Fine Tuning Algorithmic Skeletons

Denis Caromel and Mario Leyton

INRIA Sophia-Antipolis, CNRS, I3S, UNSA. 2004, Route des Lucioles, BP 93,
F-06902 Sophia-Antipolis Cedex, France
First.Last@sophia.inria.fr

Abstract. Algorithmic skeletons correspond to a high-level programming model that takes advantage of nestable programming patterns to hide the complexity of parallel/distributed applications. Programmers have to: define the nested skeleton structure, and provide the muscle (sequential) portions of code which are specific to a problem.

An inadequate structure definition, or inefficient muscle code can lead to performance degradation of the application. Related research has focused on the problem of performing optimization to the skeleton structure. Nevertheless, to our knowledge, no focus has been done on how to aide the programmer to write performance efficient muscle code.

We present the Calcium skeleton framework as the environment in which to perform fine tuning of algorithmic skeletons. Calcium provides structured parallelism in Java using ProActive. ProAcitve is a grid middleware implementing the active object programming model, and providing a deployment framework.

Finally, using a skeleton solution of the NQueens counting problems in Calcium, we validate the fine tuning approach on a grid environment.

1 Introduction

Algorithmic skeletons correspond to a high level programming model which was introduced by Cole [1]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones. All the non-functional aspects regarding parallelization and distribution are implicitly defined by the composed parallel structure. Once the structure has been defined, the programmer completes the program by providing the application's sequential functional aspects to the skeleton, which we refer to as *muscle* codes.

Skeletons are considered a high level programming paradigm because lower level details are hidden from the programmer. Achieving high performance for an application becomes the responsibility of the skeleton framework by performing optimizations on the skeleton structure [2,3], and adapting to the environment's dynamicity [4]. However, while these techniques are known to improve performance, by themselves they are not sufficient. The functional aspects of the application (ie the muscle), which is provided by the programmer, can be inefficient or generate performance degradations inside the framework.

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 72–81, 2007.

Detecting the performance degradation, providing the programmer with an explanation, and suggesting how to solve the performance bugs are the main motivations of this work. The challenge arises because skeleton programming is a high level programming model. All the complex details of the parallelism and distribution are hidden from the programmer. Therefore, the programmer is unaware of how her muscle code will affect the performance of the application. Inversely, low level information of the framework has no meaning to the programmer to fine tune her muscle code.

In this paper we contribute by: (i) providing performance metrics that apply, not only to master-slave, but also to other common skeleton patterns; (ii) taking into consideration the nesting of task and data parallel skeletons as a producer/consumer problem; (iii) introducing the concept of muscle workout; and (iv) introducing a blaming phase that relates performance inefficiency causes with the actual muscle code. In this paper we also present a skeleton framework called *Calcium*. Calcium is written in Java and provides a library for nesting task and data parallel skeletons using dynamic task generation. Support for distributed programming in Calcium is provided by ProActive's [5] active object model [6], and deployment framework [7].

2 Calcium Skeleton Framework

The Calcium skeleton framework is greatly inspired on Lithium [8,9] and its successor Muskel [10]. It is written in Java [11] and is provided as a library. To achieve distributed computation Calcium uses ProActive. ProActive is a Grid middleware [12] providing, among others, a deployment framework [7], and a programming model based active objects with transparent first class futures [6].

Basic task and data parallel skeletons supported in Calcium can be combined and nested to solve more complex applications, in the following way:

$$\begin{aligned}\triangle ::= & farm(\triangle) \mid pipe(\triangle_1, \triangle_2) \mid seq(f_e) \mid \\ & if(f_b, \triangle_{true}, \triangle_{false}) \mid while(f_b, \triangle) \mid for(i, \triangle) \\ & map(f_d, \triangle, f_c) \mid fork(f_d, \triangle_1, ..., \triangle_n, f_c) \mid d\&c(f_d, f_b, \triangle, f_c)\end{aligned}$$

Where the task parallel skeletons are: $farm$ for task replication; *pipe* for staged computation; *seq* for wrapping execution functions; if for conditional branching; and $while/for$ for iteration. The data parallel skeletons are: *map* for single instruction multiple data; $fork$ for multiple instruction multiple data; and $d\&c$ for divide and conquer.

The Calcium framework can be viewed as a producer/consumer problem. A central task pool stores and keeps track of tasks. Tasks are inputed into the task pool by the users who provide the initial configuration of the state parameter. Interpreters consume tasks from the task pool, compute the tasks according to the skeleton instructions, and return the computed tasks to the task pool. Additionally, new tasks can be dynamically produced by the interpreters when

data parallelism is encountered, in a similar fashion as in [13]. Dynamically produced tasks are referred to as subtasks, while the task that created them is referred to as the parent task.

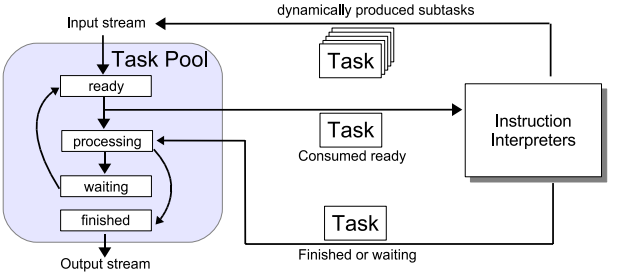
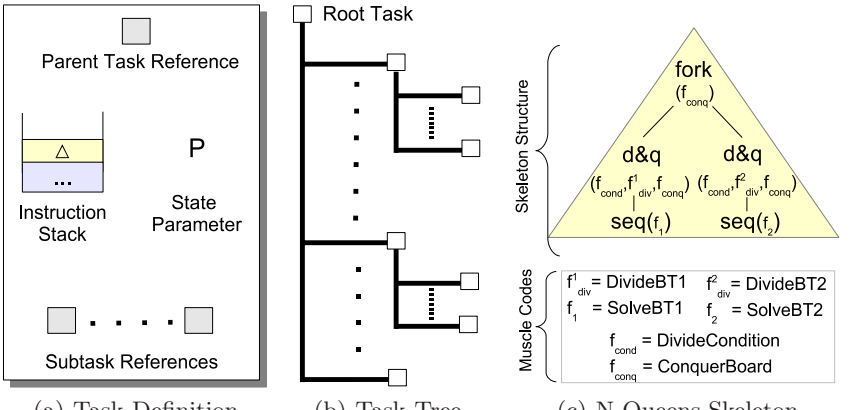


Fig. 1. Calcium Framework Task Flow

As shown in Figure 2(a), a (sub)task is mainly composed of: a skeleton instruction stack, and a state parameter. The instruction stack corresponds to the parsing and execution of the skeleton program, and each instruction is specified with the associated muscle codes. The state parameter is the glue between the skeleton instructions, since the result of one instruction is passed as parameter to the following one.

Since subtasks can also spawn their own subtasks, a task tree is generated as shown in Figure 2(b). The root of the tree corresponds to a task inputed into the framework by the user, while the rest of the nodes represent dynamically generated tasks. In the task tree, the tasks that represent the leaf nodes are



(a) Task Definition (b) Task Tree (c) N-Queens Skeleton

Fig. 2. Skeleton Programming Concepts

the ones that are ready for execution or being executed, while the inner nodes represent tasks that are waiting for their subtasks to finish.

When a task has no unfinished children, and no further skeleton instructions need to be executed, then the task is finished. When all brother tasks are finished, they are returned to the parent task for reduction. The parent may then continue with the execution of its own skeleton, and perhaps generate new subtasks. When a root task reaches the finished state it can be delivered to the user.

The task pool is therefore composed of tasks in 4 different states: ready, processing, waiting and finished. The ready state represents tasks that are currently ready for execution. The processing state keeps track of tasks currently being executed. The waiting state holds the tasks that are waiting for their children to finish. The finished state contains the root tasks that have been finished but have not yet been collected by the user.

3 Muscle Tuning of Algorithmic Skeletons

The global concepts that are presented in this section are depicted in Figure 3. After the execution of an application, the performance metrics are used to determine the causes of the performance inefficiency. Once the causes are identified, a blaming process takes places by considering the workout of the muscle code. The result of the blaming process yields the blamed muscle code. The programmer can then analyze the blamed code to fine tune her application.

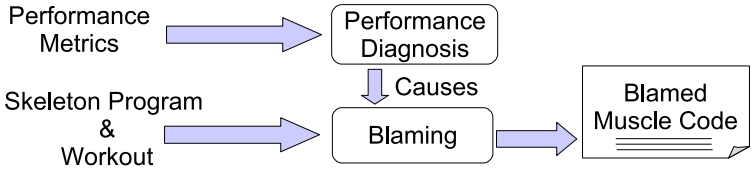


Fig. 3. Finding the tunable muscle code

3.1 Performance Diagnosis

Figure 4 shows a generic inference tree, which can be used to diagnose the performance of data parallelism in algorithmic skeletons. The diagnosis uses the metrics identified in section 3.2 to find the causes of performance bugs. Causes can correspond to two types. *External causes*, such as the framework deployment overhead, or framework overload; and *tunable causes*, which are related with the muscle code of the skeleton program. Since external causes are unrelated with the application's code, in this paper we focus on the tunable causes.

The inference tree has been constructed by considering the performance issues we have experienced with the skeleton framework. As such, the thresholds $\{\alpha, \beta, \ldots\}$ have been determined experimentally, and are provided to the users in three flavors: *weak*, *normal*, and *strong*.

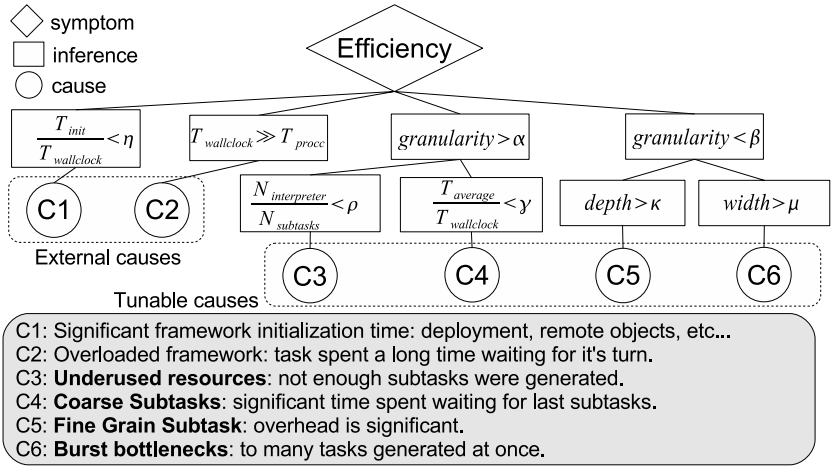


Fig. 4. Generic Cause Inference Tree for Data Parallelism

3.2 Performance Metrics

Task Pool: Number of tasks in each state: $N_{processing}$, N_{ready}, $N_{waiting}$, and $N_{finished}$.

Time: Time spent by a task in each state: $T_{processing}$, T_{ready}, $T_{waiting}$, and $T_{finished}$. For $T_{processing}$ and T_{ready} this represents the accumulated time spent by all the task's subtree family members in the state. For the $T_{waiting}$ and $T_{finished}$, this represents only the time spent by the root task in the waiting state. There is also the $T_{wallclock}$ and $T_{computing}$. The overhead time is defined as: $T_{overhead} = T_{processing} - T_{computing}$, and represents mainly the cost of communication time between the task pool and the interpreters.

Granularity: Provides a reference of the the task granularity achieved during the execution of data parallel skeletons by monitoring the task tree: size, span, and depth ($span^{depth} = size$), and the $granularity = \frac{T_{computing}}{T_{overhead}}$.

3.3 Muscle Workout

Let $m_0, ..., m_k$ be an indexed representation of all the muscle code inside a skeleton program $\triangle$. We will say that workout is a function that, given a skeleton program and a state parameter, after the application is executed, returns a list of all the executed muscle codes with the computation time for each instruction:

$$workout(\triangle, p) = [(m_i, t_0), ..., (m_j, t_n)]$$

The skeleton workout represents a trace of how the muscle codes were executed for this skeleton program. The same muscle code can appear more than once in the workout, having different execution times.

3.4 Code Blaming

Since algorithmic skeletons abstract lower layers of the infrastructure from the programming of the application, low level causes have no meaning to the programmer. Therefore, we must link the causes with something that the programmer can relate to, and this corresponds to the muscle codes which have been implemented by the programmer. The process of relating lower level causes with the responsible code is what we refer to as *code blaming*.

A blaming mechanism must link each inferred cause with the relevant muscle codes. Thus, the blaming must consider: lower level causes (the result of the performance diagnosis), the skeleton program, and the muscle code's workout.

Let us recall that for any skeleton program, it's muscle codes must belong to one of the following types: $f_{execution}, f_{condition}, f_{division}, f_{conquer}$. Additionally, the semantics of the muscle code depend on the skeleton pattern where it is used. A simple implementation of a blaming algorithm is the following:

(C3) Underused resources: Blame the most invoked $f_{cond} \in \{d\&q\}$ and $f_{div} \in \{map, d\&q, fork\}$. Suggest incrementing the times f_{cond} returns *true*, and suggest that f_{div} divides into more parts.

(C4) Coarse subtasks: Blame the least invoked $f_{cond} \in \{d\&q\}$. Suggest incrementing the times f_{cond} returns *true*.

(C5) Fine grain subtasks: Blame the most invoked $f_{cond} \in \{d\&q\}$. Suggest reducing the number of times f_{cond} returns *true*.

(C6) Burst Bottlenecks: Blame the $f_{div} \in \{map, d\&q, fork\}$ which generated the most number of subtasks. Suggest modifying f_{div} to perform less divisions per invocation.

While more sophisticated blaming mechanisms can be introduced, as we shall see in the NQueens test case, even this simple blaming mechanism can provide valuable information to the programmer.

4 NQueens Test Case

The experiments were conducted using the *sophia* and *orsay* sites of Grid5000 [14], a french national grid infrastructure. The machines used AMD Opteron CPU at 2Ghz, and 1 GB RAM. The task pool was located on the sophia site, while the interpreters where located on the orsay site. The communication link between sophia and orsay was of $1[\frac{Mbit}{sec}]$ with $20[ms]$ latency.

As a test case, we implemented a solution of the NQueens counting problem: How many ways can n non attacking queens be placed on a chessboard of nxn? Our implementation is a skeleton approach of the Takaken algorithm [15], which takes advantage of symmetries to count the solutions. The skeleton program is shown in Figure 2(c), where two `d&c` are forked to be executed simultaneously. The first `d&c` uses the *backtrack1* algorithm which counts solutions with 8 symmetries, while the other `d&c` uses the *backtrack2* algorithm that counts solutions with 1, 2, and 4 symmetries.

W	Speedup	Efficiency	Granularity	Size	Width	Depth	T_avg [ms]	T_avg/T_wall
16	11,68	0,12	0,13	24892	11	4,2	0,59	0,05
17	97,27	0,97	5,33	2178	13	3	6,8	3,45
18	67,59	0,68	59,18	166	13	2	90,99	33,89

(a) Performance Metrics

	w = 16		w = 17		w = 18	
Muscle Code	Invoqued [#]	Time[ms]	Invoqued [#]	Time[ms]	Invoqued [#]	Time[ms]
DivideBT1	259	19	18	1	1	1
DivideBT2	1918	154	147	26	10	1
ForkDefaultDivide	1	2	1	2	1	2
SolveBT2	19872	13619957	1771	13730299	137	13910314
ConquerBoard	2178	175	166	15	12	1
SolveBT1	2842	1103186	241	1097325	17	1207444
DivideCondition	24891	523	2177	64	165	10

(b) Workout Summary

Fig. 5. Performance Metrics & Workout Summary for $n = 20$, $w \in \{16, 17, 18\}$

We have chosen the NQueens problem because the fine tuning of the application is centered in one parameter, and therefore represents a comprehensible test case. Task division is achieved by specifying the first $n - w$ queens on the board, where w is the number of queens that have to be backtracked. The problem is thus known to be $O(n^n)$, with exponential increase of the number of tasks as w decreases. Therefore, the programmer of the NQueens problem must tune a proper value for w.

Since the output of the blaming phase corresponds to the blamed muscle code, it is up to the user to identify the parameters that change the behavior of the muscle code. Therefore, the same approach can be used for applications that require tuning of multiple parameters.

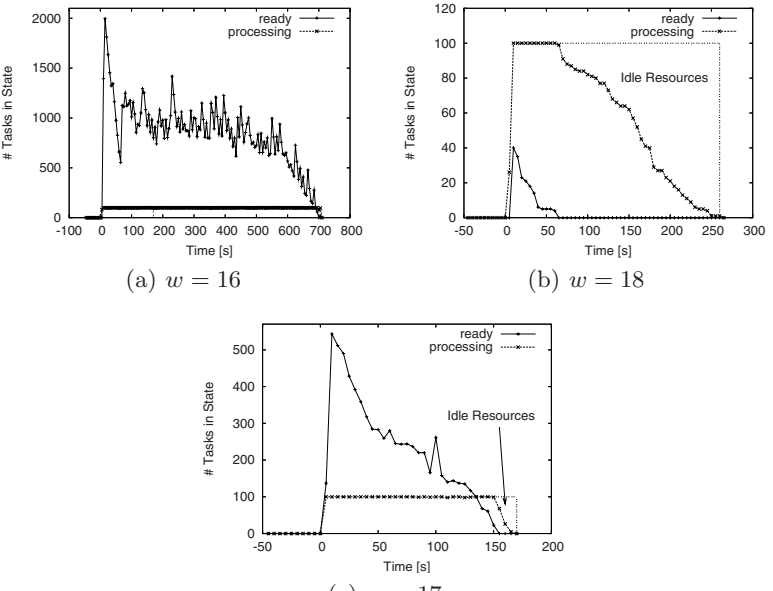
We tested a problem instances for $n = 20$, $w \in \{16, 17, 18\}$, $nodes = 100$. Relevant metrics are shown in Figure 5(a), and a summary of the workout is shown in Figure 5(b). From the performance point of view, several guides can be obtained by simple inspection of the workout summary. For example, that the method `DivideCondition` must remain computationally lite because it is

```
//(n = 20, w = 16)
  Performance inefficiency found.
    Cause: Subtask are too fine grain, overhead is significant.
    Blamed Code: public boolean nqueens.DivideCondition.evalCondition(nqueens.Board)
    Suggested Action: This method should return true less often.

//(n = 20, w = 17)
  No inefficiency found.

//(n = 20, w = 18)
  Performance inefficiency found.
    Cause: Subtask are too coarse, significant time spent waiting for last subtasks.
    Blamed Code: public boolean nqueens.DivideCondition.evalCondition(nqueens.Board)
    Suggested Action: This method should return true more often.
```

Fig. 6. Fine Tuning Output for $n = 20, w \in \{16, 17, 18\}$



(a) $w = 16$ (b) $w = 18$

(c) $w = 17$

Fig. 7. Number of ready and processing subtasks for $n = 20$, with 100 nodes

called the most number of times, and that further optimizations of `SolveBT2` will provide the most performance gain.

The result of the blaming process is shown in Figure 6, where the $f_{condition}$ muscle code (`DivideCondition`) appears as the main tuning point of the application. For $w = 16$ the results suggest that the granularity of the subtasks is to fine, while for $w = 18$ the granularity is to coarse. To better understand why this takes place Figures 7(a), 7(c), 7(b) show the number of subtasks in ready and processing state during the computation. The figures concur with the fine tuning output. As we can see, the fine tuning of a skeleton program can have a big impact on the performance of the application. In the case of the NQueens test case, the performance improves up to 4 times when choosing $w = 17$ instead of $w \in \{16, 18\}$.

5 Related Work

Calcium is mainly inspired by Lithium [8,9] and Muskel [10] frameworks, where skeleton are provided to the programmer through a Java API. Research related to Lithium and Muskel has mainly focused on providing optimizations based on: skeleton rewriting techniques [8,2], task lookahead, and server-to-server lazy binding [3].

In Calcium we explore deeper into the internals of the skeleton task pool. Understanding it's design is vital to comprehend and expose adequate performance metrics concerning the operation of the skeleton framework.

Dynamic performance tuning tools have been designed to aid developers in the process of detecting and explaining performance bugs. An example of such tools is POETRIES [16], which proposes taking advantage of the knowledge about the structure of the application to develop a performance model. Hercules [17] is another tool that has also suggested the use of pattern based performance knowledge to locate and explain performance bugs. Both POETRIES and Hercules have focused on the master-slave pattern, and promoted the idea of extending their models to other common patterns. Nevertheless, to our knowledge, none have yet considered performance tuning of *nestable patterns* (i.e. skeletons), nor have provided a mechanism to relate the performance bugs with the responsible muscle codes of the program.

For skeletons, in [18] performance modeling is done using process algebra for improving scheduling decisions. Contrary to the previously mentioned approaches, this approach is static and mainly aimed at improving scheduling decisions, not at providing performance tuning.

6 Conclusions and Future Work

We have shown a mechanism to perform fine tuning of algorithmic skeletons' muscle code. The approach extends previous performance diagnosis techniques that take advantage on pattern knowledge by: taking into consideration netsable skeleton patterns, and relating the performance inefficiency causes with the skeleton's responsible muscle code. This is necessary because skeleton programming is a higher-level programming model, and as such, low level causes of performance inefficiencies have no meaning to the programmer.

The proposed approach can be applied to fine tune applications that are composed of nestable skeleton patterns. To program such applications we have presented the Calcium skeleton framework. The relation of inefficiency causes with the responsible muscle code is found by taking advantage of the skeleton structure, which implicitly informs the role of each muscle code.

We have validated the approach with a test case of the NQueens counting problem. The experiments where conducted on Grid5000 with up to a 100 nodes.

In the future we would like to improve the performance diagnosis inference tree, and the code blaming mechanism to consider, among others, stateful skeletons. Additionally, in the case where many causes are found, we would like to provide a prioritization scheme to be able to help the programmer decide which are the most significant and relevant muscle codes that must be fine tuned.

References

1. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge, MA, USA (1991)
2. Aldinucci, M., Danelutto, M.: Stream parallel skeleton optimization. In: Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, USA, November 1999, pp. 955–962. IASTED, ACTA press (1999)

3. Aldinucci, M., Danelutto, M., Dünnweber, J.: Optimization techniques for implementing parallel skeletons in grid environments. In: Gorlatch, S. (ed.) Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming, Stirling, Scotland, UK, July 2004, pp. 35–47. Universität Münster, Germany (2004)
4. Danelutto, M.: Qos in parallel programming through application managers. In: PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05), Washington, DC, USA, pp. 282–289. IEEE Computer Society Press, Los Alamitos (2005)
5. ProActive: `http://proactive.objectweb.org`
6. Caromel, D.: Toward a method of object-oriented concurrent programming. Communications of the ACM 36(9), 90–102 (1993)
7. Baude, F., Caromel, D., Mestre, L., Huet, F., Vayssière, J.: Interactive and descriptor-based deployment of object-oriented grid applications. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, July 2002, pp. 93–102. IEEE Computer Society Press, Los Alamitos (2002)
8. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Generation Computer Systems 19(5), 611–626 (2003)
9. Danelutto, M., Teti, P.: Lithium: A structured parallel programming enviroment in Java. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) ICCS 2002. LNCS, vol. 2330, pp. 844–853. Springer, Heidelberg (2002)
10. Danelutto, M., Dazzi, P.: Joint structured/unstructured parallelism exploitation in Muskel (to appear). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2006. LNCS, vol. 3991, Springer, Heidelberg (2006)
11. Microsystems, S.: Java, `http://java.sun.com`
12. Caromel, D., Delbe, C., Costanzo, A., Leyton, M.: Proactive: an integrated platform for programming and running applications on grids and p2p systems. Computational Methods in Science and Technology 12 (2006)
13. Priebe, S.: Dynamic task generation and transformation within a nestable workpool skeleton. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 615–624. Springer, Heidelberg (2006)
14. Grid5000: Official web site, `http://www.grid5000.fr`
15. Takaken: N queens problem, `http://www.ic-net.or.jp/home/takaken/e/queen/`
16. Cesar, E., Mesa, J.G., Sorribes, J., Luque, E.: Modeling master-worker applications in poetries. hips 00, 22–30 (2004)
17. Li, L., Malony, A.: Model-based performance diagnosis of master-worker parallel computations. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 35–46. Springer, Heidelberg (2006)
18. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Evaluating the performance of skeleton-based high level parallel programs. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2004. LNCS, vol. 3036, pp. 299–306. Springer, Heidelberg (2004)