Runtime Support for Dynamic Skeletons Implementation

Javier Fresno, Arturo Gonzalez-Escribano, and Diego R. Llanos
Departamento de Informática
Edif. Tecn. de la Información, Universidad de Valladolid,
Campus Miguel Delibes, 47011 Valladolid, Spain
E-mail: {jfresno, arturo, diego}@infor.uva.es

***Abstract*—Algorithmic skeletons have proved to be a good solution to the problem of implementing parallel applications with specify communication structures. They define the overall structure of the computation, hiding the complex communication details. Nowadays, the different frameworks available offer a fixed set of skeletons. The programmer can implement efficient programs if the computation and communication patterns match the available skeletons. Because of that, the usage of skeleton frameworks has been limited to an important but relative small set of patterns featuring the most common parallel structures, such as map, pipeline, farm, or wavefront.**

In this paper, we present a programming model that can be used to implement efficient and portable parallel skeletons. We also discuss its implementation and integration into Hitmap, a tool for hierarchical tiling and mapping. This combined proposal allows to develop tailored static and dynamic skeletons while still hiding implementation and communication details. The performance of the implementation is measured against a well-known skeleton framework.

***Keywords*—Algorithm skeletons, Parallel programming models, Dynamic computation**

I. Introduction

Development of parallel software is a quite complicated task. Typically, programming for parallel machines is based on message passing libraries such as MPI [1] or shared memory APIs like OpenMP [2]. These solutions allow to write portable code for different machine architectures. However, the programmer has to deal with several non-trivial issues such as problem decomposition, data distribution across processes, local computation, data exchanges, load balancing, or synchronization. Consequently, implementing and debugging a parallel application can be a tedious and error-prone task.

Algorithm skeletons [3], [4] offer the programmer a different view, based on the fact that many parallel algorithms share common computation patterns. Skeletons are a high-level parallel programming model that aims to encapsulate the overall structure of computation, hiding the complex details of parallel applications. With skeletons, programmers do not have to write the code to perform the coordination or communication. They only have to provide the specific code to solve the problem, using the skeleton as a template.

The skeletons could be classified in two groups depending on the nature of their computation structure. A skeleton with a static computation structure (i.e. based on a stencil) can be implemented with coarse-grain partition techniques, using a static scheduling that can be pre-calculated at compilation or initialization time. However, in a dynamic computation structure (i.e. a farm), where data dependent tasks flow through diverse computation stages, dynamic load-balancing solutions are needed to develop efficient programs.

Due to the advantages provided by the use of algorithmic skeletons, a significant number of frameworks and libraries have been developed so far. However, each one of them offers a limited set of skeletons focused on particular techniques or architectures. Thus, the programmer has to choose a solution that may not be ideal for the problem and/or not portable, betraying the original idea of the skeletons.

We propose to simplify the implementation of efficient and portable skeletons with a simple and generic programming model. This model is based on Petri nets [5], [6], a well-known and established formalism for modeling and analyzing systems. Our model represents the task flow of a skeleton with two simple element types (processes and containers). These elements can be combined to model the structure of any skeleton. The model supports both static and dynamic structures. However, we focus on dynamic skeletons since the static ones can be more easily implemented with static scheduling techniques.

This paper also shows how to efficiently implement the proposed model. We have developed an implementation integrated into Hitmap, a tool for hierarchical tiling and mapping of dense arrays and sparse structures. Hitmap already offers solutions to automatize programs with static computation structures. It incorporates data partition techniques that automatically adapt the program to the current data size and current available computational units. Our extension adds support for dynamic skeletons in Hitmap.

Experimental work has been conducted to prove that the implementation achieves good performance with a case of study. Any framework using the proposed abstraction layer can take advantage of this generic model to design skeletons while obtaining efficient implementations.

The rest of the paper is organized as follows. Section II describes some related work in the field. A list of common skeletons and related concepts is given in Sect. III. Section IV discusses the design of our solution model. Section V provides an overview of the Hitmap library, while Sect. VI shows the implementation of our solution in Hitmap. Section VII presents experimental work conducted to test this implementation. Finally, Section VIII concludes our paper.

II. Related work

This section describes some related skeleton frameworks. Each one of them has a different approach, offering a set of skeletons, or focusing on a particular architecture. A more exhaustive survey can be found in [7].

Some skeleton frameworks are designed with a distributed memory model in mind. For example, the Edinburgh Skeleton Library (eSkel) [8] is a C library that uses the standard message passing interface (MPI). It defines several data and task skeletons that are presented as collective operations involving groups of processes.

Another distributed skeleton framework is the Münster Skeleton Library Muesli [9]. It also uses MPI for communications. Muesli follows a two-tier model, where data parallel skeletons can be nested inside task parallel ones. This library is implemented with C++ and takes advantage of object-oriented features, such as polymorphic types.

A successful solution for shared-memory multi-core architectures is Threading Building Blocks (TBB) [10], a library developed by Intel. TBB provides a portable implementation of parallel patterns, thread-safe containers, and synchronization primitives. The core of the library is a thread pool managed by a task scheduler. This scheduler efficiently maps tasks onto threads, balancing the computational load using a work-stealing algorithm.

A well-known model for processing large data sets is Map-Reduce [11]. It is a two-stage model and it is used to process pairs of key/value elements. There are several implementations of this model, for example the open-source project Apache Hadoop.

Finally, SkelCL [12] is a GPU skeleton library based on data-parallel algorithmic skeletons. It generates OpenCL code, that is compiled by OpenCL at runtime.

III. A Skeleton Taxonomy

Skeletons are generally classified as *data parallel* and *task parallel*. Previous surveys add an extra category with part of the task skeletons class, named *resolution skeletons* [7], [9]. Data parallel skeletons work with data structures and manipulate their elements according with computation patterns in a fine grain. Task parallel skeletons compute workflows of tasks. Resolution skeletons solve a family of problems with iterative phases of computation, communication, and control.

We propose to classify the skeletons in *static* or *dynamic*. depending on the nature of their computation structure. Static skeletons maintain the same structure during all their execution, whereas dynamic skeletons have a mutable computation structure. Static skeletons can take advantage of static scheduling methods pre-calculated during the initialization phase. While dynamic skeletons need dynamic scheduling and load balancing techniques.

Table I shows the relation between both classifications. Data parallel skeletons are static. On the other hand, resolutions skeletons are dynamic because their computation structure depend on the particular data being processed. Finally, we find in the task-parallel skeletons class both static and dynamic examples.

	Static	**Dynamic**
Data parallel	map, fork, zip, reduce, scan, stencil	-
Task parallel	pipeline, wavefront	farm
Resolution	-	divide and conquer (D&C), branch and bound (B&B), mapreduce

TABLE I. Algorithm taxonomy.

A. Summary of skeleton solutions

There are several design concepts that have to be taken into account when developing a new skeleton framework. This section collects the details introduced previously in the literature.

a) Nesting mode: If a skeleton uses internally another one, there are two possible nesting modes: transient or persistent [8], [13]. In a transient nesting, the outer skeleton calls an inner one to process some internal data. The inner skeleton only exists during the invocation of the external stage. A new instance is created each time. In a persistent nesting, the input and output of the outer skeleton is mapped to the inner one. The instance is persistent between invocations.

b) Interaction mode: This concept defines the relationship between the skeleton input and output. There are two possible interaction modes: implicit and explicit [8]. In an implicit interaction mode, a skeleton produces an output for each consumed input. In an explicit interaction mode, a stage in the skeleton can produce an output arbitrarily without a previous input. Moreover, a skeleton can process an input without producing a result.

c) Task scheduler: Several skeletons such as Farm or Divide&Conquer are composed of a set of workers. This kind of skeleton needs a mechanism to send the tasks to workers and to collect the results. The use of a dispatcher and a collector is one of the possible solutions. However, it has been proved that this solution does not achieve a good performance [14]. Instead, distributed solutions, such as the TBB scheduler [10] or a distributed work pool [15], are preferred because they avoid the contention and bottleneck that may arise with the use of a centralized scheme.

d) Task distribution: A work pool requires a distribution scheme to assign task to workers. Since the time required to process a particular task is usually not known, many work pools assume that each one requires the same time. Under these conditions, there are two independent distribution schemes: Random and cyclic. Both schemes lead to similar performance when there are a big number of tasks. However, a cyclic distribution performs a fairer distribution when the number of tasks is small [14]. More complex schemes with load balancing can be applied if there is information about the actual load of each task and worker.

IV. Uniform model for skeleton implementation

In this section we present our proposed model to represent algorithm skeletons. The model is based on Petri nets [6], [5], a mathematical modeling language for the description of systems. A Petri net is a particular kind of directed bipartite graph, whose nodes represent transitions. We will add new

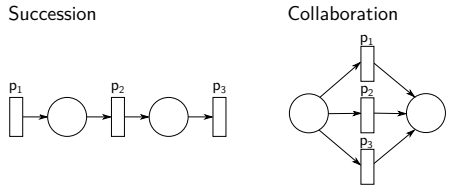

Fig. 1. Representation of the composition operators.

concepts to the original Petri nets definition to describe the tasks involved in the computational patterns of skeletons.

The top level element of the model is an *Application*. It corresponds to a Petri net, and it is defined as a 3-tuple, $A = (C, P, F)$ where:

- $C = \{c_1, c_2, \ldots, c_n\}$ is a finite set of *Task Containers*. This is one of the partitions of the bipartite graph. The task containers correspond to the Petri net *places*, although task containers are typed and store tasks instead of tokens. The task type has to agree with the container type.
- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *Processes*. They are the equivalent to the Petri net *transitions*. This set of process is the other partition of the bipartite graph. A process executes a function with state, they are defined by the user to implement the particular skeleton application. The function has r inputs and s outputs: $f_i : x_1, x_2, \ldots, x_r \rightarrow y_1, y_2, \ldots, y_s$
- The last element of the application $F \subseteq (P \times T) \cup (T \times P)$, is a set of *Flow Relationships* (Petri arcs) between task containers and processes, and vice versa, defining the edges of the bipartite graph.

Based upon the arcs, we can define the input containers. A container is called an *Input Container* for a process if there is an arc from it to the process. *Output Containers* can be defined analogously.

An application net can be nested inside a process node. Transient and persistent nesting modes can be represented with this model. In transient nesting, a process will execute another Application as part of the function, while in persistent mode a process can be replaced by another net, keeping its inputs and outputs.

A. Composition operators

We define two operators that help to define the structure of the application: *succession* and *collaboration*. The succession operator links several processes one after the other using containers and creating the flow relationships between them. The collaboration operator creates a different structure where the processes share the input and output containers. Fig. 1 shows the representation of the composition operators.

B. Execution semantics

Once created, the structure of an application is fixed, although its state (the distribution of tasks in the containers) can change. The behavior of the application is described in term of those states. The tasks in the containers are consumed/-generated by the processes based on the following rules:

- At the initial state, the containers are empty.
- When a process is executed, it consumes tasks from each of its input containers. There have to be at least one task at each of them.
- The retrieved tasks are fed to the process function. The result tasks are sent to the output containers. The process function may do not produce tasks for all the output containers.
- The evolution of the application is not deterministic. When more than one process could be executed, we can not tell which one will be executed first.
- The execution finishes when all tasks have been processed.

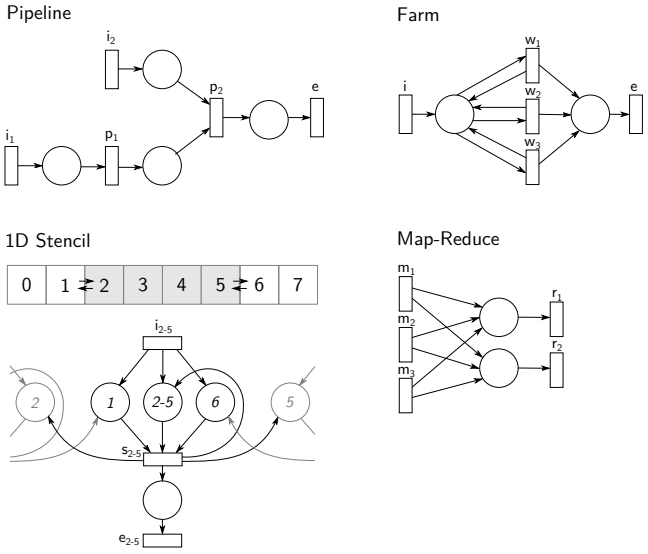

Fig. 2. Common skeletons using the model. A circle represents a container and a rectangle indicates a process.

C. Representing skeletons with this model

We discuss in this section how the algorithm skeletons can be represented using processes and containers. We have selected one representative example from each group defined in the taxonomy of secction III. Fig. 2 shows the structure for each example. The figure uses the standard Petri net representation, where a circle indicates a container, a process is shown as a rectangle, and the flow relationships are arrows from/to elements.

e) Pipeline: A Pipe skeleton is composed of a set of connected stages. The output of one stage is the input of the following one. The structure of this skeletons is just a set of processes (one for each stage) that exchanges tasks using containers. As shown in Fig. 2, a process can receive tasks from several stages using different containers, leading to a more complex pipeline structure. In the same way, a process can feed tasks to more than one output container.

f) Farm: A Farm skeleton, also known as master-slave/worker, consists of a farmer and several workers. The farmer receives a sequence of independent tasks and schedules them across the workers. In a farm skeleton structure, the farmer and the workers are independent processes. There exist two tasks containers. The first one is shared by all the workers and it keeps the tasks that are scheduled to them. The other one is used to store the output results. In some farm configurations, the workers can add more tasks to the input container.

g) Stencil: Although the model is more useful to represent dynamic structure skeletons, it can also represent static ones. A Stencil skeleton updates the value of each element of a data structure applying an operation with the values of their neighbor elements. Fig. 2 shows an example of an 1D stencil. The structure to represent this skeleton has a container for its local elements and containers for the values of the neighbors. Each process updates its local part and inserts the values needed by its neighbors in the appropriate containers.

h) Map-Reduce: This is a distributed programming model used by Google for efficient large-scale computations [11]. The model proposes two steps: map and reduce. The computation in the map step takes a set of input key/value pairs and processes them in parallel. The result for each pair is another set of intermediate output key/value pairs. The reduce step merges together all the intermediate pair associated with the same key, returning a smaller set of output key/value pairs. A Map-Reduce structure has a pair of process sets, one with the processes performing the map operation and the other performing the reduction. They are connected by several task containers that hold the intermediate key/value pairs.

V. The Hitmap library

Before describing the implementation of this model in Hitmap, we will briefly show the main features of the Hitmap library.

Hitmap [16] is a library for hierarchical tiling and mapping, with support for dense and sparse data structures [17]. It is based on a distributed SPMD programming model, using abstractions to declare data structures with a global view, automatizing the partition, mapping, and communication of hierarchies of tiles, while still delivering good performance.

Hitmap was designed with an object-oriented approach, although it is implemented in C language. The classes are implemented as C structures with associated functions.

Hitmap abstractions allow to represent different data domains with a single interface. This interface has currently implementations for dense arrays, subspaces of array indexes with regular jumps, and sparse domains, such as Compressed Sparse Row (CSR) or Bitmaps. Hitmap also has functionalities to modify the domains, make selections, allocate memory for an index subspace, or make efficient data copies.

Hitmap has a runtime plug-in system to distribute the data domains. Plug-ins with different partition methods can be selected. They divide the domains according to the actual processors arranged in a virtual topology. Hitmap has different partitioning and load-balancing techniques implemented. Moreover, programmers may include their own new techniques. It also allows to define communication patterns in terms of the mapping results and neighbor relationships, that automatically adapt the data distribution and communication scheme at execution time.

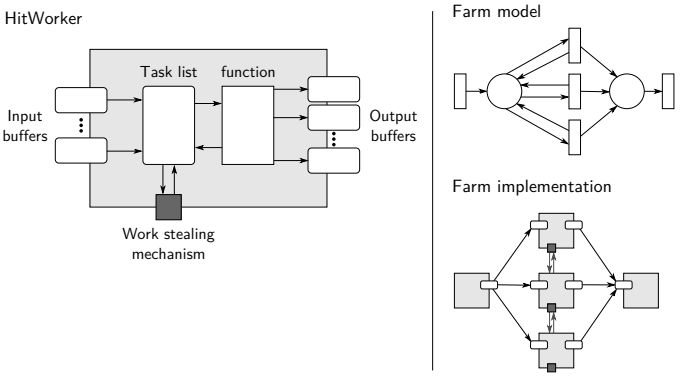

Fig. 3. Implementation of the HitWorker class, and an example of the farm skeleton.

The library is built on top of the MPI communication library, for portability across different architectures. Hitmap internally exploits several MPI techniques that increase performance, such as MPI derived data-types and asynchronous communications. The Hitmap library is publicly available [18].

VI. Skeleton Model Implementation using Hitmap

This section explains how we have extended the Hitmap library to implement the proposed model.

This extension constitutes another step in the development of Hitmap. The library has functionalities to deal with static computation structures, being able to partition different kinds of data structures at initialization time. This proposal adds support for load balancing, and dynamic distribution of tasks.

i) Implementation of the model elements: The current implementation of the model adds two new classes to the architecture: a *HitTask*, and a *HitWorker*. A HitTask class, an abstract datatype, is used to encapsulate the data that flows between application stages. This HitTask class has a weight attribute that is used in load-balancing decisions. A HitWorker is a generic worker that runs over a process, executing a user function.

The containers of the model are implemented as lists of tasks inside the workers. When a worker generates a task for another process, the task is sent using MPI communications, and it is inserted in the worker task list. If several processes share an input container, for example in a farm structure, it is implemented as a distributed list. Each worker has a local list and the tasks are communicated using a work stealing mechanism to balance the load.

The arcs of the bipartite graph are task channels between workers, creating successor-predecessor relations. This allows them to be arbitrarily nested. Fig. 3 shows the internal details of an example HitWorker object with two input and three output channels. It also shows how several workers can be linked to implement a farm skeleton structure defined using the model.

j) Worker operations: There are several operations that can be processed at the same time in a worker: Tasks reception, task sending, function execution, and work stealing. To minimize the impact between operations, the worker has been implemented using several threads that handle these operations independently. A worker works in the following way:

- There is a thread that waits to receive tasks from any of its predecessors, inserting them in the local list.
- Another thread executes the user function. This function processes the available tasks in the list. The execution of the function can: (1) generate new tasks for the local list, (2) generate new tasks for the successors, or (3) produce no output.
- If the list is a distributed one, a work-staling mechanism runs in the background. When the local list is going to be emptied, it tries to get tasks from the other workers that share the virtual container.

A. Optimization details

This section describes the most relevant optimization details of the implementation described above. Using MPI to communicate a single task is not efficient in a generic case, due to the time and memory overhead of the communication operations. To avoid this, we have implemented input and output tasks buffers that group tasks before communication. The granularity of the buffers can be modified in terms of task weights. These buffers use asynchronous blocking MPI calls managed by different threads. In this way, communication operations can overlap. In addition, MPI Communicators are used to isolate message contexts for the work-stealing mechanism, also allowing skeleton nesting.

Special care has been taken to design the worker's internal list of tasks. The different threads of the worker can modify this list, so mutual exclusion should be ensured to avoid inconsistent states. Moreover, the task in the list can be originated from different sources: predecessors workers, work stealing exchanges, and locally generated tasks. The user function extracts and inserts tasks using one end of the list while the input buffers and worker stealing mechanism use the other one. This improves task locality for the applications that can exploit it.

Our tool allows to change different parameters in order to test which configurations are better for a given program. We can change the list and buffers sizes, change the task grouping policy, or disable the work-stealing mechanism.

To allow explicit, as well as implicit interaction modes, the required user function is executed once. This allows to declare and free internal data structures to keep state, wrapping the task management loop. The implementation offers to the programmer of the function an API of methods to retrieve tasks from the list, access the data of the tasks, create new tasks, or send tasks to the output buffers. The input and output are not limited to the one to one relation of the implicit interaction mode. The programmer is free to combine the functions of this API to created any skeleton stage.

Support for both transient and persistent nesting modes has been considered. Persistent mode is achieved just by linking workers. The transient mode poses a bigger challenge, it implies that sub-skeletons could be created during the execution of the upper one. This is solved with a pile of execution contexts.

B. Implementation examples

This section exemplifies how to create skeletons with our implementation of the model. Fig. 4 shows an example of a pipeline. The function in line 5 creates a three stage pipeline. It receives three function pointers as parameters, one for each stage. These user functions must receive a HitWorker structure as defined in the typedef of their prototype on line 2. The *pipe3* function creates three workers with the corresponding user function and defines the number and types of the inputs and outputs. Then, it combines the workers with the succession operator to create the relationships. The result is another worker that encapsulates the previous ones. We can use it to execute the whole pipe with a single call in the main function (line 19).

Creating a farm is a similar process but uses the collaborator operator instead. More complex computation structures can be created manually defining how the inputs and outputs of the different workers are linked.

VII. Experimental results

Experimental work has been conducted to show that the implementation developed achieves good performance with different configurations, and compared with another skeleton framework. We use two different experimental platforms with different architectures: A multicore shared-memory machine and a distributed cluster of commodity PCs. The shared-memory system, Geopar, is an Intel S7000FC4URE server with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32GB of RAM. The distributed system is a homogeneous Beowulf cluster composed by 20 AMD Athlon 3000+ single-core processors at 1.8GHz and 1Gb of RAM each. The cluster is interconnected by a 100Mbit Ethernet network. The MPI implementation used is MPICH2.

A. Mandelbrot set benchmark

We have chosen a simple benchmark with no complex application interactions, to focus on the efficiency of the implementation. The selected benchmark calculates the Mandelbrot set [19], one of the best-known examples of mathematical visualization. It has become popular as a benchmark in parallel computing since it is easily parallelizable but introduces a load-balancing problem [1]. Several skeleton frameworks use it as a case study [4], [12].

The Mandelbrot set is defined in the following way. Given a complex number $c \in \mathbb{C}$ and the sequence $z_{n+1} = z_n + c$, starting with $z_0 = 0$, c belongs to the Mandelbrot set if, when applying the iteration repeatedly, the sequence remains bounded regardless of how big n gets.

The benchmark computes the iterative equation for each point to calculate whether the sequence tends to infinity. If this sequence does not cross a given threshold before reaching a given number of iterations, it is considered that the sequence will converge. This problem is straightforwardly parallelizable

```
// Typedef for the user function pointer.
typedef void (*HitWorkerFunc) (HitWorker*);

// Three-stage pipeline
HitWorker pipe3(HitWorkerFunc f_ini, HitWorkerFunc f_mid, HitWorkerFunc f_end){
    HitWorker pipe;
    HitWorker * workers = malloc(3 * sizeof(HitWorker));

    hit_workerCreate(&workers[0], f_ini, 0, 1, HIT_DOUBLE);
    hit_workerCreate(&workers[1], f_mid, 1, 1, HIT_DOUBLE, HIT_DOUBLE);
    hit_workerCreate(&workers[2], f_end, 1, 0, HIT_DOUBLE);

    hit_workerOpSuccession(&pipe,3,&workers[0],&workers[1],&workers[2]);

    return pipe;
}

// Main function
int main(int argc, char ** argv) {

    hit_comInit(&argc,&argv);
    HitWorker pipe = pipe3(init,process,end);
    hit_workerExecute(&pipe);
    hit_comFinalize();
}
```

Fig. 4. Fragment of code showing the creation of a pipe and a farm.

because the calculation of the equation on a particular point is independent on the result from any other point. However, it can present significant load imbalances, because some points reach the threshold after only a few iterations, others could take longer, and the points that belong to the set require the maximum number of iterations.

B. Performance

To test the performance obtained by the implementation of the model, we measure and compare the run-time of several implementations of the Mandelbrot benchmark: (1) the Hitmap implementation; (2) the Hitmap implementation with the work stealing mechanism disabled, (3) a code using the Muesli skeleton framework [9] forced to use only MPI processes, and (4) the Muesli code forced to use OpenMP threads instead of MPI processes.

The two plots in Fig. 5 show the results of the previously-described implementations of the benchmark for both architectures considered. The programs have been run with a square matrix of $4\,000 \times 4\,000$ elements, a limit of $2\,000$ iterations, and task grain of 16×16 elements. Hitmap obtains a good scalability in the shared-memory machine. In the cluster, the results are not as good because the selected task grain is not enough, and the communications drag down the scalability. The difference of using work stealing in Hitmap is not noticeable for this application. The Muesli implementations do not scale as good as Hitmap. It is specially noticeable in the Beowulf cluster. As expected, in the shared-memory machine, the experiment with the Muesli code using only OpenMP threads has slightly better performance than using MPI processes. It is remarkable than Hitmap performs better in both cases.

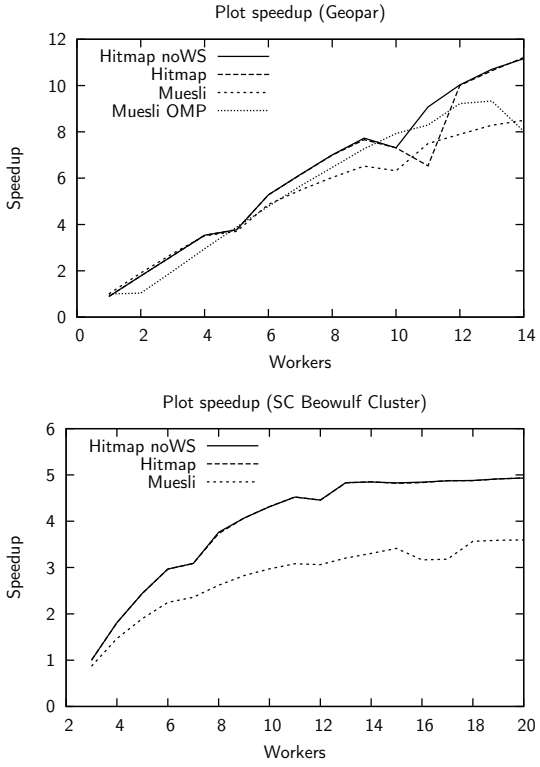

Fig. 5. Speedup comparison for Hitmap, Hitmap without work stealing, and Muesli implementations of the Mandelbrot benchmark.

The plots in Fig. 6 show the performance of the Hitmap

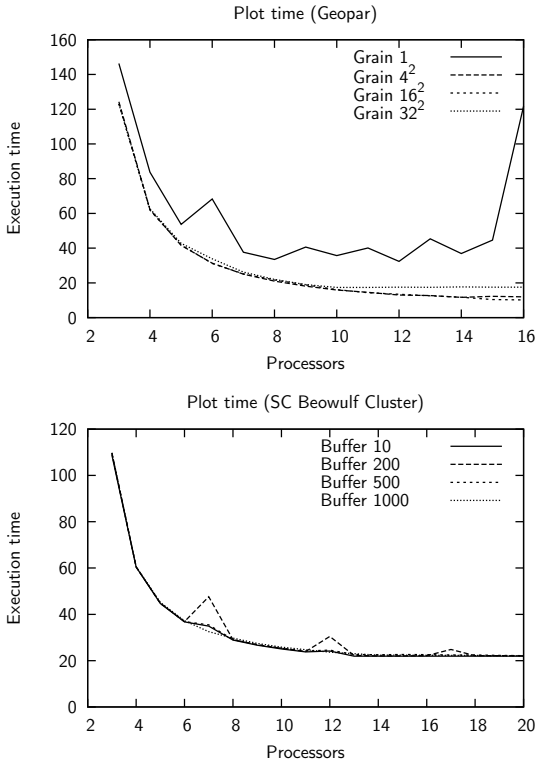

Fig. 6. Execution time comparison for different task grain sizes and communication buffer sizes of the Hitmap implementation.

implementation with different task grain sizes and buffer sizes using the same configuration as the previous experiments. The results for the architectures not presented in the plots are similar on both of them. The plot comparing the task grain shows that extreme values (1 element and 32×32 elements block) do not achieve good performance and scalability. A granularity value which is appropriated for the target system should be chosen. The last plot shows that the buffer size does not have a clear impact on the performance for this application.

VIII. Conclusions

In this paper we present a simple model that can be used to represent algorithm skeletons. We focus our solution on dynamic-structure skeletons, the ones which impose dynamic task-creation, load-balancing, or data-flow issues.

We discuss how to use the proposed model to represent a set of well-known skeletons. We have also developed an implementation of the model, integrating this skeleton support into Hitmap, a library for efficient partition and communication of dense and sparse data structures.

To illustrate the usage of the implementation, we have implemented a simple task skeleton benchmark. We have used it to compare our solution with the Muesli skeleton framework. Our experimental results show that the implementation is highly efficient and configurable.

Our ongoing work includes the creation and encapsulation of more complex skeletons using this model to show its applicability for production parallel applications.

Acknowledgments

The authors would like to thank Prof. Murray Cole for many fruitful discussions. This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2); Ministerio de Industria, Spain (CENIT OCEANLIDER); MICINN (Spain) and the European Union FEDER (Mogecopp project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E); and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative.

References

[1] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI : Portable Parallel Programming With the Message-passing Interface*. MIT Press, 1999.

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*, 1st ed. Morgan Kaufmann, 2001.

[3] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[4] ——, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, Mar. 2004.

[5] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[6] W. M. P. van der Aalst and K. van Hee, *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2002.

[7] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[8] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, "Flexible skeletal programming with eSkel," in *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Lisbon, Portugal, 2005, pp. 761–770.

[9] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The Münster Skeleton Library Muesli – A Comprehensive Overview," Westfälsche Wilhelms-Universität Münster (WWU) - European Research Center for Information Systems (ERCIS), Tech. Rep. 7, 2009.

[10] J. Reinders, *Intel® threading building blocks: Outfitting C++ for Multi-Core Processor Parallelism*, 1st ed. O'Reilly Media, 2007.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*. USENIX Association, 2004.

[12] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL – A Portable Skeleton Library for High-Level GPU Programming," in *Proc. 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011, pp. 1176–1182.

[13] A. Benoit and M. Cole, "Two Fundamental Concepts in Skeletal Parallel Programming," in *The International Conference on Computational Science (ICCS)*, 2005, pp. 764–771.

[14] M. Poldner and H. Kuchen, "On Implementing the Farm Skeleton," *Parallel Processing Letters*, vol. 18, no. 1, pp. 117–131, 2008.

[15] ——, "Algorithmic Skeletons for Branch and Bound," *Software and Data Technologies*, pp. 204–219, 2008.

[16] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos, "An extensible system for multilevel automatic data partition and mapping," *IEEE Transactions on Parallel and Distributed Systems*, 2013, to appear.

[17] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos, "Extending a hierarchical tiling arrays library to support sparse data partitioning," *The Journal of Supercomputing*, vol. 64, no. 1, pp. 59–68, Apr. 2013.

[18] Trasgo Group, "Hitmap Repository," 2013, http://trasgo.dcs.fi.uva.es/hitmap.

[19] B. B. Mandelbrot, "Fractal aspects of the iteration of $z \mapsto \lambda z(1-z)$," *Annals of the New York Academy of Sciences*, vol. 357, pp. 249–259, 1980.