Parallelizing with BDSC, a Resource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems

Dounia Khaldi[a,*], Pierre Jouvelot[a], Corinne Ancourt[a]

[a] *CRI, Mathématiques et systèmes, MINES ParisTech*
35 rue Saint-Honoré, 77305 Fontainebleau

Abstract

We introduce a new parallelization framework for scientific computing based on BDSC, an efficient automatic scheduling algorithm for parallel programs in the presence of resource constraints on the number of processors and their local memory size. BDSC extends Yang and Gerasoulis's Dominant Sequence Clustering (DSC) algorithm; it uses sophisticated cost models and addresses both shared and distributed parallel memory architectures. We describe BDSC, its integration within the PIPS compiler infrastructure and its application to the parallelization of four well-known scientific applications: Harris, ABF, equake and IS. Our experiments suggest that BDSC's focus on efficient resource management leads to significant parallelization speedups on both shared and distributed memory systems, improving upon DSC results, as shown by the comparison of the sequential and parallelized versions of these four applications running on both OpenMP and MPI frameworks.

Keywords: Task parallelism, Static scheduling, DSC algorithm, Shared memory, Distributed memory, PIPS

1. Introduction

"Anyone can build a fast CPU. The trick is to build a fast system." Attributed to Seymour Cray, this quote is even more pertinent when looking at multiprocessor systems that contain several fast processing units; parallel system architectures introduce subtle system constraints to achieve good performance. Real world applications, which operate on a large amount of data, must be able to deal with limitations such as memory requirements, code size and processor features. These constraints must also be addressed by parallelizing compilers

[*] Corresponding author, phone: +33 (1) 6469 4708
Email addresses: khaldi.dounia@gmail.com (Dounia Khaldi), pierre.jouvelot@mines-paristech.fr (Pierre Jouvelot), corinne.ancourt@mines-paristech.fr (Corinne Ancourt)

that target such applications and translate sequential codes into efficient parallel ones.

One key issue when attempting to parallelize sequential programs is to find solutions to graph partitioning and scheduling problems, where vertices represent computational tasks and edges, data exchanges. Each vertex is labeled with an estimation of the time taken by the computation it performs; similarly, each edge is assigned a cost that reflects the amount of data that need to be exchanged between its adjacent vertices. Task scheduling is the process that assigns a set of tasks to a network of processors such that the completion time of the whole application is as small as possible while respecting the dependence constraints of each task. Usually, the number of tasks exceeds the number of processors; thus some processors are dedicated to multiple tasks. Since finding the optimal solution of a general scheduling problem is NP-complete [1], providing an efficient heuristic to find a good solution is needed. Efficiency is strongly dependent here on the accuracy of the cost information encoded in the graph to be scheduled. Gathering such information is a difficult process in general, in particular in our case where tasks are automatically generated from program code.

Scheduling approaches that use static predictions of task characteristics such as execution time and communication cost can be categorized in many ways, such as static/dynamic and preemptive/non-preemptive. Dynamic (online) schedulers make run-time decisions regarding processor mapping whenever new tasks arrive. These schedulers are able to provide good schedules even while managing resource-constraints (see for instance Tse [2]) and are particularly well suited when task time information is not perfect, even though they introduce run-time overhead. In this class of dynamic schedulers, one can even distinguish between preemptive and non-preemptive techniques. In preemptive schedulers, the current executing task can be interrupted by an other higher-priority task, while, in a non-preemptive scheme, a task keeps its processor until termination. Preemptive scheduling algorithms are instrumental in avoiding possible deadlocks and for implementing real-time systems, where tasks must adhere to specified deadlines. EDF (Earliest-Deadline-First) [3] and EDZL (Earliest Deadline Zero Laxity) [4] are examples of preemptive scheduling algorithms for real-time systems. Preemptive schedulers suffer from possible loss of predictability, when overloaded processors are unable to meet the tasks' deadlines.

In the context of the automatic parallelization of scientific applications we focus on in this paper, for which task time can be assessed with rather good precision (see Section 4.2), we decide to focus on (non-preemptive) static scheduling policies[1]. Even though the subject of static scheduling is rather mature (see Section 6.1), we believe the advent and widespread market use of multi-core architectures, with the constraints they impose, warrant to take a fresh look at its potential. Indeed, static scheduling mechanisms have, first, the strong advan-

[1]Note that more expensive preemptive schedulers would be required if fairness concerns were high, which is not frequently the case in the applications we address here.

tage of reducing run-time overheads, a key factor when considering execution time and energy usage metrics. One other important advantage of these schedulers over dynamic ones, at least over those not equipped with detailed static task information, is that the existence of efficient schedules is ensured prior to program execution. This is usually not an issue when time performance is the only goal at stake, but much more so when memory constraints might impede a task to be executed at all on a given architecture. Finally, static schedules are predictable, which helps both at the specification (if such a requirement has been introduced by designers) and debugging levels.

We introduce thus a new non-preemptive static scheduling heuristic that strives to give as small as possible schedule lengths, i.e., parallel execution time, in order to extract the task-level parallelism possibly present in sequential programs, while enforcing architecture-dependent constraints. Our approach takes into account resource constraints, i.e., the number of processors, the computational cost and memory use of each task and the communication cost for each task exchange[2], to provide hopefully significant speedups on realistic shared and distributed computer architectures. Our technique, called BDSC, is based on an existing best-of-breed static scheduling heuristic, namely Yang and Gerasoulis's DSC (Dominant Sequence Clustering) list-scheduling algorithm [5] [6], that we equip to deal with new heuristics that handle resource constraints. One key advantage of DSC over other scheduling policies (see Section 6), besides its already good performance when the number of processors is unlimited, is that it has been proven optimal for fork/join graphs: this is a serious asset given our focus on the program parallelization process, since task graphs representing parallel programs often use this particular graph pattern. Even though this property may be lost when constraints are taken into accounts, our experiments on scientific benchmarks suggest that our extension still provides good performance speedups (see Section 5).

If scheduling algorithms are key issues in sequential program parallelization, they need to be properly integrated into compilation platforms to be used in practice. These environments are in particular expected to provide the data required for scheduling purposes, a difficult problem we already mentioned. Beside BDSC, our paper introduces also new static program analysis techniques to gather the information required to perform scheduling while ensuring that resource constraints are met, namely a static instruction and communication cost models, a data dependence graph to enforce scheduling constraints and static information regarding the volume of data exchanged between program fragments.

The main contributions of this paper, which introduces a new task-based parallelization approach that takes into account resource constraints during the static scheduling process, are:

[2]Note that processors are usually not considered as a resource in the literature dealing with scheduling theory. One originality of the approach used in this paper is to suggest to consider this factor as a resource among others, such as memory limitations.

- "Bounded DSC" (BDSC), an extension of DSC that simultaneously handles two resource constraints, namely a bounded amount of memory per processor and a bounded number of processors, which are key parameters when scheduling tasks on actual parallel architectures;
- a new BDSC-based hierarchical scheduling algorithm (HBDSC) that uses a new data structure, called the Sequence Data Dependence Graph (SDG), to represent partitioned parallel programs;
- an implementation of HBDSC-based parallelization in the PIPS [7] source-to-source compilation framework, using new cost models based on time complexity measures, convex polyhedral approximations of data array sizes and code instrumentation for the labeling of SDG vertices and edges;
- performance measures related to the BDSC-based parallelization of four significant programs, targeting both shared and distributed memory architectures: the image and signal processing applications Harris and ABF, the SPEC2001 benchmark equake and the NAS parallel benchmark IS.

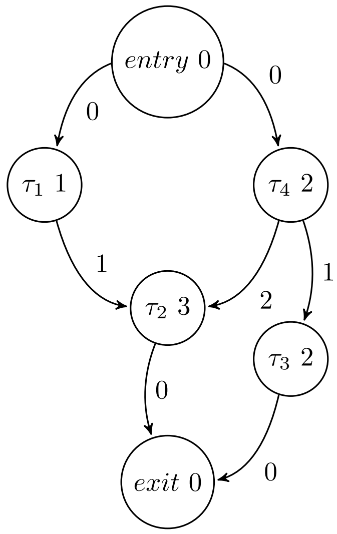
This paper is organized as follows. Section 2 presents the original DSC algorithm that we intend to extend. We detail our algorithmic extension, BDSC, in Section 3. Section 4 introduces the partitioning of a source code into a Sequence Dependence Graph (SDG), our cost models for the labeling of this SDG and a new BDSC-based hierarchical scheduling algorithm (HBDSC). Section 5 provides the performance results of four scientific applications parallelized on the PIPS platform: Harris, ABF, equake and IS. We also assess the sensitivity of our parallelization technique on the accuracy of the static approximations of the code execution time used in task scheduling. Section 6 compares the main existing scheduling algorithms and parallelization platforms with our approach. Finally Section 7 concludes the paper and addresses future work.

2. List Scheduling: the DSC Algorithm

In this section, we introduce the notion of list-scheduling heuristics and present the list-scheduling heuristic called DSC [5].

2.1. List-Scheduling Processes

A labelled direct acyclic graph (DAG) G is defined as $G = (T, E, D)$, where (1) $T = \mathit{vertices}(G)$ is a set of n tasks (vertices) τ annotated with an estimation of their execution time $\mathit{task\_time}(\tau)$, (2) E, a set of m edges $e = (\tau_i, \tau_j)$ between two tasks, and (3) D, a $n \times n$ communication edge cost matrix $\mathit{edge\_cost}(e)$; $\mathit{task\_time}(\tau)$ and $\mathit{edge\_cost}(e)$ are assumed to be numerical constants, although we show how we lift this restriction in Section 4.2. The functions $\mathit{successors}(\tau, G)$ and $\mathit{predecessors}(\tau, G)$ return the list of immediate successors and predecessors of a task τ in the DAG G. Figure 1 provides an example of a simple graph, with vertices τ_i; vertex times are listed in the vertex circles while edge costs label arrows.



step	task	tlevel	blevel	DS	scheduled tlevel		
					κ_0	κ_1	κ_2
1	τ_4	0	7	7	0*		
2	τ_3	3	2	5	2	3*	
3	τ_1	0	5	5			0*
4	τ_2	4	3	7	2*		4

Figure 1: A Directed Acyclic Graph (left) and its scheduling (right); starred tlevels (*) correspond to the selected clusters

A list scheduling process provides, from a DAG G, a sequence of its vertices that satisfies the relationship imposed by E. Various heuristics try to minimize the schedule total length, possibly allocating the various vertices in different clusters, which ultimately will correspond to different processes or threads. A cluster κ is thus a list of tasks; if $\tau \in \kappa$, we note $cluster(\tau) = \kappa$. List scheduling is based on the notion of vertex priorities. The priority for each task τ is computed using the following attributes:

- The top level $tlevel(\tau, G)$ of a vertex τ is the length of the longest path from the entry vertex of G to τ. The length of a path is the sum of the communication cost of the edges and the computational time of the vertices along the path. Tlevels are used to estimate the start times of vertices on processors: the tlevel is the earliest possible start time. Scheduling in an ascending order of tlevel tends to schedule vertices in a topological order. The algorithm for computing the top level of a vertex τ in a graph is given in Algorithm 1.

- The bottom level $blevel(\tau, G)$ of a vertex τ is the length of the longest path from τ to the exit vertex of G. The maximum of the blevel of vertices is the length $cpl(G)$ of a graph's critical path, which has the longest path in the DAG G. The latest start time of a vertex τ is the difference $(cpl(G) - blevel(\tau, G))$ between the critical path length and the bottom level of τ. Scheduling in a descending order of blevel tends to schedule critical path vertices first. The algorithm for computing the bottom level of τ in a graph is given in Algorithm 2.

To illustrate these notions, the tlevels and blevels of each vertex of the graph

ALGORITHM 1: The tlevel of Task τ in Graph G

```
function tlevel($\tau$, G)
  tl = 0;
  foreach $\tau_i$ $\in$ predecessors($\tau$, G)
    level = tlevel($\tau_i$, G)+task_time($\tau_i$)+edge_cost($\tau_i$, $\tau$);
    if (tl < level) then tl = level;
  return tl;
end
```

ALGORITHM 2: The blevel of Task τ in Graph G

```
function blevel($\tau$, G)
  bl = 0;
  foreach $\tau_j$ $\in$ successors($\tau$, G)
    level = blevel($\tau_j$, G)+edge_cost($\tau$, $\tau_j$);
    if (bl < level) then bl = level;
  return bl+task_time($\tau$);
end
```

presented in the left of Figure 1 are provided in the adjacent table (we discuss the other entries in this table later on).

The general algorithmic skeleton for list scheduling a graph G on P clusters (P can be infinite and is assumed to be always strictly positive) is provided in Algorithm 3: first, priorities $priority(\tau)$ are computed for all currently unscheduled vertices; then, the vertex with the highest priority is selected for scheduling; finally, this vertex is allocated to the cluster that offers the earliest start time. Function f characterizes each specific heuristic, while the set of clusters already allocated to tasks is *clusters*. Priorities need to be computed again for (a possibly updated) graph G after each scheduling of a task: task times and communication costs change when tasks are allocated to clusters. This is performed by the *update_priority_values* function call.

2.2. The DSC Algorithm

DSC (Dominant Sequence Clustering) is a list-scheduling heuristic for an unbounded number of processors. The objective is to minimize the top level of each task. A DS (Dominant Sequence) is a path that has the longest length in a partially scheduled DAG; a graph critical path is thus a DS for the totally scheduled DAG. The DSC heuristic computes a Dominant Sequence (DS) after each vertex is processed, using $tlevel(\tau, G)+blevel(\tau, G)$ as $priority(\tau)$. A ready

ALGORITHM 3: List scheduling of Graph G on P processors

```
procedure list_scheduling(G, P)
  clusters = $\emptyset$;
  foreach $\tau_i \in$ vertices(G)
    priority($\tau_i$) = $f$(tlevel($\tau_i$, G), blevel($\tau_i$, G));
  UT = vertices(G);                // unscheduled tasks
  while UT $\neq \emptyset$
    $\tau$ = select_task_with_highest_priority(UT);
    $\kappa$ = select_cluster($\tau$, G, P, clusters);
    allocate_task_to_cluster($\tau$, $\kappa$, G);
    update_graph(G);
    update_priority_values(G);
    UT = UT$-\{\tau\}$;
end
```

vertex τ, i.e., for which all predecessors have already been scheduled[3], on one of the current DSs, i.e., with the highest priority, is clustered with a predecessor τ_p when this reduces the tlevel of τ by zeroing, i.e., setting to zero, the cost of the incident edge (τ_p, τ).

To decide which predecessor τ_p to select, DSC applies the minimization procedure *tlevel_decrease*, which returns the predecessor that leads to the highest reduction of tlevel for τ if clustered together, and the resulting tlevel; if no zeroing is accepted, the vertex τ is kept in a new single vertex cluster[4]. More precisely, the minimization procedure *tlevel_decrease* for a task τ, in Algorithm 4, tries to find the cluster $cluster(min\_\tau)$ of one of its predecessors τ_p that reduces the tlevel of τ as much as possible by zeroing the cost of the edge (min_τ, τ). All clusters start at the same time, and each cluster is characterized by its running time, $cluster\_time(\kappa)$, which is the cumulated time of all tasks τ scheduled into κ; idle slots within clusters may exist and are also taken into account in this accumulation process. The condition $cluster(\tau_p) \neq cluster\_undefined$ is tested on predecessors of τ in order to make it possible to apply this procedure for ready and unready τ vertices; an unready vertex has at least one unscheduled predecessor.

DSC is the instance of Algorithm 3 where *select_cluster* is replaced by the code in Algorithm 5 (*new_cluster* extends *clusters* with a new empty cluster; its cluster time is set to 0). Note that *min_tlevel* will be used in Section 2.3. Since priorities are updated after each iteration, DSC computes dynamically the critical path based on both tlevel and blevel information. The table in Figure 1

[3]Part of the *allocate_task_to_cluster* procedure is to ensure that $cluster(\tau) = \kappa$, which indicates that Task τ is now scheduled on Cluster κ.

[4]In fact, DSC implements a somewhat more involved zeroing process, by selecting multiple predecessors that need to be clustered together with τ. We implemented this more sophisticated version, but left these technicalities outside of this paper for readability purposes.

represents the result of scheduling the DAG in the same figure using the DSC algorithm.

ALGORITHM 4: Minimization DSC procedure for Task τ in Graph G

```
function tlevel_decrease($\tau$, G)
  min_tlevel = tlevel($\tau$, G);
  min_$\tau$ = $\tau$;
  foreach $\tau_p$ ∈ predecessors($\tau$, G)
        where cluster($\tau_p$) ≠ cluster_undefined
    start_time = cluster_time(cluster($\tau_p$))),
    foreach $\tau'_p$ ∈ predecessors($\tau$, G) where
            cluster($\tau'_p$) ≠ cluster_undefined
      if($\tau_p$ ≠ $\tau'_p$) then
        level = tlevel($\tau'_p$, G)+task_time($\tau'_p$)+edge_cost($\tau'_p$, $\tau$);
        start_time = max(level, start_time);
    if(min_tlevel > start_time) then
      min_tlevel = start_time;
      min_$\tau$ = $\tau_p$;
  return (min_$\tau$, min_tlevel);
end
```

ALGORITHM 5: DSC cluster selection for Task τ for Graph G on P processors

```
function select_cluster($\tau$, G, P, clusters)
  (min_$\tau$, min_tlevel) = tlevel_decrease($\tau$, G);
  return (cluster(min_$\tau$) ≠ cluster_undefined) ?
                cluster(min_$\tau$) : new_cluster(clusters);
end
```

2.3. Dominant Sequence Length Reduction Warranty (DSRW)

DSRW is an additional greedy heuristic within DSC that aims to further reduce the schedule length. A vertex on the DS path with the highest priority can be ready or not ready. With the DSRW heuristic, DSC schedules the ready vertices first, but, if such a ready vertex τ_r is not on the DS path, DSRW verifies, using the procedure in Algorithm 6, that the corresponding zeroing does not affect later the reduction of the tlevels of the DS vertices τ_u that are partially ready, i.e., such that there exists at least one unscheduled predecessor of τ_u. To do this, we check if the "partial top level" of τ_u, which does not take into account unexamined (unscheduled) predecessors and is computed using *tlevel_decrease*, is reducible, once τ_r is scheduled.

The table in Figure 1 illustrates an example where it is useful to apply the DSRW optimization. There, the DS column provides, for the task scheduled

ALGORITHM 6: DSRW optimization for Task τ_u when scheduling Task τ_r for Graph G

```
function DSRW($\tau_r$, $\tau_u$, clusters, G)
  (min_$\tau$, min_tlevel) = tlevel_decrease($\tau_r$, G);

  // before scheduling $\tau_r$
  ($\tau_b$,ptlevel_before) = tlevel_decrease($\tau_u$, G);

  // scheduling $\tau_r$
  allocate_task_to_cluster($\tau_r$, cluster(min_$\tau$), G);
  saved_edge_cost = edge_cost(min_$\tau$, $\tau_r$);
  edge_cost(min_$\tau$,$\tau_r$) = 0;

  // after scheduling $\tau_r$
  ($\tau_a$,ptlevel_after) = tlevel_decrease($\tau_u$, G);
  if (ptlevel_after > ptlevel_before) then

    // (min_$\tau$,$\tau_r$) zeroing not accepted
    edge_cost(min_$\tau$, $\tau_r$) = saved_edge_cost;
    return false;
  return true;
end
```

κ_0	κ_1
τ_4	τ_1
τ_3	τ_2

κ_0	κ_1	κ_2
τ_4		τ_1
τ_2	τ_3	

Figure 2: Result of DSC on the graph in Figure 1 without (left) and with (right) DSRW

at each step, its priority, i.e., the length of its dominant sequence, while the last column represents, for each possible zeroing, the corresponding task tlevel; starred tlevels (*) correspond to the selected clusters. Task τ_4 is mapped to Cluster κ_0 in the first step of DSC. Then, τ_3 is selected because it is the ready task with the highest priority. The mapping of τ_3 to Cluster κ_0 would reduce its tlevel from 3 to 2. But the zeroing of (τ_4, τ_3) affects the tlevel of τ_2, τ_2 being the unready task with the highest priority. Since the partial tlevel of τ_2 is 2 with the zeroing of (τ_4,τ_2) but 4 after the zeroing of (τ_4,τ_3), DSRW will fail, and DSC allocates τ_3 to a new cluster, κ_1. Then, τ_1 is allocated to a new cluster, κ_2, since it has no predecessors. Thus, the zeroing of (τ_4,τ_2) is kept thanks to the DSRW optimization; the total schedule length is 5 (with DSRW) instead of 7 (without DSRW) (Figure 2).

3. BDSC: A Memory-Constrained, Number of Processor-Bounded Extension of DSC

This section details the key ideas at the core of our new scheduling process BDSC, which extends DSC with a number of important features, namely (1) verifying predefined memory constraints, (2) targeting a bounded number of processors and (3) trying to make this number as small as possible.

3.1. DSC Weaknesses

A good scheduling solution is a solution that is built carefully, by having knowledge about previous scheduled tasks and tasks to arrive in the future. Yet, as stated in [8], "an algorithm that only considers blevel or only tlevel cannot guarantee optimal solutions". Even though DSC is a policy that uses the critical path for computing dynamic priorities based on both the blevel and the tlevel for each vertex, it has some limits in practice.

The key weakness of DSC for our purpose is that the number of processors cannot be predefined; DSC yields blind clusterings, disregarding resource issues. Therefore, in practice, a thresholding mechanism to limit the number of generated clusters should be introduced. When allocating new clusters, one should verify that the number of clusters does not exceed a predefined threshold P (Section 3.3). Also, zeroings should handle memory constraints, i.e., by verifying that the resulting clustering does not lead to cluster data sizes that exceed a predefined cluster memory threshold M (Section 3.3).

Finally, DSC may generate a lot of idle slots in the created clusters. It adds a new cluster when no zeroing is accepted without verifying the possible existence of gaps in existing clusters. We handle this case in Section 3.4, adding an efficient idle cluster slot allocation routine in the task-to-cluster mapping process.

3.2. Resource Modeling

Since our extension deals with computer resources, we assume that each vertex in a DAG is equipped with an additional information, $task\_data(\tau)$, which is an over-approximation of the memory space used by Task τ; its size is assumed to be always strictly less than M. A similar *cluster_data* function applies to clusters, where it represents the collective data space used by the tasks scheduled within it. Since BDSC, as DSC, needs execution times and communication costs to be numerical constants, we discuss in Section 4.2 how this information is computed.

Our improvement to the DSC heuristic intends to reach a tradeoff between the gained parallelism and the communication overhead between processors, under two resource constraints: finite number of processors and amount of memory. We track these resources in our implementation of *allocate_task_to_cluster* given in Algorithm 7; note that the aggregation function *data_merge* is defined in Section 4.2.

ALGORITHM 7: Task allocation of Task τ in Graph G to Cluster κ, with resource management

```
procedure allocate_task_to_cluster($\tau$, $\kappa$, G)
  cluster($\tau$) = $\kappa$;
  cluster_time($\kappa$) = max(cluster_time($\kappa$), tlevel($\tau$, G)) +
                           task_time($\tau$);
  cluster_data($\kappa$) = regions_union(cluster_data($\kappa$),
                                        task_data($\tau$));
end
```

Efficiently allocating tasks on the target architecture requires reducing the communication overhead and transfer cost for both shared and distributed memory architectures. If zeroing operations, that reduce the start time of each task and nullify the corresponding *edge_cost*, are obviously meaningful for distributed memory systems, they are also worthwhile on shared memory architectures. Merging two tasks in the same cluster keeps the data in the local memory, and even possibly cache, of each thread and avoids their copying over the shared memory bus. Therefore, transmission costs are decreased and bus contention is reduced.

3.3. Resource Constraint Warranty

Resource usage affects speed. Thus, parallelization algorithms should try to limit the size of the memory used by tasks. BDSC introduces a new heuristic to control the amount of memory used by a cluster, via the user-defined memory upper bound parameter M. The limitation of the memory size of tasks is important when (1) executing large applications that operate on large amount of data, (2) M represents the processor local (or cache) memory size, since, if the memory limitation is not respected, transfer between the global and local memories may occur during execution and may result in performance degradation, and (3) targeting embedded systems architecture. For each task τ, BDSC computes an over-approximation of the amount of data that τ allocates to perform read and write operations; it is used to check that the memory constraint of Cluster κ is satisfied whenever τ is included in κ. Algorithm 8 implements this memory constraint warranty MCW; *data_merge* and *data_size* are functions that respectively merge data and yield the size (in bytes) of data (see Section 4.2).

The previous line of reasoning is well adapted to a distributed memory architecture. When dealing with a multicore equipped with a purely shared memory, such per-cluster memory constraint is less meaningful. We can nonetheless keep the MCW constraint check within the BDSC algorithm even in this case, if we set M to the size of the global shared memory. A positive by-product of this design choice is that BDSC is able, in the shared memory case, to reject computations that need more memory space than available, even within a single cluster.

ALGORITHM 8: Resource constraint warranties, on memory size M and processor number P

```
function MCW(τ, κ, M)
  merged_data = data_merge(cluster_data(κ), task_data(τ));
  return data_size(merged_data) ≤ M;
end
function PCW(clusters, P)
  return |clusters| < P;
end
```

Another scarce resource is the number of processors. In the original policy of DSC, when no zeroing for τ is accepted, i.e. that would decrease its start time, τ is allocated to a new cluster. In order to limit the number of created clusters, we propose to introduce a user-defined cluster threshold P. This processor constraint warranty PCW is defined in Algorithm 8.

3.4. Efficient Task-to-Cluster Mapping

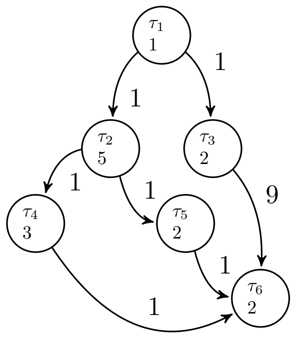
In the original policy of DSC, when no zeroings are accepted – because none would decrease the start time of Vertex τ or DSRW failed –, τ is allocated to a new cluster. This cluster creation is not necessary when idle slots are present at the end of other clusters; thus, we suggest to select instead one of these idle slots, if this can decrease the start time of τ, without affecting the scheduling of the successors of the vertices already in these clusters. To insure this, these successors must have already been scheduled or they must be a subset of the successors of τ. Therefore, in order to efficiently use clusters and not introduce additional clusters without needing it, we propose to schedule τ to the cluster that verifies this optimizing constraint, if no zeroing is accepted.

This extension of DSC we introduce in BDSC amounts thus to replacing each definition of the cluster of τ to a new cluster by a call to *end_idle_clusters*. The *end_idle_clusters* function given in Algorithm 9 returns, among the idle clusters, the ones that finished the most recently before τ's top level or the empty set, if none is found. This assumes, of course, that τ's dependencies are compatible with this choice.

To illustrate the importance of this heuristic, suppose we have the DAG presented in Figure 3. Table 4 exhibits the difference in scheduling obtained by DSC and our extension on this graph. We observe here that the number of clusters generated using DSC is 3, with 5 idle slots, while BDSC needs only 2 clusters, with 2 idle slots. Moreover, BDSC achieves a better load balancing than DSC, since it reduces the variance of the clusters' execution loads, defined, for a given cluster, as the sum of the costs of all its tasks: 0.25, for BDSC, vs. 6, for DSC. Finally, with our efficient task-to-cluster mapping, in addition to decreasing the number of generated clusters, we gain also in the total execution time. Indeed, our approach reduces communication costs by allocating tasks to

ALGORITHM 9: Efficiently mapping Task τ in Graph G to clusters, if possible

```
function end_idle_clusters($\tau$, G, clusters)
  idle_clusters = clusters;
  foreach $\kappa \in$ clusters
    if(cluster_time($\kappa$) $\leq$ tlevel($\tau$, G)) then
      end_idle_p = TRUE;
      foreach $\tau_\kappa \in$ vertices(G) where cluster($\tau_\kappa$) = $\kappa$
        foreach $\tau_s \in$ successors($\tau_\kappa$, G)
          end_idle_p $\wedge$= cluster($\tau_s$) $\neq$ cluster_undefined $\vee$
                         $\tau_s \in$ successors($\tau$, G);
      if($\neg$end_idle_p) then
        idle_clusters = idle_clusters$-\{\kappa\}$;
  last_clusters = argmax$_{\kappa \in \texttt{idle\_clusters}}$ cluster_time($\kappa$);
  return (idle_clusters != $\emptyset$) ? last_clusters : $\emptyset$;
end
```



step	task	tlevel	blevel	DS	scheduled tlevel	
					κ_0	κ_1
1	τ_1	0	15	15	0*	
2	τ_3	2	13	15	1*	
3	τ_2	2	12	14	3	2*
4	τ_4	8	6	14		7*
5	τ_5	8	5	13	8*	10
6	τ_6	13	2	15	11*	12

Figure 3: A DAG amenable to cluster minimization (left) and its BDSC step-by-step scheduling (right)

the same cluster; for example, as shown in Figure 4, the total execution time with DSC is 14, but is equal to 13 with BDSC.

To get a feeling for the way BDSC operates, we detail the steps taken to get this better scheduling in the table of Figure 3. BDSC is equivalent to DSC until Step 5, where κ_0 is chosen by our cluster mapping heuristic, since $successors(\tau_3, G) \subset successors(\tau_5, G)$; no new cluster needs to be allocated.

3.5. The BDSC Algorithm

BDSC extends the list scheduling template provided in Algorithm 3 by taking into account the various extensions discussed above. In a nutshell, the BDSC *select_cluster* function, which decides in which cluster κ a task τ should be allocated, tries successively the four following strategies:

κ_0	κ_1	κ_2	total time
τ_1			1
τ_3	τ_2		7
	τ_4	τ_5	10
τ_6			14

κ_0	κ_1	total time
τ_1		1
τ_3	τ_2	7
τ_5	τ_4	10
τ_6		13

Figure 4: DSC (left) and BDSC (right) cluster allocations and execution times

1. choose κ among the clusters of τ's predecessors that decrease the start time of τ, under MCW and DSRW constraints;
2. or, assign κ using our efficient task-to-cluster mapping strategy, under the additional constraint MCW;
3. or, create a new cluster if the PCW constraint is satisfied;
4. otherwise, choose the cluster among all clusters in $MCW\_clusters\_min$ under the constraint MCW. Note that, in this worst case scenario, the tlevel of τ can be increased, leading to a decrease in performance since the length of the graph critical path is also increased.

BDSC is described in Algorithms 10 and 11; the entry graph G_u is the whole unscheduled program DAG, P, the maximum number of processors, and M, the maximum amount of memory available in a cluster. UT denotes the set of unexamined tasks at each BDSC iteration, RL, the set of ready tasks and URL, the set of unready ones. We schedule the vertices of G according to the four rules above in a descending order of the vertices' priorities. Each time a task τ_r has been scheduled, all the newly readied vertices are added to the set RL (ready list) by the *update_ready_set* function.

BDSC returns a *scheduled* graph, i.e., an updated graph where some zeroings may have been performed and for which the *clusters* function yields the clusters needed by the given schedule; this schedule includes, beside the new graph, the cluster allocation function on tasks, *cluster*. If not enough memory is available, BDSC returns the original graph, and signals its failure by setting *clusters* to the empty set.

We suggest to apply here an additional heuristic, in that, if multiple vertices have the same priority, the vertex with the greatest bottom level is chosen for τ_r (likewise for τ_u) to be scheduled first to favor the successors that have the longest path from τ_r to the exit vertex. Also, an optimization could be performed when calling *update_priority_values*(G); indeed, after each cluster allocation, only the tlevels of the successors of τ_r need to be recomputed instead of those of the whole graph.

Theorem 1. *The time complexity of Algorithm 10 (BDSC) is $\mathcal{O}(n^3)$, n being the number of vertices in Graph G.*

Proof. In the "while" loop of BDSC, the most expensive computation is the function *end_idle_cluster* used in $find\_cluster$ that locates an existing cluster suitable to allocate there Task τ; such reuse intends to optimize the use of the

ALGORITHM 10: BDSC scheduling Graph G_u, under processor and memory bounds P and M

```
function BDSC(G_u, P, M)
  if (P ≤ 0) then
    return error('Not enough processors', G_u) ;
  G = graph_copy(G_u);
  foreach  $\tau_i$ ∈ vertices(G)
    priority($\tau_i$) = tlevel($\tau_i$, G) + blevel($\tau_j$, G);
  UT = vertices(G);
  RL = {$\tau$ ∈ UT / predecessors($\tau$, G) = ∅};
  URL = UT − RL;
  clusters = ∅;
  while UT ≠ ∅
    $\tau_r$ = select_task_with_highest_priority(RL);
    ($\tau_m$, min_tlevel) = tlevel_decrease($\tau_r$, G);
    if ($\tau_m$ ≠ $\tau_r$ ∧ MCW($\tau_r$, cluster($\tau_m$), M)) then
      $\tau_u$ = select_task_with_highest_priority(URL);
      if (priority($\tau_r$) < priority($\tau_u$)) then
        if (¬DSRW($\tau_r$, $\tau_u$, clusters, G)) then
          if (PCW(clusters, P)) then
            $\kappa$ = new_cluster(clusters);
            allocate_task_to_cluster($\tau_r$, $\kappa$, G);
          else
            if (¬find_cluster($\tau_r$, G, clusters, P, M)) then
              return error('Not enough memory', G_u);
      else
        allocate_task_to_cluster($\tau_r$, cluster($\tau_m$), G);
        edge_cost($\tau_m$, $\tau_r$) = 0;
    else if (¬find_cluster($\tau_r$, G, clusters, P, M)) then
      return error('Not enough memory', G_u);
    update_priority_values(G);
    UT = UT−{$\tau_r$};
    RL = update_ready_set(RL, $\tau_r$, G);
    URL = UT−RL;
  clusters(G) = clusters;
  return G;
end
```

limited of processors. Its complexity is proportional to

$$\sum_{\tau_\kappa \in vertices(G)} |successors(\tau_\kappa, G)|,$$

which is of worst case complexity $\mathcal{O}(n^2)$. Thus the total cost for n iterations of the “while” loop is $\mathcal{O}(n^3)$. $\square$

ALGORITHM 11: Attempt to allocate cluster in clusters for Task τ in Graph G, under processor and memory bounds P and M, returning true if successful

```
function find_cluster($\tau$, G, clusters, P, M)
  MCW_idle_clusters =
              {$\kappa \in$ end_idle_clusters($\tau$, G, clusters, P) /
               MCW($\tau$, $\kappa$, M)};
  if (MCW_idle_clusters $\neq \emptyset$) then
    $\kappa$ = choose_any(MCW_idle_clusters);
    allocate_task_to_cluster($\tau$, $\kappa$, G);
  else if (PCW(clusters, P)) then
    allocate_task_to_cluster($\tau$, new_cluster(clusters), G);
  else
    MCW_clusters = {$\kappa \in$ clusters / MCW($\tau$, $\kappa$, M)};
    MCW_clusters_min = argmin$_{\kappa\ \in\ \text{MCW\_clusters}}$ cluster_time($\kappa$);
    if (MCW_clusters_min $\neq \emptyset$) then
      $\kappa$ = choose_any(MCW_clusters_min);
      allocate_task_to_cluster($\tau$, $\kappa$, G);
    else
      return false;
  return true;
end

function error(m, G)
  clusters(G) = $\emptyset$;
  return G;
end
```

Even though BDSC's worst case complexity is larger than DSC's, which is $\mathcal{O}(n^2log(n))$ [5], it remains polynomial, with a small exponent. Our experiments (see Section 5) showed this theoretical slowdown is indeed not a significant factor in practice.

4. BDSC-Based Hierarchical Parallelization

In this section, we detail how BDSC can be used, in practice, to schedule applications. We show how to build from an existing program source code what we call a Sequence Dependence Graph (SDG), which will play the role of DAG G above, how to then generate the numerical cost of vertices and edges in SDGs and how to perform what we call Hierarchical Scheduling (HBDSC) for SDGs. We use PIPS to illustrate how these new ideas can be integrated in an optimizing compilation platform.

PIPS [7] is a powerful, source-to-source compilation framework initially developed at MINES ParisTech in the 1990s. Thanks to its open-source nature, PIPS has been used by multiple partners over the years for analyzing and transforming C and Fortran programs, in particular when targeting vector, parallel and hybrid architectures. Its advanced static analyses provide sophisticated information about possible program behaviors, including use-def chains, preconditions, transformers, in-out array regions and worst-case code complexities. All information within PIPS is managed via specific APIs that are automatically provided from data structure specifications written with the Newgen domain specific language [9].

4.1. Hierarchical Sequence Dependence DAG Mapping

PIPS represents user code as abstract syntax trees. We define a subset of its grammar in Figure 5, limited to the statements S at stake in this paper. $\mathtt{E}_{cond}$, $\mathtt{E}_{lower}$ and $\mathtt{E}_{upper}$ are expressions, while `I` is an identifier. The semantics of these constructs is straightforward. Note that, in PIPS, assignments are seen as function calls, where left hand sides are parameters passed by reference. We use the notion of control flow graph `CFG` to represent parallel code.

```
S ∈ Statement ::= sequence(S_1;....;S_n) |
                  test(E_cond,S_t,S_f) |
                  forloop(I, E_lower, E_upper, S_body) |
                  call |
                  CFG(C_entry, C_exit)
C ∈ Control ::= control(S, L_succ, L_pred)
L ∈ Control*
```

Figure 5: Abstract syntax tree `Statement` syntax

We assume that each task τ includes a statement $S = task\_statement(\tau)$, which corresponds to the code it runs when scheduled.

In order to partition into tasks real applications, which include loops, tests and other structured constructs[5], into dependence DAGs, our approach is to first build a Sequence Dependence DAG (SDG) which will be the input for the BDSC algorithm. Then, we use the code presented in form of an AST to define a hierarchical mapping function, that we call H, to map each sequence statement of the code to its SDG. H is used for the input of the HBDSC algorithm. We present in this section what SDGs are and how an H is built upon them.

4.1.1. Sequence Dependence DAG

A Sequence Dependence DAG (SDG) G is a data dependence DAG where task vertices τ are labeled with statements, while control dependences are encoded in the abstract syntax trees of statements. Any statement S can label a DAG vertex, i.e. each vertex τ contains a statement S, which corresponds

[5] In this paper, we only handle structured parts of a code, i.e., the ones that do not contain `goto` statements. Therefore, within this context, PIPS implements control dependences in its IR since it is equivalent to an AST (for structured programs, CDG and AST are equivalent).

to the code it runs when scheduled. We assume that there exist two functions *vertex_statement* and *statement_vertex* such that, on their respective domains of definition, they satisfy $S =$ *vertex_statement(*τ*)* and *statement_vertex(S,G)* $= \tau$. In contrast to the usual program dependence graph defined in [10], an SDG is thus not built only on simple instructions, represented here as *call* statements; compound statements such as test statements (both true and false branches) and loop nests may constitute indivisible vertices of the SDG.

To compute the SDG G for a sequence $S = \texttt{sequence}(S_1; S_2; .....; S_m)$, one may proceed as follows. First, a vertex τ_i for each statement S_i in S is created; for loop and test statements, their inner statements are recursively traversed and transformed into SDGs. Then, using the Data Dependence Graph D, dependences coming from all the inner statements of each S_i are gathered to form cumulated dependences. Finally, for each statement S_i, we search for other statements S_j such that there exists a cumulated dependence between them and add a dependence edge *(*τ_i*,*τ_j*)* to G. G is thus the quotient graph of D with respect to the dependence relation.

Figure 7 illustrates the construction, from the DDG given in Figure 6 (right), the SDG of the C code (left). The figure contains two SDGs corresponding to the two sequences in the code; the body $S0$ of the first loop (in blue) has also an SDG $G0$. Note how the dependences between the two loops have been deduced from the dependences of their inner statements (their loop bodies). These SDGs and their printouts have been generated automatically with PIPS.

4.1.2. Hierarchical SDG Mapping

We presented above how sequences of statements can be transformed into SDGs. This section suggests to handle the other types of statements, such as loops and tests, by adopting a hierarchical view of the source code, encoded in a new data structure. A *hierarchical SDG mapping* function H maps each statement S to an SDG $G = H(S)$ if S is a sequence statement; otherwise G is equal to $\perp$. In the figure 7 we already saw, a hierarchical SDG mapping H is illustrated. Here, $H(S)$ is G, while, for the SDG $G0$ corresponding to the body $S0$ of the loop, one has $G0 = H(S0)$. These SDGs have been generated automatically with PIPS; we use the Graphviz tool for pretty printing [11].

Our introduction of the notions of SDGs and hierarchical mappings is motivated by the following observations, which also support our design decisions:

1. The true and false statements of a test are control dependent upon the condition of the test statement, while every statement within a loop (i.e., statements of its body) is control dependent upon the loop statement header. If we define a *control area* as a set of statements transitively linked by the control dependence relation, our SDG construction process insures that the control area of the statement of a given vertex is in the vertex. This way, we keep all the control dependences of a task in our SDG within itself.
2. We decided to consider test statements as single vertices in the SDG to

```
void main()
{
 int a[10], b[10];
 int i,d,c;
 //S
 {
  c=42;
  for(i=1;i<=10;i++){
    a[i] = 0;
  }
  for(i=1;i<=10;i++){
   //S0
   a[i] = bar(a[i]);
   d = foo(c);
   b[i] = a[i]+d;
  }
 }
 return;
}
```

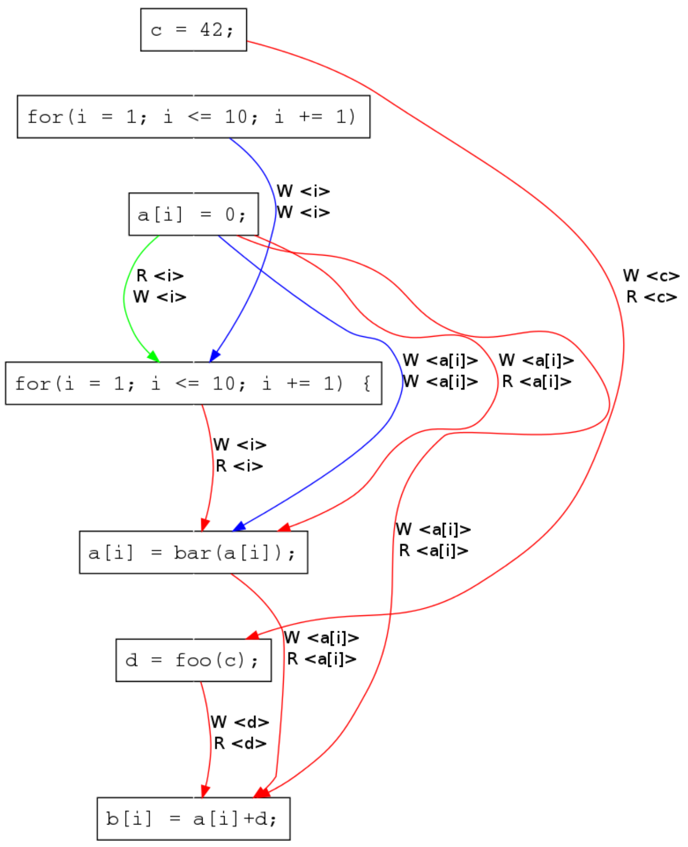


Figure 6: Example of a C code (left) and the DDG D of its internal S sequence (right, where red denotes true data-flow dependences, blue, output dependences, and green, anti-dependences)

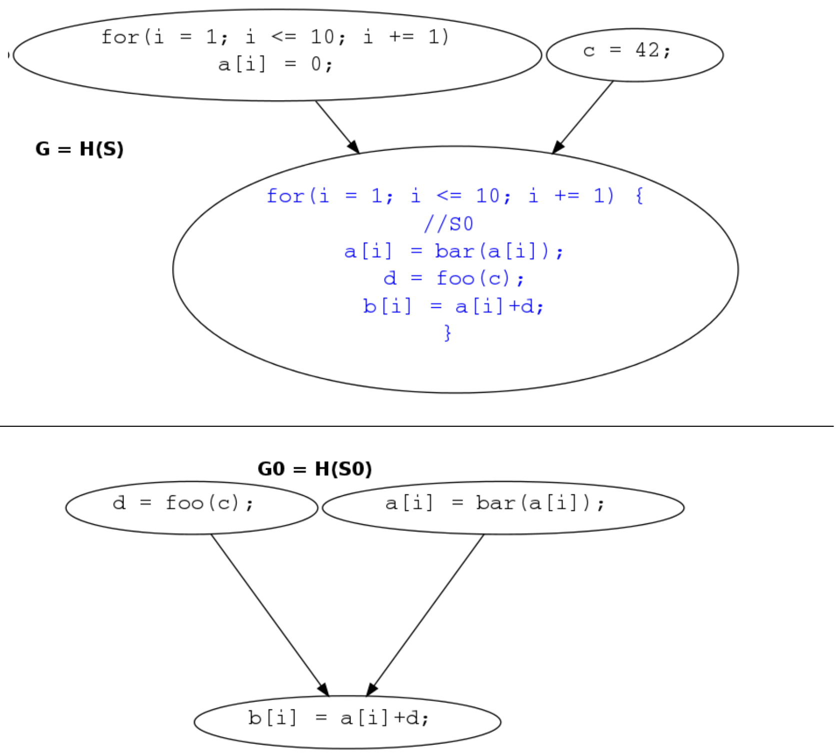
ensure that they are scheduled on one cluster[6], which guarantees the execution of the inner code (true or false statements), whichever branch is taken, on this cluster.

3. We do not group successive simple *call* instructions into a single "basic block" vertex in the SDG in order to let BDSC fuse the corresponding statements so as to maximize parallelism and minimize communications. Note that PIPS performs interprocedural analyses, which will allow call sequences to be efficiently scheduled whether these calls represent trivial assignments or complex function calls.

4.2. Cost Models Generation

Since the volume of data used or communicated by SDG tasks are key factors in the BDSC scheduling process, we need to as precisely as possible assess

[6]A cluster is a logical entity which will correspond to one process or thread.



```
for(i = 1; i <= 10; i += 1)
   a[i] = 0;
```

```
c = 42;
```

```
for(i = 1; i <= 10; i += 1) {
   //S0
   a[i] = bar(a[i]);
   d = foo(c);
   b[i] = a[i]+d;
}
```

```
d = foo(c);
```

```
a[i] = bar(a[i]);
```

```
b[i] = a[i]+d;
```

Figure 7: SDGs of S (top) and $S0$ (bottom) computed from the DDG (see the right of Figure 6); S and $S0$ are specified in the left of Figure 6

this information. PIPS provides an intra- and inter-procedural analysis of array data flow called *regions analysis* [12] that computes dependences for each array element access. Sets of array elements are gathered into array regions, which are represented by convex polyhedra expressions over the variables values in the current memory store. For each statement S, two types of sets R of regions r are considered in this paper: $read\_regions(S)$ and $write\_regions(S)$ contain the array elements respectively read and written by S. The two types of regions are distinguished by a label, either `R` or `W`. For instance, in Figure 8, PIPS is able to infer sets of regions such as:

$$R_{wa} = \{\mathtt{W}, a(\phi_1)/1 \leq \phi_1, \phi_1 \leq 10\}$$
$$R_{ra} = \{\mathtt{R}, a(\phi_1)/6 \leq \phi_1, \phi_1 \leq 21\}$$

where the write regions R_{wa} of Array a, modified in the first loop, are the array elements of a with indices in the interval [1,10]. The read regions R_{ra} of Array a in the second loop represents the elements with indices in [6,21].

```
                // {R, a(φ1)/1 ≤ φ1, φ1 ≤ 10}
                // {W, a(φ1)/1 ≤ φ1, φ1 ≤ 10}
                // {W, b(φ1)/1 ≤ φ1, φ1 ≤ 10}
for(i = 1; i <= 10; i++) {
  a[i] = f(a[i]);
  b[i] = 42;
}
                // {R, a(φ1)/6 ≤ φ1, φ1 ≤ 21}
                // {W, b(φ1)/6 ≤ φ1, φ1 ≤ 20}
for(j = 6; j <= 20; j++)
  b[j]=g(a[j],a[j+1]);
```

Figure 8: Example of array region analysis

4.2.1. From Convex Polyhedra to Ehrhart Polynomials

Our analysis uses the following operations on sets R_i of regions (convex polyhedra)[7]:

1. *regions_intersection(R_1,R_2)* is a set of regions; each region r in this set is the intersection of two regions $r_1 \in R_1$ and $r_2 \in R_2$ referencing the same array. The convex polyhedron of r is the intersection of the two convex polyhedra of r_1 and r_2, which is also a convex polyhedron.
2. *regions_union(R_1,R_2)* is a set of regions; each region r in this set is the union of two regions $r_1 \in R_1$ and $r_2 \in R_2$ with the same reference. The convex polyhedron of r is the union of the two convex polyhedra of r_1 and r_2, which is not necessarily a convex polyhedron. An approximated convex hull is thus computed in order to return the smallest enclosing polyhedron.

Since we are interested in the size of these regions to precisely assess communication costs and memory requirements, we compute Ehrhart polynomials [13], which represent the number of integer points contained in a given parameterized polyhedron, from this region. To manipulate these polynomials, we use various operations using the Ehrhart API provided by the polylib library [14].

Communication Edge Cost To assess the communication cost between two SDG vertices, τ_1 as source and τ_2 as sink vertices, we rely on the number of bytes involved in true data-flow dependences, of type "read after write" (RAW), using the read and write regions as follows:

R_{w1} = *write_regions(vertex_statement(τ_1))*
R_{r2} = *read_regions(vertex_statement(τ_2))*

[7]Note that regions must be defined with respect to a common memory store for these operations to be properly defined.

$$edge\_cost\_bytes(\tau_1,\, \tau_2) = \sum_{r \in regions\_intersection(R_{w1}, R_{r2})} ehrhart(r)$$

In practice, in our experiments (see Section 5), in order to compute communication times, this polynomial, which represents the message size to communicate, expressed in number of bytes, is multiplied by the transfer time of one byte (β), to which is then added the latency time (α). These two coefficients are dependent on the specific target machine. We note:

$$edge\_cost(\tau_1,\, \tau_2) = \alpha + \beta \times edge\_cost\_bytes(\tau_1,\, \tau_2)$$

Local Storage Task Data To provide an estimation of the volume of data used by each vertex τ, we use the number of bytes of data read and written by the task statement, via the following definitions :

S = *task_statement*(τ);
task_data(τ) = *data_merge(read_regions(S), write_regions(S))*

where we define *data_merge* and *data_size* as follows:

data_merge(R_1, R_2) = *regions_union*(R_1, R_2)
data_size(R) = $\sum_{r \in R} ehrhart(r)$

Execution Task Time In order to determine an average execution time for each vertex in the SDG, we use a static execution time approach based on a program complexity analysis provided by PIPS. There, each statement S is automatically labeled with an expression, represented by a polynomial over program variables *complexity_estimation*(S), that denotes an estimation of the execution time of this statement, assuming that each basic operation (addition, multiplication...) has a fixed, architecture-dependent execution time. This sophisticated static complexity analysis is based on inter-procedural information such as preconditions. Using this approach, one can define *task_time* as:

task_time(τ) = *complexity_estimation(task_statement*(τ))

4.2.2. From Polynomials to Values

We have just seen how to represent cost, data and time information in terms of polynomials; yet, running BDSC requires actual values. This is particularly a problem when computing tlevels and blevels, since cost and time are cumulated there. We decided to convert cost information into time by assuming that communication times are proportional to costs, which amounts in particular to setting communication latency to zero[8].

When program variables used in the above-defined polynomials are numerical values, each polynomial is a constant; this happens to be the case for one of our applications, ABF. However, when input data are unknown at compile time (as for the Harris application), we suggest to use a very simple heuristic to

[8]This assumption is validated by our experimental results, and the fact that our data arrays are large.

approximate the values of the polynomials. When all polynomials at stake are monomials on the same base, we simply keep the coefficient of these monomials. Even though this heuristics appears naive at first, it actually is quite useful in the Harris application: Table 1 shows the complexities and time estimation generated for each function of Harris using PIPS default operation cost model, where the `sizeN` and `sizeM` variables represent the input image size.

Function	**Complexity (polynomial)**	**Time estimation**
InitHarris	9 × `sizeN` × `sizeM`	9
SobelX	60 × `sizeN` × `sizeM`	60
SobelY	60 × `sizeN` × `sizeM`	60
MultiplY	20 × `sizeN` × `sizeM`	20
Gauss	85 × `sizeN` × `sizeM`	85
CoarsitY	34 × `sizeN` × `sizeM`	34
One image transfer	4 × `sizeN` × `sizeM`	4

Table 1: Execution and communication time estimations for Harris using PIPS default cost models

The general case deals with polynomials that are functions of many variables, as is the case in equake, where they depend on variables such as `ARCHelems` or `ARCHnodes`. In such cases, we suggest to first instrument the input sequential code and run it once in order to obtain the numerical values of the polynomials. The instrumented code contains the initial user code plus instructions that compute the values of the cost polynomials for each statement. BDSC is then applied, using this cost information, to yield the final parallel program. Note that this approach is sound since BDSC ensures that the value of a variable (and thus a polynomial) is the same, whichever scheduling is used. Of course, this approach will work well, as our experiments suggest, when a program performance does not change when some part of its input parameters are modified; this is the case for many signal processing applications, where performance is mostly a function of structure parameters such as image size, and is independent of the actual signal (pixel) values upon which the program acts.

We show an example of this final case using a part of the instrumented equake code[9] in Figure 9. The added instrumentation instructions are `fprintf` statements, the second parameter of which represents the statement number of the following statement, and the third, the value of its execution time for task time instrumentation. For edge cost instrumentation, the second parameter is the number of the incident statements of the edge, and the third, the edge cost polynomial. After execution of the instrumented code, the numerical results of the polynomials are printed in the file `instrumented_equake.in`. This file will be an entry for the PIPS implementation of BDSC.

[9]We do not show the instrumentation on the statements inside the loops for readability purposes.

```
FILE * finstrumented = fopen("instrumented_equake.in", "w");
fprintf(finstrumented,
       "task_time 62 = %ld\n", 179 * ARCHelems + 3);
for (i = 0; i < ARCHelems; i++){
  for (j = 0; j < 4; j++)
    cor[j] = ARCHvertex[i][j];
}
fprintf(finstrumented,
       "task_time 163 = %ld\n", 20 * ARCHnodes + 3);
for(i = 0; i <= ARCHnodes-1; i += 1)
  for(j = 0; j <= 2; j += 1)
    disp[disptplus][i][j] = 0.0;
fprintf(finstrumented,
       "edge_cost 163 → 166 = %ld\n", ARCHnodes * 9);
fprintf(finstrumented,
       "task_time 166 = %ld\n", 110 * ARCHnodes + 106);
smvp_opt(ARCHnodes, K,
         ARCHmatrixcol, ARCHmatrixindex,
         disp[dispt], disp[disptplus]);
```

Figure 9: Excerpt of instrumented equake ($S0$ is the inner loop sequence)

4.3. Hierarchical Scheduling (HBDSC)

Now that all the information needed by the basic version of BDSC presented above has been gathered, we detail in Algorithm 12 how we suggest to adapt it to different SDGs linked hierarchically via the mapping function H introduced above in order to eventually generate nested parallel code when possible. We adopt in this section the graph-based parallel programming model since it offers the freedom to implement arbitrary parallel patterns and since SDGs implement this model. Therefore, we use the `CFG` construct of the PIPS IR to encode the generated parallel code.

4.3.1. Recursive Top-Down Scheduling

Hierarchically scheduling a given statement S of SDG $H(S)$ in a cluster κ is seen here as the definition of a *hierarchical schedule* σ which maps each inner statement s of S to $\sigma(s) = (s', \kappa, n)$. If there are enough processor and memory resources to schedule S using BDSC, (s', κ, n) is a triplet made of a *parallel statement* $s' = \mathit{parallel}(\sigma(s))$, the cluster $\kappa = \mathit{cluster}(\sigma(s))$ where s is being allocated and the number $n = \mathit{nbclusters}(\sigma(s))$ of clusters the inner scheduling of s' requires. Otherwise, scheduling is impossible, and the program stops. In a scheduled statement, all sequences are replaced by parallel `CFG` statements.

A successful call to the *HBDSC($S, H, \kappa, P, M, \sigma$)* function defined in Algorithm 12, which assumes that P is strictly positive, yields a new version of σ that schedules S into κ and takes into account all inner statements of S;

only P clusters, with a data size at most M each, can be used for scheduling. $\sigma[S \rightarrow (S', \kappa, n)]$ is the function equal to σ except for S, where its value is (S', κ, n). H is the function that yields an SDG for each S to be scheduled using BDSC.

ALGORITHM 12: BDSC-based update of Schedule σ for Statement S of SDG $H(S)$, with P and M constraints

```
function HBDSC(S, H, $\kappa$, P, M, $\sigma$)
  switch (S)
    case call:
      return $\sigma$[S $\rightarrow$ (S, $\kappa$, 0)];
    case sequence(S$_1$;...;S$_n$):
      G$_{seq}$ = closure(S, H(S))
      G$'$ = BDSC(G$_{seq}$, P, M, $\sigma$);
      iter = 0;
      do
        $\sigma'$ = HBDSC_step(G$'$, H, $\kappa$, P, M, $\sigma$);
        G = G$'$;
        G$'$ = BDSC(G$_{seq}$, P, M, $\sigma'$);
        if (clusters(G$'$) = $\emptyset$) then
          abort('Unable to schedule');
        iter++;
        $\sigma$ = $\sigma'$;
      while (completion_time(G$'$) < completion_time(G) $\wedge$
             |clusters(G$'$)| $\leq$ |clusters(G)| $\wedge$
             iter $\leq$ MAX_ITER)
      return $\sigma$[S $\rightarrow$(dag_to_cfg(G), $\kappa$, |clusters(G)|)];
    case forloop(I, E$_{lower}$, E$_{upper}$, S$_{body}$):
      $\sigma'$ = HBDSC(S$_{body}$, H, $\kappa$, P, M, $\sigma$);
      (S$'_{body}$, $\kappa_{body}$, nbclusters$_{body}$) = $\sigma'$(S$_{body}$);
      return $\sigma'$[S $\rightarrow$ (forloop(I, E$_{lower}$, E$_{upper}$, S$'_{body}$),
                      $\kappa$, nbclusters$_{body}$)];
    case test(E$_{cond}$, S$_t$, S$_f$):
      $\sigma$ = HBDSC(S$_t$, H, $\kappa$, P, M, $\sigma'$);
      $\sigma''$ = HBDSC(S$_f$, H, $\kappa$, P, M, $\sigma'$);
      (S$'_t$, $\kappa_t$, nbclusters$_t$) = $\sigma''$(S$_t$);
      (S$'_f$, $\kappa_f$, nbclusters$_f$) = $\sigma''$(S$_f$);
      return $\sigma''$[S $\rightarrow$ (test(E$_{cond}$, S$'_t$, S$'_f$),
                       $\kappa$, max(nbclusters$_t$, nbclusters$_f$))];
end
```

Our approach is top-down in order to yield tasks that are as coarse grained as possible when dealing with sequences. In the *HBDSC* function, we distinguish four cases of statements. First, the constructs of loops[10] and tests are simply traversed, scheduling information being recursively gathered in different SDGs. Then, for a call statement, there is no descent in the call graph, the call statement is returned. In order to handle the corresponding call function, one has to treat separately the different functions. Next, for a sequence S, one first accesses its SDG and computes a closure of this DAG, G_{seq}, using the function *closure*. The purpose of the *closure* function (see [15]) is to provide a self-contained version of $H(S)$: $H(S)$ is completed with a set of entry vertices and edges in order to represent the dependences coming from outside S, yielding the closed SDG G_{seq}. Finally, G_{seq} is scheduled using BDSC to generate a scheduled SDG G'.

The hierarchical scheduling process is then recursively performed, to take into account inner statements of S, within Function *HBDSC_step* defined in Algorithm 13 on each statement s of each task of G'. There, G' is traversed along a topological sort-ordered descent using the function *topsort*(G') yields a list of stages of computation, each *cluster_stage* being a list of independent lists L of tasks τ, one L for each cluster κ generated by BDSC for this particular stage in the topological order.

The recursive hierarchical scheduling via the function *HBDSC*, within the function *HBDSC_step*, of each statement $s = \mathit{vertex\_statement}(\tau)$ may take advantage of at most P' available clusters, since $|\mathit{cluster\_stage}|$ clusters are already reserved to schedule the current stage *cluster_stage* of tasks for Statement S. It yields a new scheduling function $\sigma_{\mathtt{s}}$. Otherwise, if no clusters are available, all inner statements of s are scheduled on the same cluster as their parent, κ. We use the straightforward function *same_cluster_mapping* (not provided here) to affect recursively $(s_e, \kappa, 0)$ to $\sigma(s_e)$ for each inner s_e of s.

Figure 10 illustrates the various entities involved in the computation of such a scheduling function. Note that one needs to be careful in *HBDSC_step* to ensure that each rescheduled inner statement s is allocated a number of clusters consistent with the one used when computing its parallel execution time; we check the condition $\mathit{nbclusters}'_{\mathtt{s}} \geq \mathit{nbclusters}_{\mathtt{s}}$, which ensures that the parallelism assumed when computing time complexities within s remains available.

Cluster allocation information for each inner statement s whose vertex in G' is τ is maintained in σ via the recursive call to *HBDSC*, this time with the current cluster $\kappa = \mathit{cluster}(\tau)$. For the non-sequence constructs in Function *HBDSC*, cluster information is set to κ, the current cluster.

The scheduling of a sequence yields a parallel CFG statement; we use the function *dag_to_cfg*(G) that returns a PIPS control flow graph statement S_{cfg} from the SDG G, where the vertices of G are the statement control vertices s_{cfg} of S_{cfg}, and the edges of G constitute the list of successors $\mathtt{L}_{succ}$ of s_{cfg}

[10]Regarding parallel loops, since we adopt the task parallelism paradigm, note that, initially, it may be useful to apply the tiling transformation and then perform full unrolling of the outer loop (we give more details in the protocol of our experiments in Section 5.1). This way, the input code contains more potentially parallel tasks resulting from the initial (parallel) loop.

ALGORITHM 13: Iterative hierarchical scheduling step for DAG fixpoint computation

```
function HBDSC_step(G', H, κ, P, M, σ)
  foreach cluster_stage ∈ topsort(G')
    P' = P − |cluster_stage|;
    foreach L ∈ cluster_stage
      nbclusters_L = 0;
      foreach τ ∈ L
        s = vertex_statement(τ);
        if (P' ≤ 0) then
          σ = σ[s → same_cluster_mapping(s, κ, σ)];
        else
          nbclusters_s = (s ∈ domain(σ)) ?
                                   nbclusters(σ(s)) : 0;
          σ_s = HBDSC(s, H, cluster(τ), P', M, σ);
          nbclusters'_s = nbclusters(σ_s(s));
          if (nbclusters'_s ≥ nbclusters_s ∧
               task_time(τ, σ) ≥ task_time(τ, σ_s) then
            nbclusters_s = nbclusters'_s;
            σ = σ_s;
          nbclusters_L = max(nbclusters_L, nbclusters_s);
      P' −= nbclusters_L;
  return σ;
end
```

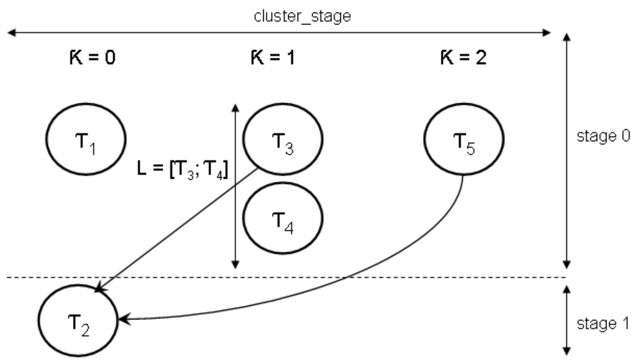


Figure 10: $topsort(G)$ for the hierarchical scheduling of sequences

while the list of predecessors $\mathtt{L}_{pred}$ of s_{cfg} is deduced from $\mathtt{L}_{succ}$. Note that vertices and edges of G are not changed before and after scheduling; however, information of scheduling is saved in σ.

4.3.2. Iterative Scheduling for Resource Optimization

BDSC is called in *HBDSC* before inner statements are hierarchically scheduled. However, a unique pass over inner statements could be suboptimal, since parallelism may exist within inner statements. It may be discovered by later recursive calls to *HBDSC*. Yet, if this parallelism had been known ahead of time, previous values of *task_time* used by BDSC would have been possibly smaller, which could have had an impact on the higher-level scheduling. In order to address this issue, our hierarchical scheduling algorithm iterates the top down pass *HBDSC_step* on the new DAG G' in which BDSC takes into account these modified task complexities; iteration continues while G' provides a smaller DAG schedule length than G and the iteration limit *MAX_ITER* has not been reached. We compute the completion time of the DAG G, as follows:

$$completion\_time(G) = \max_{\kappa \in clusters(G)} cluster\_time(\kappa)$$

One constraint due to the iterative nature of the hierarchical scheduling is that, in BDSC, zeroing cannot be made between the entry vertices and their successors. This keeps an independence in terms of allocated clusters between the different levels of the hierarchy. Indeed, at a higher level, for S, if we assume that we have scheduled the parts S_e inside (hierarchically) S; attempting to reschedule S iteratively cancels the precedent schedule of S but maintains the schedule of S_e and vice versa. Therefore, for each sequence, we have to deal with a new set of clusters; and thus, zeroing cannot be made between these entry vertices and their successors.

Note that our top-down, iterative, hierarchical scheduling approach also helps dealing with limited memory resources. If BDSC fails at first because not enough memory is available for a given task, the *HBDSC_step* function is nonetheless called to schedule nested statements, possibly loosening up the memory constraints by distributing some of the work on less memory-challenged additional clusters. This might enable the subsequent call to BDSC to succeed.

4.3.3. Parallel Cost Models

In Section 4.2, we present the sequential cost models usable in the case of sequential codes, i.e, for each first call to BDSC. When an inner statement S_e of S is parallelized, the parameters *task_time*, *task_data* and *edge_cost* are modified for S_e and thus for S. Thus, hierarchical scheduling must use extended definitions of *task_time*, *task_data* and *edge_cost* for tasks τ using statements $S =$ *vertex_statement*(τ) that are `CFG` statements, extending the definitions provided in Section 4.2, which still apply to non-CFG statements. For such a case, we assume that BDSC and other relevant functions take σ as an additional argument to access the scheduling result associated to statement sequences and handle the modified definitions of *task_time*, *edge_cost* and *task_data*. These functions can be found in [15].

4.3.4. Complexity of HBDSC Algorithm

Theorem 2. *The time complexity of Algorithm 12 (HBDSC) over Statement S is $\mathcal{O}(k^n)$, where n is the number of call statements in S and k a constant* *greater than 1.*

Proof. Let $t(l)$ be the worst-case time complexity for our hierarchical scheduling algorithm on the structured statement S of hierarchical level[11] l. Time complexity increases significantly only in sequences, loops and tests being simply managed by straightforward recursive calls of *HBDSC* on inner statements. For a sequence S, $t(l)$ is proportional to the time complexity of BDSC followed by a call to *HBDSC_step*; the proportionality constant is k =*MAX_ITER* (supposed to be greater than 1).

The time complexity of BDSC for a sequence of m statements is at most $\mathcal{O}(m^3)$ (see Theorem 1). Assuming that all subsequences have a maximum number m of (possibly compound) statements, the time complexity for the hierarchical scheduling step function is the time complexity of the topological sort algorithm followed by a recursive call to *HBDSC*, and is thus $\mathcal{O}(m^2 + mt(l-1))$. Thus $t(l)$ is at most proportional to $k(m^3 + m^2 + mt(l-1)) \sim km^3 + kmt(l-1)$. Since $t(l)$ is an arithmetico-geometric series, its analytical value $t(l)$ is $\frac{(km)^l(km^3+km-1)-km^3}{km-1} \sim (km)^l m^2$. Let l_S be the level for the whole Statement S. The worst performance occurs when the structure of S is flat, i.e., when $l_S \sim n$ and m is $\mathcal{O}(1)$; hence $t(n) = t(l_S) \sim k^n$. $\square$

Even though the worst case time complexity of *HBDSC* is exponential, we expect and our experiments suggest that it behaves more tamely on actual, properly structured code. Indeed, note that $l_S \sim log_m(n)$ if S is balanced for some large constant m; in this case, $t(n) \sim (km)^{log(n)}$, showing a subexponential time complexity.

5. Experiments

The BDSC algorithm presented in this paper has been designed to offer better task parallelism extraction performance for parallelizing compilers than traditional list-scheduling techniques such as DSC. To verify its effectiveness, BDSC has been implemented in PIPS and tested on actual applications written in C. In this section, we provide preliminary experimental BDSC-vs-DSC comparison results based on the parallelization of four such applications, namely ABF, Harris, equake and IS. We chose these particular applications since they are well-known benchmarks and exhibit task parallelization that we hope our approach will be able to take advantage of. They are: (1) ABF (Adaptive Beam Forming), a 1,065-line program that performs adaptive spatial radar signal processing [16]; (2) Harris, a 105-line image processing corner detector [17]; (3) the 1,432-line SPEC benchmark equake [18], which is used in the finite element simulation of seismic wave propagation; and (4) Integer Sort (IS), one of the eleven benchmarks in the NAS Parallel Benchmarks suite [19], with 1,076 lines.

[11]Levels represent the hierarchy structure between statements of the AST and are counted up from leaves to the root.

5.1. Protocol

We have extended PIPS with our implementation in C of BDSC-based hierarchical scheduling. To compute the static execution time and communication cost estimates needed by BDSC, we relied upon the PIPS run time complexity analysis and a more realistic, architecture-dependent communication cost matrix (Table 1 was computed using the simpler PIPS default cost models). For each code S of our four test application, PIPS performed automatic parallelization, applying our hierarchical scheduling process $hierarchical\_schedule(S, P, M, \perp)$ (using either BDSC or DSC) on these sequential programs to yield σ. PIPS automatically generated an OpenMP [20] version from the scheduled SDGs in $\sigma(S)$, using `omptask` directives; another version, in MPI [21], was generated from the scheduled SDGs. We also applied DSC in the hierarchical scheduling process of these applications and generated the corresponding OpenMP and MPI codes. Compilation times for these applications were quite reasonable, the longest (equake) being 84 seconds. In this last instance, most of the time (79 seconds) was spent by PIPS to gather semantic information such as regions, complexities and dependences; our prototype implementation of BDSC is only responsible for the remaining 5 seconds.

We ran all these parallelized codes on two shared and distributed memory computing systems. To increase available coarse-grain task parallelism in our test suite, we have used both unmodified and modified versions of our applications. We tiled and fully unrolled the four most costly loops in ABF and equake; the tiling factor for the BDSC version is the number of available processors, while we had to find the proper one for DSC, since DSC puts no constraints on the number of needed processors but returns the number of processors its scheduling requires. For Harris and IS, our experiments have looked at both tiled and untiled versions of the applications.

5.2. Experiments on Shared Memory Systems

We measured the execution time of the parallel OpenMP codes on the P = 1, 2, 4, 6 and 8 cores of a host Linux machine with a 2-socket AMD quad-core Opteron with 8 cores, with M = 16 GB of RAM, running at 2.4 GHz. Figure 11 shows the performance results of the generated OpenMP code on the two versions scheduled using DSC and BDSC on ABF and equake. The speedup data show that the DSC algorithm is not scalable, when the number of cores is increased; this is due to the generation of more clusters with empty slots than with BDSC, a costly decision given that, when the number of clusters exceeds P, they have to share the same core as multiple threads.

Figure 11 shows the performance results of the generated OpenMP code on the two versions scheduled using DSC and BDSC on ABF and equake. The speedup data show that the DSC algorithm is not scalable on these examples, when the number of cores is increased; this is due to the generation of more clusters (task creation overhead) with empty slots (poor potential parallelism and bad load balancing) than with BDSC, a costly decision given that, when the number of clusters exceeds P, they have to share the same core as multiple threads.

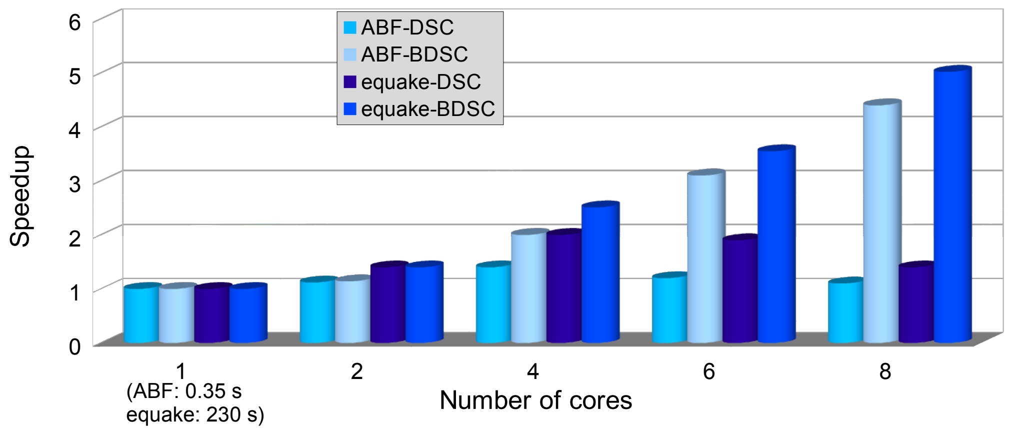


Figure 11: ABF and equake speedups with OpenMP

Figure 12 shows the hierachically scheduled SDG for Harris, generated automatically with PIPS using the Graphviz tool for three cores without tiling any loops (we used three cores because the maximum parallelism in Harris is three, as can be seen in the graph).

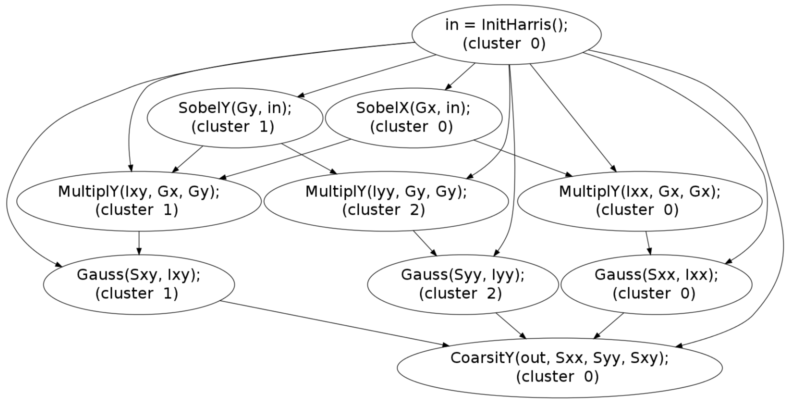

in = InitHarris(); (cluster 0)
SobelY(Gy, in); (cluster 1)
SobelX(Gx, in); (cluster 0)
MultiplY(Ixy, Gx, Gy); (cluster 1)
MultiplY(Iyy, Gy, Gy); (cluster 2)
MultiplY(Ixx, Gx, Gx); (cluster 0)
Gauss(Sxy, Ixy); (cluster 1)
Gauss(Syy, Iyy); (cluster 2)
Gauss(Sxx, Ixx); (cluster 0)
CoarsitY(out, Sxx, Syy, Sxy); (cluster 0)

Figure 12: Hierarchically scheduled SDG for Harris, using P=3 cores

Figure 13 presents the speedup obtained using $P = 3$, since the maximum parallelism in Harris is three, assuming no exploitation of data parallelism, for two parallel versions: BDSC with and BDSC without tiling of the kernel `CoarsitY` (we tiled by 3). The performance is given using three different input image sizes: 1024×1024, 2048×1024 and 2048×2048. The best speedup corresponds to the tiled version with BDSC because, in this case, the three cores are fully loaded. The DSC version (not shown in the figure) yields the same results as our versions because the code can be scheduled using three cores.

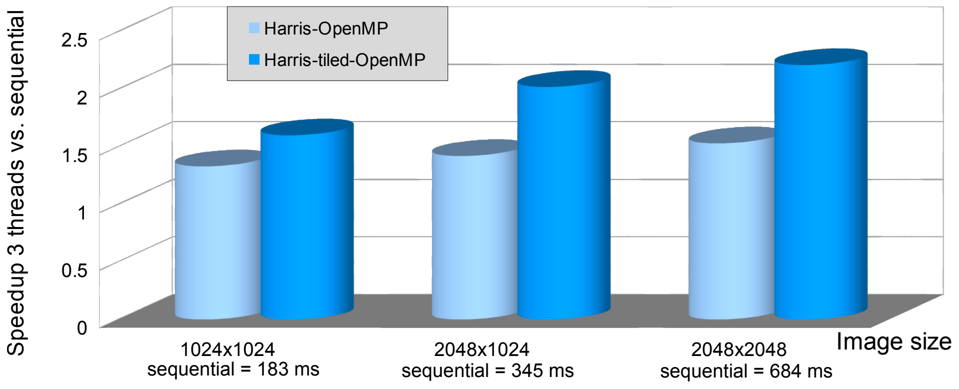


Figure 13: Speedups with OpenMP: impact of tiling (P=3)

Figure 14 shows the performance results of the generated OpenMP code on the NAS benchmark IS after applying BDSC. The maximum task parallelism without tiling in IS is two, which is shown in the first subchart; the other subcharts are obtained after tiling. The program has been run with three IS input classes (A, B and C [19]). The bad performance of our implementation for Class A programs is due to the large task creation overhead, which dwarfs the potential parallelism gains, even more limited here because of the small size of Class A data.

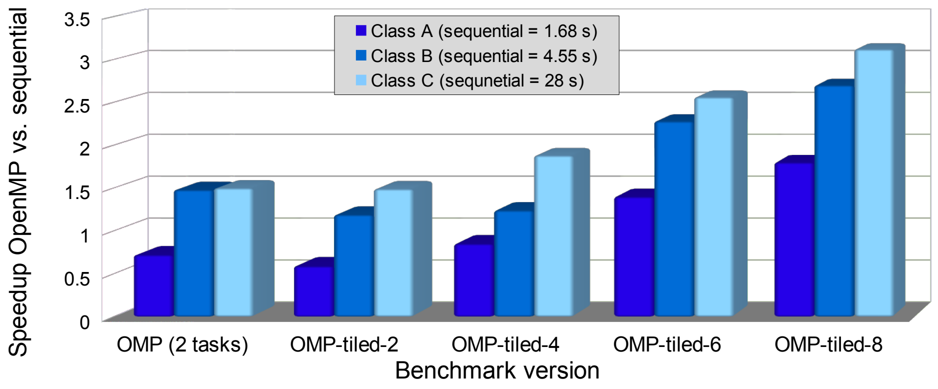


Figure 14: Speedups with OpenMP for different class sizes (IS)

5.3. Experiments on Distributed Memory Systems

We measured the execution time of the parallel codes on $P = 1$, 2, 4 and 6 processors of a host Linux machine with 6 bicore processors Intel(R) Xeon(R), with $M = 32$ GB of RAM per processor, running at 2.5 GHz. Figure 15 presents the speedups of the parallel MPI vs. sequential versions of ABF and equake using $P = 2$, 4 and 6 processors. As before, the DSC algorithm is not scalable, when the number of processors is increased, since the generation of more clusters with empty slots leads to higher process scheduling cost on processors and communication volume between them.

Figure 15 presents the speedups of the parallel MPI vs. sequential versions of ABF and equake using $P = 2$, 4 and 6 processors. As before, the DSC algorithm is not scalable, when the number of processors is increased, since the generation of more clusters with empty slots leads to higher process scheduling cost on processors and communication volume between them.

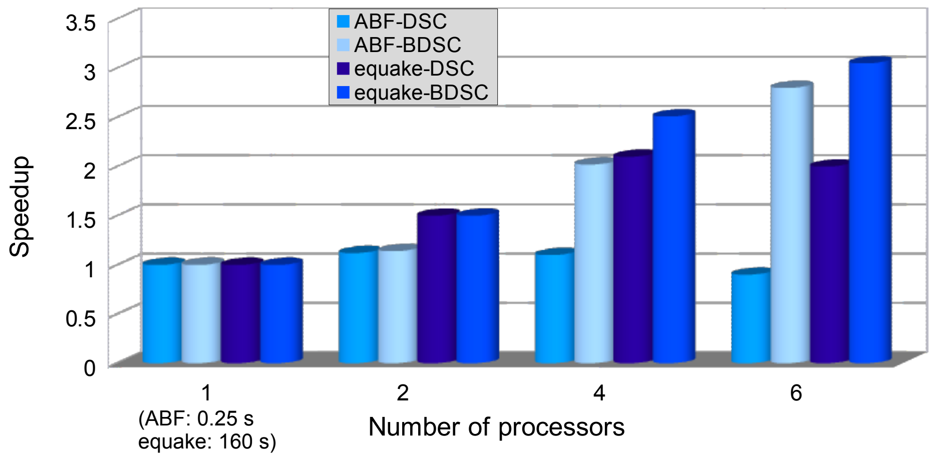


Figure 15: ABF and equake speedups with MPI

Figure 16 presents the speedups of the parallel MPI vs. sequential versions of Harris using three processors. The tiled version with BDSC gives the same result as the non-tiled version since the communication overhead is so important when the three tiled loops are scheduled on three different processors that BDSC scheduled them on the same processor; this led thus to a schedule equivalent to the one of the non-tiled version. Compared to OpenMP, the speedups decrease when the image size is increased because the amount of communication between processors increases. The DSC version (not shown on the figure) gives the same results as the BDSC version because the code can be scheduled using three processors.

Figure 17 shows the performance results of the generated MPI code on the NAS benchmark IS after application of BDSC. The same analysis as the one for OpenMP applies here, in addition to communication overhead issues.

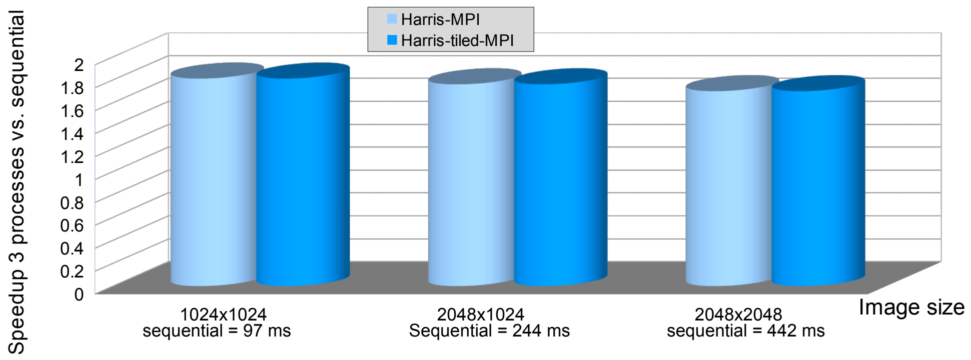


Figure 16: Speedups with MPI: impact of tiling (P=3)

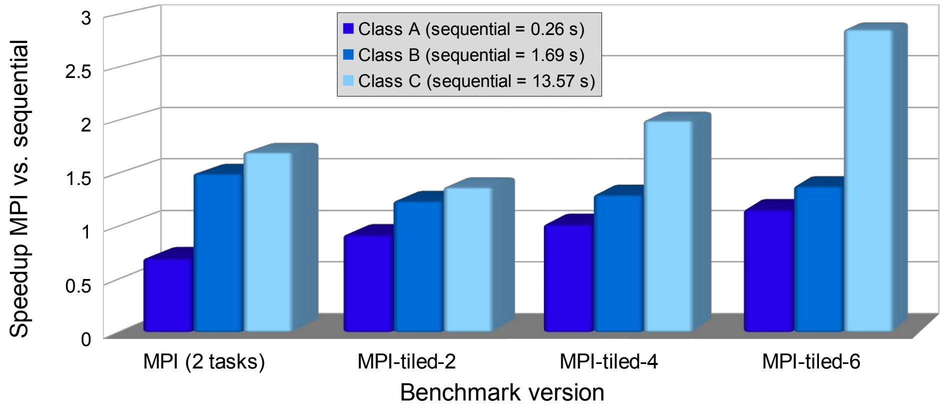


Figure 17: Speedups with MPI for different class sizes (IS)

5.4. Scheduling Robustness

Since our BDSC scheduling heuristic relies on the numerical approximations of the execution time and communication costs of tasks, one needs to assess its sensitivity over the accuracy of these estimations. Since a mathematical analysis of this issue is made difficult by the heuristic nature of BDSC and, in fact, of scheduling processes in general, we provide below experimental data that show that our approach is rather robust.

In practice, we ran multiple versions of each application using various static execution and communication cost models:

- the naive variant, in which all execution times and communications costs are supposed constant (only data dependence is enforced during the scheduling process);

Application	Language	BDSC	Naive	$\Delta = 50(\%)$	80	100	200	1000	3000
Harris	OpenMP	153	277	153	153	153	153	153	277
	MPI	303	378	303	303	303	303	303	378
ABF	OpenMP	214	321	214	230	230	246	246	297
	MPI	240	310	240	260	260	287	287	310
equake	OpenMP	58	134	58	58	80	80	80	102
	MPI	106	206	106	106	162	162	162	188
IS	OpenMP	16	35	20	20	20	25	25	29
	MPI	25	50	32	32	32	39	39	46

Table 2: Run-time sensitivity of BDSC with respect to static cost estimation (in ms for Harris and ABF; in s for equake and IS).

- the default BDSC cost models described above;
- a biased BDSC cost models, where we modulated each execution time and communication cost value randomly by at most $\Delta\%$ (the default BDSC cost models would thus correspond to $\Delta = 0$).

Our intent, with introduction of different cost models, is to assess how small to large differences to our estimation of task times and communication costs impact the performance of BDSC-scheduled parallel code. We would expect that parallelization based on the naive variant cost models would yield the worst schedules, thus motivating our use of complexity analysis for parallelization purposes if the schedules that use our default cost models are indeed better. Adding small random biases to task times and communication costs should not modify too much the schedules (to demonstrate stability), while adding larger ones might, showing the quality of the default cost models used for parallelization.

Table 2 provides, for each application (Harris, ABF, equake and IS) and execution environment (OpenMP and MPI), the worst execution time obtained within batches of about 20 runs of programs scheduled using the naive, default and biased cost models. For this last case, we only kept in the table the entries corresponding to significant values of Δ, namely those at which, for at least one application, the running time changed. So, for instance, when running ABF on OpenMP, the naive approach run time is 321 ms, while BDSC clocks at 214; adding random increments to the task communication and execution estimations provided by our cost models (Section 4.2) of up to, but not including, 80% does not change the scheduling, and thus running time. At 80%, running time increases to 230, and reaches 297 when $\Delta = 3,000$.

As expected, the naive variant always provides schedules that have the worst execution times, thus motivating the introduction of performance estimation in the scheduling process. Even more interestingly, our experiments show that one needs to introduce rather large task time and communication cost estimation errors, i.e., values of Δ, to make the BDSC-based scheduling process switch to less efficient schedules. This set of experimental data thus suggests that BDSC is a rather useful and robust heuristic, well adapted to the efficient parallelization of scientific applications.

6. Related Work

In this section, we survey the main existing list-scheduling algorithms and review the key approaches to automate the parallelization of programs using different scheduling policies; we compare them to BDSC and our BDSC-based parallelization process.

6.1. Scheduling Algorithms

Given the breadth of the literature on scheduling, we limit this presentation to heuristics that implement static list-scheduling processes. We first compare BDSC with six scheduling algorithms for a bounded number of clusters, namely HLFET, ISH, MCP, HEFT, CEFT and ELT. Then, we compare BDSC with four scheduling algorithms or techniques that regroup clusters on physical processors, i.e., LPGS, LSGP, Triplet and PYRROS.

6.1.1. Bounded Number of Clusters

The Highest Level First with Estimated Times (HLFET) [22] and Insertion Scheduling Heuristic (ISH) [23] algorithms use static blevels for ordering; scheduling is performed according to a descending order of blevels. To schedule a task, they select the cluster that offers the earliest execution time, using a non-insertion approach, i.e., not taking into account idle slots within existing clusters to insert that task. If scheduling a given task introduces an idle slot, ISH adds the possibility of inserting from the ready list tasks that can be scheduled to this idle slot. Since, in both algorithms, only blevels are used for scheduling purposes, optimal schedules for fork/join graphs cannot be guaranteed.

The Modified Critical Path (MCP) algorithm [24] uses the latest start times, i.e., the critical path length minus blevel, as task priorities. It constructs a list of tasks in an ascending order of latest start times, and searches for the cluster yielding the earliest execution using the insertion approach. As before, it cannot guarantee optimal schedules for fork/join structures.

The Heterogeneous Earliest-Finish-Time (HEFT) algorithm [25] selects the cluster that minimizes the earliest finish time using the insertion approach. The priority of a task, its upward rank, is the task blevel. Since this algorithm is based on blevels only, it cannot guarantee optimal schedules for fork/join structures.

The Constrained Earliest Finish Time (CEFT) [26] algorithm schedules tasks on heterogeneous systems. It uses the concept of constrained critical paths (CCPs) that represent the tasks ready at each step of the scheduling process. CEFT schedules the tasks in the CCPs using the finish time in the entire CCP. The fact that CEFT schedules critical path tasks first cannot guarantee optimal schedules for fork and join structures even if sufficient processors are provided.

Contrarily to the five proposals above, BDSC preserves, when no resource constraints exist, the DSC characteristics of optimal scheduling for fork/join structures, since it uses the critical path length for computing dynamic priorities, based on blevels and tlevels. HLFET, ISH and MCP guarantee that the current critical path will not increase, but they do not attempt to decrease the critical path length; BDSC decreases the length of each task DS and starts with a ready vertex to simplify the computation time of new priorities. When resource scarcity is a factor, BDSC introduces a simple, two-step heuristics for task allocation: to schedule tasks, it first searches for possible idle slots in already existing clusters and, otherwise, picks a cluster with enough memory. Our experiments suggest that such an approach provides good schedules.

Extended Latency Time (ELT) algorithm [27] assigns tasks to a parallel machine with shared memory. It uses the attribute of synchronization time instead of communication time because this does not exist in a machine with shared memory. BDSC targets both shared and distributed memory systems.

Kwork and Ahmad [28] have implemented and compared 15 scheduling algorithms. They found that, among the critical-path-based algorithms, dynamic-list algorithms such as DSC perform better than static-list ones. The insertion technique, which puts tasks within idle slots, improves scheduling. DSC does not implement this technique, while, thanks to our efficient task-to-cluster mapping strategy, which uses an insertion technique, BDSC yields better performance.

6.1.2. Cluster Regrouping on Physical Processors

The Locally Parallel-Globally Sequential (LPGS) [29] and Locally Sequential-Globally Parallel (LSGP) [30] algorithms are two techniques that, from a schedule for an unbounded number of clusters, remap the solutions to a bounded number of clusters. In LSGP, clusters are partitioned into blocks, each block being assigned to one cluster (locally sequential). The blocks are handled separately by different clusters, which can be run in parallel (globally parallel). LPGS links each original one-block cluster to one processor (locally parallel); blocks are executed sequentially (globally sequential). BDSC computes a bounded schedule on the fly and covers many more other possibilities of scheduling than LPGS and LSGP.

Triplet [31] is a clustering algorithm for heterogeneous architectures. It proceeds, first, by clustering tasks while assuming an unbounded number of clusters and, then, a second clustering of these first clusters is performed to merge them on actual processors. Here, the sorting of tasks is based on tlevel estimates only, contrarily to BDSC, which uses better information.

PYRROS [32] is also based on a two-step method for scheduling. The first step assumes an unbounded number of clusters and uses the DSC algorithm. Then, in the second step, an other clustering, on P processors, is performed, using cluster merging. This mapping sorts the clusters in the ascending order of their loads and then maps each cluster to a processor in such a way that all processors are as well balanced as possible. An another mapping is also used in order to minimize the communication costs between the P processors, using a pairwise interchange and task reordering heuristic. Note that, in this method, each step may change the decisions taken in the precedent step. For example, the ordering of tasks performed using DSC is modified during the second step. Also, in this second step, load balancing may be lost when performing the communication reduction heuristic.

The main difference between BDSC and the techniques that remap clusters on physical processors is that BDSC computes, by design, a bounded schedule. This is efficient in term of algorithmic complexity. Moreover, communication minimization is done once, while load balancing is ensured by our efficient task-to-cluster mapping heuristic. BDSC handles completion time and communication cost minimization and load balancing as parts of a single process.

Note that all the works surveyed here and in Section 6.1.1 do not explain how information about costs of communication or execution times of tasks is obtained; they also do not address the construction of the DAG; finally, BDSC is the only scheduling algorithm that takes into account the memory parameter, ignored in all these works. Our paper provides a much broader picture, from analyzing sequential codes up to the generation of scheduled task graphs.

6.2. Parallelization Tools

In this section, we review several approaches that intend to automate the parallelization of programs using different granularities and scheduling policies. Given the breadth of literature on this subject, we limit this presentation to approaches that focus on static list-scheduling methods.

Sarkar's work on the partitioning and scheduling of parallel programs [33] for multiprocessors introduced a compile-time method where a GR (Graphical Representation) program is partitioned into parallel tasks at compile time. A GR graph has four kinds of vertices: "simple", to represent an indivisible sequential computation, "function call", "parallel", to represent parallel loops, and "compound", for conditional instructions. Sarkar presents an approximation parallelization algorithm. Starting with an initial fine granularity partition, P_0, tasks (chosen by heuristics) are iteratively merged till the coarsest partition P_n (with one task containing all vertices), after n iterations, is reached. The partition P_{min} with the smallest parallel execution time in the presence of overhead (scheduling and communication overhead) is chosen. For scheduling, Sarkar introduces the EZ (Edge-Zeroing) algorithm that uses blevels for ordering: it is based on edge weights for clustering; all edges are examined from the largest edge weight to the smallest; it then proceeds by zeroing the highest edge weight if the completion time decreases. While this algorithm is based only on the blevel for an unbounded number of processors and does not recompute the priorities after zeroings, BDSC adds resource constraints and is based on both blevels and dynamic tlevels.

The OSCAR Fortran Compiler [34] is used as a preprocessor from Fortran to parallelized OpenMP Fortran. OSCAR partitions a program into a macro-task graph, where vertices represent macro-tasks of three kinds, namely basic, repetition and subroutine blocks. The coarse grain task parallelization proceeds as follows. First, the macro-tasks are generated by decomposition of the source program. Then, a macro-flow graph is generated to represent data and control dependences on macro-tasks. The macro-task graph is subsequently generated via the analysis of parallelism among macro-tasks using an earliest executable condition analysis that represents the conditions on which a given macro-task may begin its execution at the earliest time, assuming precedence constraints. If a macro-task graph has only data dependence edges, macro-tasks are assigned to processors by static scheduling. If a macro-task graph has both data and control dependence edges, macro-tasks are assigned to processors at run time by a dynamic scheduling routine. In addition to dealing with a richer set of resource constraints, BDSC targets both shared and distributed memory systems with cost models based on communication, used data and time estimations.

Pedigree [35] is a compilation tool based on the program dependence graph (PDG). The PDG is extended by adding a new type of vertex, a Par vertex, which groups children vertices reachable via the same branch conditions. Pedigree proceeds by estimating a latency for each vertex and data dependences edge weights in the PDG. The scheduling process orders the children and assigns them to a subset of the processors. For scheduling, vertices with minimum early and late times are given highest priority; the highest priority ready vertex is selected for scheduling based on the synchronization overhead and latency. While Pedigree operates on assembly code, PIPS and our extension for task-parallelism using BDSC offer a higher-level, source-to-source parallelization framework. Moreover, Pedigree generated code is specialized for only symmetric multiprocessors, while BDSC targets many architecture types, thanks to its resource constraints and cost models.

The SPIR (Signal Processing Intermediate Representation) compiler [36] takes a sequential dataflow program as input and generates a multithreaded parallel program for a multicore system. First, SPIR builds a stream graph where a vertex corresponds to a kernel function call or to the condition of an “if” statement; an edge denotes a transfer of data between two kernel function calls or a control transfer by an “if” statement (true or false). Then, for task scheduling purposes, given a stream graph and a target platform, the task scheduler assigns each vertex to a processor in the target platform. It allocates stream buffers, and generates DMAs under given memory and timing constraints. The degree of automation of BDSC is larger than SPIR’s, because this latter system needs several keywords extensions plus the C code denoting the streaming scope within applications. Also, the granularity in SPIR is a function, whereas BDSC uses several granularity levels.

7. Conclusion

This paper presents a new parallelization framework for scientific computing based on the resource-constrained, list-scheduling heuristic BDSC, which extends the DSC (Dominant Sequence Clustering) algorithm, and its practical use, via hierarchical scheduling, when parallelizing scientific applications. This extension improves upon DSC by dealing with memory- and number-of-processors-constrained parallel architectures, while still yielding faster task schedules thanks to efficient task-to-cluster mapping. The use of BDSC benefits from a sophisticated execution and communication cost models, based on either static code analysis or a dynamic-based instrumentation assessment tool.

Preliminary experimental results suggest that BDSC provides more efficient schedules than DSC. We illustrate the positive impact of the integration of

BDSC within the PIPS automatic parallelization compiler infrastructure using the signal processing application ABF (Adaptive Beam Forming), the image processing application Harris, the SPEC benchmark equake and the NAS parallel benchmark IS on both shared and distributed memory systems.

Future work might address the analysis of the potential benefits that an hybrid approach that would mix BDSC and dynamic scheduling could offer.

Acknowledgments

We thank Francois Irigoin for many insightful and helpful discussions, Francois Willot, for his help regarding our experiments on a distributed machine, and the reviewers, for their thorough comments and suggestions.

References

[1] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1990.

[2] S. Tse, Online Bounds on Balancing Two Independent Criteria with Replication and Reallocation, Computers, IEEE Transactions on 61 (11) (2012) 1601–1610.

[3] T. P. Baker, Multiprocessor EDF and Deadline Monotonic Schedulability Analysis, in: Proceedings of the 24th IEEE International Real-Time Systems Symposium, RTSS '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 120–129.

[4] M. Cirinei, T. P. Baker, EDZL Scheduling Analysis, in: Proceedings of the 19th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, 2007, pp. 9–18.

[5] T. Yang, A. Gerasoulis, DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors., IEEE Trans. Parallel Distrib. Syst. 5 (9) (1994) 951–967.

[6] A. Gerasoulis, S. Venugopal, T. Yang, Clustering Task Graphs for Message Passing Architectures, SIGARCH Comput. Archit. News 18 (3b) (1990) 447–456.

[7] F. Irigoin, P. Jouvelot, R. Triolet, Semantical Interprocedural Parallelization: An Overview of the PIPS Project, in: Proceedings of the 5th International Conference on Supercomputing, ICS '91, ACM, New York, NY, USA, 1991, pp. 244–251.

[8] Y.-K. Kwok, I. Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors, ACM Comput. Surv. 31 (1999) 406–471.

[9] P. Jouvelot, R. Triolet, Newgen: A Language Independent Program Generator, Tech. rep., CRI/A-191, MINES ParisTech (July 1989).

[10] J. Ferrante, K. J. Ottenstein, J. D. Warren, The Program Dependence Graph And Its Use In Optimization, ACM Trans. Program. Lang. Syst. 9 (3) (1987) 319–349.

[11] Graphviz, Graph Visualization Software, `http://www.graphviz.org`.

[12] B. Creusillet, F. Irigoin, Interprocedural Array Region Analyses, International Journal of Parallel Programming 24 (6) (1996) 513–546.

[13] E. Ehrhart, Polynômes arithmétiques et méthode de polyèdres en combinatoire, International Series of Numerical Mathematics (1977) 35.

[14] PolyLib, A Library of Polyhedral Functions, `http://icps.u-strasbg.fr/polylib/`.

[15] D. Khaldi, Automatic Resource-Constrained Static Task Parallelization: A Generic Approach, Ph.D. thesis, MINES ParisTech, A/538/CRI (November 2013).

[16] L. Griffiths, A Simple Adaptive Algorithm for Real-Time Processing in Antenna Arrays, Proceedings of the IEEE 57 (1969) 1696 – 1704.

[17] C. Harris, M. Stephens, A Combined Corner and Edge Detector, in: Proceedings of the 4th Alvey Vision Conference, 1988, pp. 147–151.

[18] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, J. Xu, Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers, Computer Methods in Applied Mechanics and Engineering 152 (1–2) (1998) 85–102.

[19] NPB, NAS Parallel Benchmarks, `http://www.nas.nasa.gov/publications/npb.html`.

[20] OpenMP, Specifications, `http://openmp.org/wp/openmp-specifications/`.

[21] MPI, Message Passing Interface, `http://www-unix.mcs.anl.gov/mpi`.

[22] T. L. Adam, K. M. Chandy, J. R. Dickson, A Comparison of List Schedules For Parallel Processing Systems, Commun. ACM 17 (12) (1974) 685–690.

[23] B. Kruatrachue, T. G. Lewis, Duplication Scheduling Heuristics (DSH): A New Precedence Task Scheduler for Parallel Processor Systems, Oregon State University, Corvallis, OR (1987).

[24] M.-Y. Wu, D. D. Gajski, Hypertool: A Programming Aid For Message-Passing Systems, IEEE Trans. on Parallel and Distributed Systems 1 (1990) 330–343.

[25] H. Topcuouglu, S. Hariri, M.-y. Wu, Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.

[26] M. A. Khan, Scheduling for Heterogeneous Systems using Constrained Critical Paths, Parallel Computing 38 (4–5) (2012) 175 – 193.

[27] M. Solar, M. Inostroza, A Scheduling Algorithm to Optimize Real-World Applications, in: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops, Volume 7, ICDCSW '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 858–862.

[28] Y.-K. Kwok, I. Ahmad, Benchmarking the Task Graph Scheduling Algorithms, in: Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998, 1998, pp. 531–537.

[29] D. I. Moldovan, J. A. B. Fortes, Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays, IEEE Trans. Comput. 35 (1) (1986) 1–12.

[30] K. Jainandunsing, Optimal Partitioning Scheme for Wavefront/Systolic Array Processors, in: Proceedings of IEEE Symposium on Circuits and Systems, 1986, pp. 940–943.

[31] B. Cirou, E. Jeannot, Triplet: A Clustering Scheduling Algorithm for Heterogeneous Systems, in: Parallel Processing Workshops, 2001. International Conference on, 2001, pp. 231–236.

[32] T. Yang, A. Gerasoulis, PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors, in: Proceedings of the 6th International Conference on Supercomputing, ICS '92, ACM, New York, NY, USA, 1992, pp. 428–437.

[33] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors, MIT Press, Cambridge, MA, USA, 1989.

[34] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, S. Narita, A Multi-Grain Parallelizing Compilation Scheme for OSCAR (Optimally Scheduled Advanced Multiprocessor), in: Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag, London, UK, 1992, pp. 283–297.

[35] C. Newburn, J. Shen, Automatic Partitioning of Signal Processing Programs for Symmetric Multiprocessors, in: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, 1996, pp. 269–280.

[36] Y. Choi, Y. Lin, N. Chong, S. Mahlke, T. Mudge, Stream Compilation for Real-Time Embedded Multicore Systems, in: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '09, Washington, DC, USA, 2009, pp. 210–220.