenMP. The first three environments are multithreaded programming tools that include list scheduling strategies similar to those presented in this paper in their runtime. The execution times of the OpenMP implementation are shown to have a reference to the performance. For this application, the pure sequential program executes in 5.90 s. All assessments were obtained in a quad-core multiprocessor (hyper-threading enabled) with 8 GB of RAM under a GNU-Linux operating system.

To analyze the performance of those environments we need first to consider the scheduling strategy adopted by each one. Anahy runtime implements the Help-First execution mode without migration and uses two different heuristics to prioritize threads: SCFNET and RANDOM. In the implementation is necessary to consider that the scheduling is achieved in terms of threads not in terms of tasks as a basic list algorithm does. Thus, by applying the SCFNET heuristic, the priority of a thread is given by the co-level of the currently active task in that thread context. This means that the co-level of a thread is updated on each invocation of the create and join operations. The RANDOM heuristic is presented for sake of comparison. In other hand, both Cilk and TBB runtimes implement the Work-First execution mode with migration. The priority of threads is given considering two situations: locally to a processor

and globally among processors. A FIFO heuristic is applied globally, when a processor becomes idle but has no threads ready to execute locally. When a processor becomes idle and has ready work on its local queue, a LIFO heuristic is applied, so the processor executes the most recent thread. The major difference among Cilk and TBB is that the last one allows us to describe a program with a large amount of parallel activities for this specific application.

In the graph of Figure 6 we observe that the better executions times were obtained when the maximum number of parallel threads was limited to four, except to TBB that in this experiment could exploit the hyperthreading. In this graph we highlight the difference of performance between the two curves representing two different heuristics to prioritize threads in Anahy: the implementation employing the SCFNET heuristic is clearly better than when the RANDOM one is employed. This fact indicates that the use of an elaborated scheduling strategy, taking into account the application requirements, could affect positively the final performance regardless the overhead introduced by the scheduling operations.

In this section we have presented only a small part of the set of experiments achieved in the context of our research. Those results allow us to conclude that multithread environments can apply scheduling strategies based on list algorithm to improve the performance execution. We have shown also that different heuristics to prioritize threads can modify the final performance.

7 Concluding Remarks

List scheduling algorithms are the basis for many scheduling techniques for various programming environments. In particular, Cilk and Athreads use this concept together with a heuristic to prioritize the execution of threads in the critical path to schedule threads dynamically at application level. In this kind of environment, the main challenge is to find the critical path of an application described in terms of a DCG instead of a DAG, as tipical list algorithms are used to deal with.

This work presented a strategy to build a DGC representing a multithreaded program apart from a DAG describing a dataflow program and a set of experiments to evaluate the impact of approximations of list scheduling strategies on an actual dynamic runtime environment. This allowed us to compare DAG schedules obtained in static scenarios with DCG schedules for dynamic executions, where both DAG and DCG represent the same program. It also presented a multithread scheduling strategy based in the Critical Path of the

application. This strategy considers that the Critical Path will be present in the left most threads of a DCG.

Performance results shown that the scheduling strategy applied to the DCGs can provide execution times as good as those provided by regular list scheduling applied to DAGs, when both describe the same application. The limits of list scheduling, when the distance to the Critical Path is the attribute priority of the tasks, were still preserved in the proposed scheduling strategy.

We conclude that list scheduling algorithms can be introduced in runtimes of multithreaded environments since good performances can be effectively achieved. The programmer must, nevertheless, be aware of the adopted scheduling strategy to organize the relationships among tasks and hence harvest the benefits of its characteristics.

References

1. Albers, S.: Better bounds for on-line scheduling. SIAM Journal on Computing p. 459473 (1999)
2. Bakshi, A., Prasanna, V., Reich, J., Larner, D.: The abstract task graph: A methodology for architecture-independent programming of networked sensor systema. In: Workshop on End-to-Ed, Sense-and-Respond Systems, Applications, and Services (2005)
3. Bartal, Y., Fiat, A., Karloff, H., , Vohra, R.: New algorithms for an ancient scheduling problem. Journal Computer Systems Science **51**, 359–366 (1995)
4. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on, pp. 356–368. IEEE (1994)
5. Blumofe, R.D., et al.: Cilk: an efficient multithreaded runtime system. ACM SIGPLAN Not. **30**(8) (1995)
6. Camargo, C.A.S., Cavalheiro, G.G.H., Pilla, M.L., da Costa, S.A., Foss, L.: Applying list scheduling algorithms in a multithreaded execution environment. In: Proceedings of the V Latin American Symposium on High Performance Computing (2012)
7. Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Software Eng. **14**(2), 141–154 (1988)
8. Cavalheiro, G.G.H., Gaspary, L.P., Cardozo, M.A., Cordeiro, O.C.: Anahy: A programming environment for cluster computing. In: VII High Performance Computing for Computational Science. Springer-Verlag, Berlin (2007). (LNCS 4395)
9. Cavalheiro, G.G.H., Viosa, E.: Athreads: a dataflow programming interface for multiprocessors. In: LTPD 2007. Gramado-RS (2007)
10. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan Kaufmann, San Francisco (2001)
11. Coffman Jr, E., Denning, P.: Operating systems theory. Prentice Hall Professional Technical Reference (1973)
12. Cordeiro, O., Peranconi, D., Villa Real, L., Dall'Agnol, E., Cavalheiro, G.: Exploiting multithreaded programming on cluster architectures. In: HPCS. Guelph (2005)
13. El-Rewini, H., Lewis, T.G.: Scheduling parallel program tasks onto arbitrary target machines. J. Parallel Distrib. Comput. **9**(2), 138–153 (1990). DOI http://dx.doi.org/10.1016/0743-7315(90)90042-N
14. Fleischer, R., Wahl, M.: On-line scheduling revisited. Journal of Scheduling p. 343353 (2000)
15. Graham, R.: Bounds on the performance of scheduling algorithms. Computer and Job-Shop Scheduling Theory pp. 165–227 (1976)
16. Graham, R.L.: Bounds for certain multiprocessor anomalies. The Bell Syst. Tech. J. **CLV**(9) (1966)
17. Hwang, J.J., Chow, Y.C., Anger, F.D., Lee, C.Y.: Scheduling precedence graphs in systems with interprocessor communication times. SIAM J. Comput. **18**(2), 244–257 (1989). DOI http://dx.doi.org/10.1137/0218016
18. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**(1), 1–34 (2004). DOI http://doi.acm.org/10.1145/1013208.1013209
19. Karger, D.R., Phillips, S.J., Torng, E.: A better algorithm for an ancient scheduling problem. Journal of Algorithms **20**, 400–430 (1996)
20. Reinders, J.: Intel threading building blocks, first edn. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007)
21. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology **215**, 403–410 (1990)