Project no.FP6-034442

GridCOMP

Grid programming with COMPonents : an advanced component platform for an effective invisible grid

STREP Project

Advanced Grid Technologies, Systems and Services

D.NFCF.03 – Methodology to derive performance models for component and composite components

Due date of deliverable: 31 May 2008

Actual submission date: June 24, 2008

Start date of project: 1 June 2006 **Duration:** 30 months

Organisation name of lead contractor for this deliverable: UNIPI

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PU	Public	PU

Keyword list: Skeletons, performance models, autonomic management
Responsible Partner: Marco Danelutto, UNIPI

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	3 Jun 2008	Draft	Marco DANELUTTO
1	6 Jun 2008	Draft	Marco ALDINUCCI
2	9 Jun 2008	Draft	Marco ALDINUCCI
3	24 Jun 2008	Final	Marco ALDINUCCI

Deliverable manager

- Marco Aldinucci, UNIPI

List of Contributors

- Marco Danelutto, UNIPI
- Sonia Campa, UNIPI
- Patrizio Dazzi, ISTI/CNR
- Nicola Tonellotto, ISTI/CNR

List of Evaluators

- Françoise Baude, INRIA
- Rajkumar Buyya, U. MELBOURNE

Summary This deliverable describes the methodology and the strategies used to derive the performance models used within the autonomic managers of behavioural skeletons modelling common parallelism exploitation patterns within GCM.

In particular, we describe the general features of the *functional replication* behavioural skeleton as representative of a whole class of GCM composite components modelling classical parallel patterns. Then we outline the methodology used to derive performance models. Eventually, we will shortly discuss the results achieved by implementing autonomic controllers that used that performance models to autonomically take care of performance tuning of these behavioural skeletons.

Contents

1 Introduction

Programmers developing component based grid applications are often interested in achieving high performance. After application design and implementation, and after the usual functional debugging, a *performance tuning* step is usually needed to reduce bottlenecks and to exploit available resources. In order to be able to tune performance, an idea of the theoretical performance of the application is needed. Then, in case the performance measured is less than the ideal one, both application and system[1] programmers may look for inefficiencies and consequent code transformations aiming at improving the performance of the code. In case the performance turns out to be really close to the ideal one, instead, the programmers may just refine those small details needed to match user requests and then they may deliver the application to production.

Even in case performance is not a primary concern, programmers would like to have at least an estimation of the application running time on the available resources. As an example, they may wish to be aware that faster resources are needed in case the achieved completion time of the application is not a suitable one.

Having a precise idea of the performance of a given application whose source code is known[2] is a very hard task. It can be demonstrated, for instance, that optimal mapping of generic process graphs (or component assemblies) on generic network of processing resources (or grids) is a NP-complete problem. As a consequence, figuring out the best performance (the one coming from the execution of the best component configuration) is itself an hard task. In addition, this problem happens to be statically unsolvable if the execution framework (platforms and networks) may unevenly change along time (as happens in grids).

This notwithstanding, some good approximation or even some kind of *exact* values can be derived in case the computation parallel pattern is not fully general, but rather belonging to a set of *well-known* patterns, that possibly cover (alone or properly nested) most of the parallelism exploitation patterns used in grid applications. In this case, either we are able to have a heuristic way to determine the expected performance starting with some base parameters, or, at least, we will have an idea of the effect of the variation of some basic feature of the program implementation on the overall program performance. In the former case, we can gather the needed parameters, instantiate a formula and compute the expected performance to be compared with the one actually measured. In the latter, we can try to set up heuristics that attempt to improve/tune program performance even without having a precise *quantitative* idea of the performance improvement achieved.

Notice that the idea of founding application development on a set of *restricted* pattern for parallelism exploitation is not uncommon. The whole algorithmic skeleton community, as well as the community working on parallel design patterns and the one interested in coordination languages come up with the idea that a well designed set of (restricted) parallelism exploitation patterns (or skeletons, or coordination constructs) may couple an high expressive power with a very efficient implementation [7, 13, 11, 5, 3].

The concepts from skeleton/pattern research areas can be easily migrated to component frameworks. In GCM, and in the related research activities within CoreGRID and GridCOMP, several groups worked in the direction of considering algorithmic skeleton for efficiency. Here we want to show how the skeleton concepts can be exploited to derive suitable performance models for generic (component and skeleton based) parallel/distributed grid applications, in such a way some performance tuning activities can either be completely delegated to the programming tools and to the programming environment implementation, or proper tools can be provided to skilled user that allow performance tuning to be performed manually, knowing in advance the best theoretical performance.

Therefore in this project we did not take into account generic methodologies to derive performance models for generic, possibly pre-existing, third party components. This will be very difficult to achieve, due to the poor possibilities the user has to introspect actual implementa-

[1]The ones implementing the programming framework.

[2]When using components this is a quite strong assumptions as components are *black boxes*, actually.

tion of a component, and eventually useless, unless the basic components can be assembled in a larger composite component whose semantics is under the programmer control as well as its precise performance model.

In this document, we will show how by restricting the kind (pattern) of parallel computation implemented using components we will be able to derive performance models for some notable composite components that can be used for performance debugging/tuning of generic parallel grid applications. The resulting methodology is basically made up of three distinct steps: i) precise understanding of parallel pattern[3] semantics, ii) derivation of an estimate performance model, combining some parameters coming either from the target architecture or from the application components, and iii) usage of the performance estimate to tune/debug the application.

2 Performance model targets

When dealing with *performance*, we should specify which is the performance measure of interest. Roughly speaking, performance of a (parallel) program may be intended as the time spent to see the program terminating, namely the time to compute the program result. This kind of performance is the one of interest for the doctor waiting for the CAT[4] image to appear on the video immediately after the patient body has been irradiated by a convenient X-ray dose. However, this is not the only measure of interest. Another convenient performance measure is the *service time* of an application, i.e. the time spent by an application to deliver two consecutive results (or to accept two consecutive input items). This is the kind of measure for the technician processing all the CAT images for storage (e.g. compressing them). In this case, the time spent processing the single image is not so important, provided we can process the bunch of images coming from (possibly different CAT machines and patients) in real time, that is in less than the time needed to store the single image on the disk/mass storage.

In computer science, the former kind of measures is often referred to with the term *latency*, while the second one is referred to with the term *throughput*.

Here we show how proper performance models can be derived for structured parallel components that model throughput. Similar techniques can be used in case latency is of stronger interest, but we will not address this here as the very same methodology outlined in Sect. 5 can be used.

Figure 1 shows a typical component application scenario and outlines typical "components" for latency and throughput performance measures. The latency of component COMP A accepting service requests on its single *provides* (or server) port is given by the time spent to deliver the relative "answer". Component COMP A uses (concurrently) COMP B and COMP C. Therefore the latency is given by the sum of the time spent accepting the user request (1), computing locally to COMP A (2), sending a request to COMP B and to COMP C (3) and (4), waiting for COMP B and COMP C computing the results (5) and (6), receiving results from COMP B and COMP C (7) and (8) and eventually delivering the result to the user (9). Provided that COMP A may handle concurrent requests, the throughput is given by the maximum of the latency of COMP A (2) and the maximum of the latencies of COMP B (6) and COMP C (7), as the component program can be viewed as a two stage pipeline[5].

3 Behavioural skeletons

As already outlined in Sect. 1, very often efficient and scalable grid applications happen to be exploit parallelism according to well-known parallelism exploitation patterns. Within the CoreGRID Programing model Institute [8] and GridCOMP, we decided to exploit this concept

[3]the one used in the application at hand

[4]Computerised Axial Tomography.

[5]we assume communications are smaller that internal computation, here, otherwise also the communication in between the two stages should have been taken into account as one of the pipeline stages

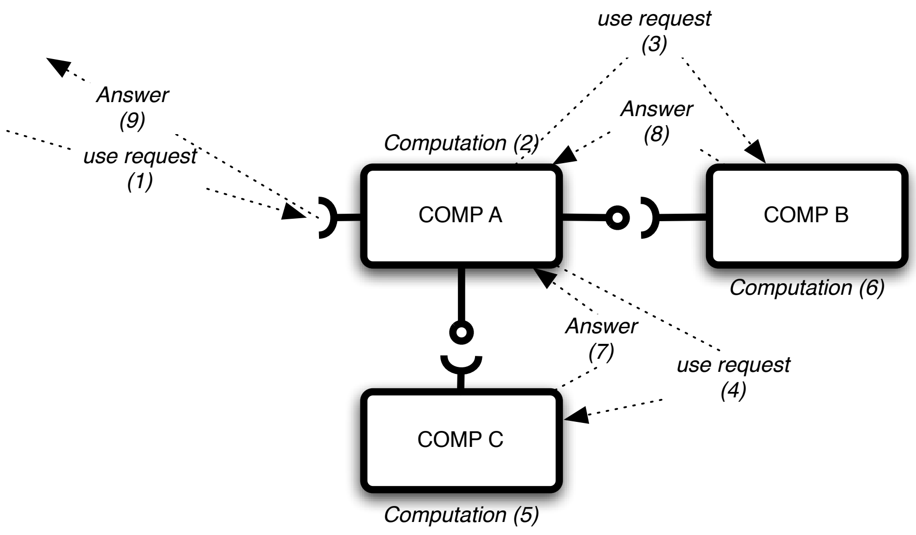


Figure 1: Component application scenario and performance measures.

far beyond what already happened in the algorithmic skeleton community [6] by introducing the *behavioural skeleton* (*BeSke*) concept [1].

A *BeSke* packs together two well-known and useful concepts:

Algorithmic Skeletons are used within a *BeSke* to represent and model the functional part of a computation, in particular the part expressing the parallel computation to be performed. Each *BeSke* assumes/models/uses a particular algorithmic skeleton. An algorithmic skeleton is a known, efficient, nestable and parametric way of expressing a parallel computation. It represents a *known* and *efficient* way of expressing parallel computations in that the parallel pattern modelled is well known, used in a large number of parallel/distributed[6] applications and efficient implementation of the skeleton are known for a large number of target architectures, including grids. Skeletons are *parametric* in the computation used to model the parallel patterns they represent. A skeleton modelling a pipeline takes pipeline stages as parameters. Therefore the skeleton is generic and the users, by providing proper parameters, can instantiate the generic skeleton to suit their particular necessities. Also, skeletons are *nestable*. Skeleton code parameters (such as a pipeline stage or a farm worker, as an example) can be themselves represented by other skeletons. The overall effect is that by providing a limited, carefully designed set of base skeletons, complex parallel applications can be expressed using quite complex (deep) skeleton trees: the root node representing to topmost parallelism exploitation pattern, the internal nodes representing more local parallel patterns and, eventually, the leaf nodes representing the sequential portions of code used in the application. Fig. 2 represents the skeleton tree for a simple parallel image processing application, using well know skeletons.

Autonomic Managers are used within *BeSke*s to implement all those controls that do not affect *what* results are computed by the applications (i.e. its functional behaviour) but rather those affecting *how* the results are computed (i.e. they manage the *non-functional* aspects of the computation). The management of non-functional features is clearly optimised with respect to the particular skeleton modelled by the *BeSke*. The management is implemented as a classical control loop (see Fig. 3). Within *BeSke*s, autonomic management [4] takes

[6]or grid, in our case. We do not claim a particular specificity of grid applications w.r.t. parallel and/or distributed applications running on more traditional target architectures, for what concerns the parallelism exploitation patterns used

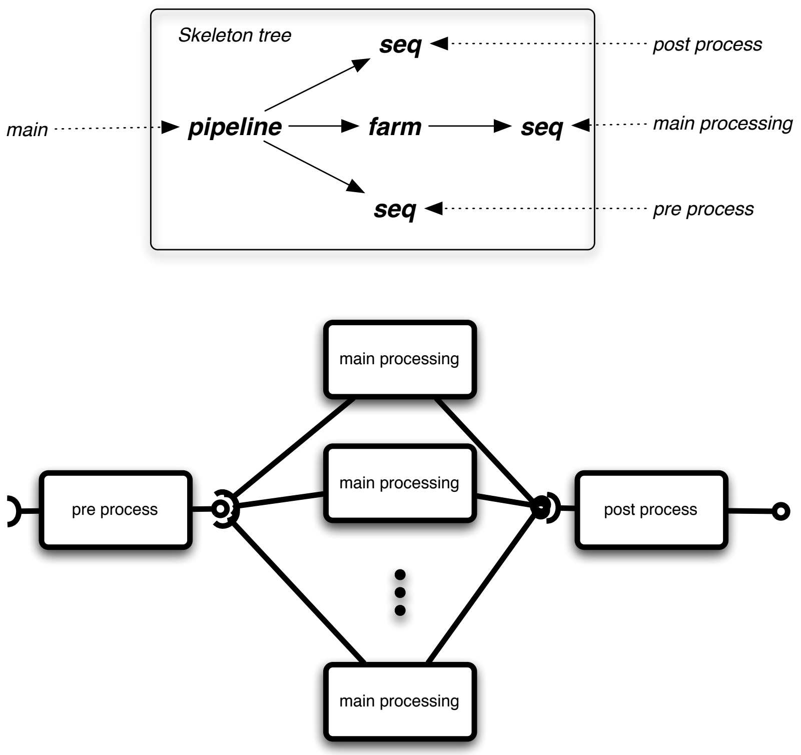


Figure 2: Sample image processing application. Skeleton structure (top) and possible component based implementation exploiting GCM collective ports (bottom).

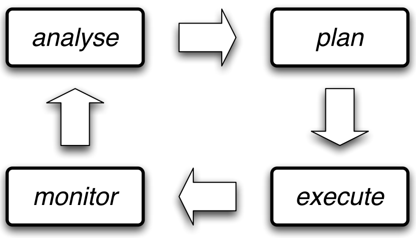


Figure 3: Autonomic manager control loop.

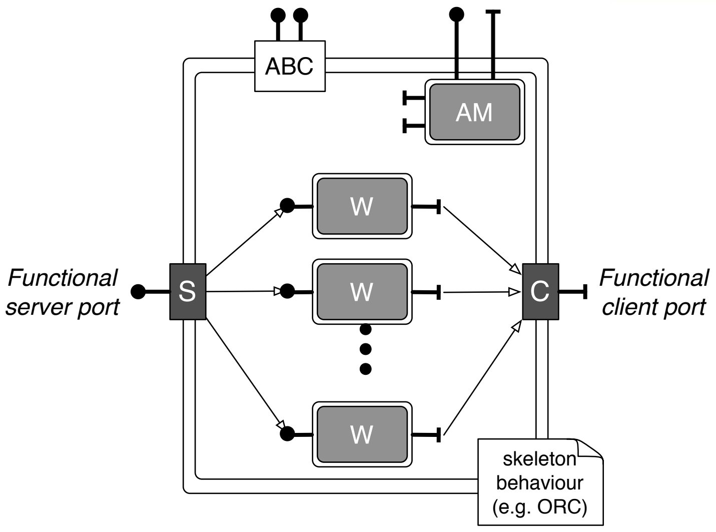


Figure 4: GCM *functional replication BeSke* family.

care of the performance aspects related to (skeleton) program execution. In particular, in the control loop:

1. in the monitor phase, some parameters are monitored, those relevant for the skeleton performance model,
2. in the analyse phase, the performance model of the skeleton is consulted, instantiated with the proper values from the monitoring phase and, in case something is going "wrong" w.r.t. the model, a corrective action is decided
3. in the plan phase, the execution of the corrective action just decided in the analyse phase is planned as a(n ordered) set of actions, and
4. eventually these corrective actions are executed in the execute phase.

More in detail (see also [2]), and taking into account that we are considering *BeSke*s in the GCM context, a *BeSke* can be represented as in Fig. 4. The figure represents a *functional replication* family of *BeSke*s. The family is represented by a *meta*-behavioural skeleton in that it can be specialised to implement a family of *BeSke*s, all those where a collection of identical *workers* (the W component in the figure) are used to process some input data appearing on the functional server (provides) port S. By specialising the behaviour of the S and C port, we can actually obtain several particularly useful instances of *BeSke*s:

task farm *BeSke* In this case the S port delivers any input item to one (and only one) of the worker components W. The C port passes each one of the input items appearing from *any* one of the workers onto the output. A task farm *BeSke* can be used to compute a function f[7] over all the elements of an input stream $\langle x_1, \ldots, x_m \rangle$ producing the result output stream $\langle f(x_1), \ldots, f(x_m) \rangle$. Whether or not the output stream preserves the ordering of the input stream is a parameter of the task farm *BeSke*.

(map) data parallel *BeSke* In this case the S port scatters any input item to all the workers. Each worker therefore receives a partition of the input data and produces a partition of the output. Port C gathers contributes to the final result from *all* the workers W and delivers onto the output the final, aggregated result. Using the same conventions adopted to describe the task farm *BeSke*, a map data parallel *BeSke* can be used to compute

$$\langle [f(x_{11}), \ldots, f(x_{1n})], \ldots, [f(x_{m1}), \ldots, f(x_{mn})] \rangle$$

[7]provided f is the function computed by the inner worker components

out of an input stream such as

$$\langle [x_{11}, \ldots, x_{1n}], \ldots, [x_{m1}, \ldots, x_{mn}] \rangle$$

fault tolerant *BeSke* In this case the S port broadcasts input values to all the workers. Each worker computes the same function f using a different algorithm. The port C gathers results from all the workers and passes to the output the "majority" result, i.e. the result computed by the majority of the workers. The fault tolerant *BeSke* can be used in all those cases where different algorithms can be used to compute the same function and we want to be absolutely sure that the "best" result possible is given as output for each one of the input stream items.

Referring to the structure of the functional replication *BeSke* family drawn in Fig. 4 we may now better outline what are the different items and which is their precise role in the *BeSke*:

S is the input server (provides) port. It accepts service request from outside and dispatches the whole input data or portions of the input data to all or to some of the inner workers according to its dispatching policy. The dispatching policy is a parameter of the functional replication *BeSke* family.

C is the output client (uses) port. It delivers results to the component whose server port is connected to this port. The delivered results can come from the single data items computed by the inner workers as well as from some kind of re-composition strategy applied to all the data items coming from the inner worker and relative to the same input data on S. The re-composition strategy is a parameter of the functional replication *BeSke* family.

W are the worker components. More precisely, these are instances of the same worker component, possibly differentiated by some input data. In case of the fault tolerant *BeSke*, these are instances of different components, actually. The common feature is that workers usually compute all the same function on the input data. Worker component(s) are parameters of the functional replication *BeSke* family.

ABC is the *Autonomic Behaviour Controller*. In the GCM jargon[8] this is a controller providing two kind of ports:

1. ports used to inspect the *BeSke* internal state. These are the ports used in the monitor phase of the autonomic cycle. As an example, a port will be provided in the ABC controller of a task farm *BeSke* that expose its service time.
2. ports used to modify the *BeSke* internal assembly. These are the ports used in the execute phase of the autonomic cycle. As an example, a port will be provided in the ABC controller of a task farm *BeSke* that increases the internal parallelism degree by inserting in the *BeSke* a fresh (new) worker component instance.

Within GridCOMP/GCM autonomic managers are currently programmed using a rule engine[9]. Therefore GCM managers are supplied with a set of rules such as:

```
rule "increaseWorkersOnLowServiceTime"
  when   $workerBean:WorkpoolBean(serviceTime >
                    userContract.requestedServiceTime())
  then   $workerBean.addWorkerToLeastUsedPE();
end
```

The rules have a pre-condition (the `when` clause) that is used to decide whether or not the rule is applicable. The rule engine executes applicable rules according to the well-known Rete algorithm [9] (priorities among rules can be specified). The rules have also an action (the `then` clause)

[8]derived from the Fractal component model [10] that represents the basis of GCM
[9]The JBoss rule engine is used [12].

expressing the actions to be taken when the rule is applicable (i.e. when the `when` condition is true). In this case, the `$workerBean` referred to in the rule is the bean actually providing access to the ABC ports.

The functional replication *BeSke* family comes with a set of pre-defined rules; nevetheless users can extend or overwrite them. It is worth pointing out that all the other parameters of the functional replication *BeSke* family are functional parameters (S, C, W) affecting the results computed by the *BeSke*, while rules supplied to the AM manager will be non-functional parameters, actually.

4 Performance models for Behavioural Skeletons

The performance goal of interest in *BeSke*s is throughput. This because within the GridCOMP (and CoreGRID) project we are mainly focusing on stream parallel applications and use cases and therefore it is quite natural to consider performance optimisation as throughput optimisation. As a consequence, we are interested in figuring out proper models that compute the service time of our GCM applications with respect to some known parameters that can be measured by monitoring the target architecture used to run the GCM application and the components building up the application itself.

However, as previously discussed, we are not interested in modelling the service time of generic components (and consequently, of generic component applications). Instead, we are interested in modelling service time of *BeSke*s and therefore of all those applications whose parallel structured is modelled using a *BeSke* (or a *BeSke* composition, in the future).

In this Section we discuss how suitable service time performance models can be derived for the class of *BeSke*s derived from the functional replication *meta BeSke*. Then, in Sect. 5 we will discuss how the procedure used to derive performance models of the functional replication *BeSke* family can be applied to more general cases, always based on *BeSke*s or on plain algorithmic skeletons in the general case.

4.1 Performance model parameters

The first aspect to be taken into account is the *kind of parameters* we will be hopefully able to monitor and measure on the target architecture and on the application at hand. Being *BeSke*s in the functional replication family defined according to the same schema, which is composed of several parts, we model their behaviour composing the behaviours of their parts. Therefore we are interested in the time spent to dispatch an input data by the S port (T_S), in the time needed to handle an output item on the C port (T_C) and on the time spent to process a single item on a worker W (T_W). As the *BeSke* obviously constitutes a part of a large application, we are interested also in the *inter arrival* time (T_A) of input tasks (e.g. the service time of the component "feeding" the functional replication *BeSke* family) and in the *capacity* (T_D) of the output connection (e.g. the service time of the component gathering and consuming functional replication *BeSke* results).

Most of these times actually depend on the input data considered. A typical example is T_W: the time spent to process two distinct data items – e.g. images in the program of Fig. 2 – may vary sensibly depending on the images considered. Being interested in throughput, we are *de facto* interested in the steady state behaviour of the application and therefore we can rely on *average* values for the parameters considered.

4.2 Performance modelling

The second aspect to take into account is the *performance modelling* of the *BeSke*. The *BeSke* in the functional replication family can be viewed and modelled as a three stage pipeline:

1. a dispatching stage, mainly using S port resources and algorithms

2. a computing stage, using the resources relative to the strand of workers, and
3. a collecting stage, mainly using C port resource and algorithms

Therefore the service time of the *BeSke* can be modelled as:

$$max\{T_S, f(T_W, n_w), T_C\}$$

being n_w the number of instances of workers in the *BeSke*. The function f depends actually on the precise kind of computation modelled by the *BeSke*. In case it is a task farm, f is simply

$$f(T_W, n_w) = \frac{T_W}{n_w}$$

whereas in case it is a map, f is

$$f(T_W, n_w) = T_W$$

as in the former case all the workers contribute to compute single items of the input stream, while in the latter a single item of the input stream (a partition of it, usually) is computed by all the workers.

4.3 Optimal performance

Eventually, we must figure out which is the *asymptotic performance* of the *BeSke*. The best performance of *BeSke*s in the functional replication family will be achieved when the service time of the *BeSke* does not exceed the inter arrival time T_A or the output link capacity T_D. As a consequence, we (the autonomic manager within the *BeSke*, actually) will aim at reducing the service time to the value

$$max\{T_A, T_D\}$$

In a task farm *BeSke*, service time can be decreased (increased) by adding (taking away) workers from the worker strand. This follows from the simple consideration that in case port S and C do not represent bottlenecks in the pipeline modelling the *BeSke*, the service time of the task farm is simply given by

$$\frac{T_W}{n_w}$$

In the map *BeSke* instead, the service time (under the same assumptions on S and C) is given by

$$T_W$$

and therefore it can only be varied by varying the service time of the single workers. To be more precise, we can't assume that all the identical workers have the same service time. Different processing resources are used to compute the different workers, and also different partitions of the same input data may lead to different computation times. As a result, in map *BeSke* we should consider

$$T_W = \forall i \in [1, n_w] : max\{T_{W_i}\}$$

This suggests that service time of the map *BeSke* can be improved by using faster resources to compute slower workers or by using a slightly modified partitioning rule giving less work (smaller partitions) to slower workers and more work (larger partitions) to faster workers.

The other technique that can be used to improve map service time is to parallelise workers[10] by using a nested task farm *BeSke* to implement each one of the workers. This is perfectly suited to the component philosophy: inner worker components are actually *BeSke*s (composite components), but they appear as regular components to the map *BeSke*.

[10]the slower ones, in case there is some slow worker and a majority of much faster workers, or the whole worker strand, in case there are no sensible differences

5 General methodology for deriving (and using) performance models

We want now to capitalise on what we discussed for the functional replication *BeSke* instances in Sect. 4. In particular we want to describe a general methodology to derive and use performance models for GCM *BeSke*s. The methodology is defined as an ordered set of steps:

1. The **first step** is relative to parallel semantics understanding. Once the *BeSke* has been defined, in order to derive a precise and worth performance model, the parallel semantics of the modelled algorithmic skeleton has to be completely understood. This means that we must figure out which are the parallel activities involved in the skeleton execution and which are the bottlenecks and parallelism sources inside the skeleton.

2. Once individuated the parallel activities, we pass to **second step**, that is to identify how the modelling of these parallel activities can be re-conducted to already known performance models for parallel patterns. As an example, in the previous section we showed how the performance model of a task farm (or of a map data parallel skeleton) can be approximated using the pipeline performance model. Most of times, the simple models for pipeline and "strand of workers", such as those used in the models for the functional replication *BeSke* family, are sufficient to model the complete performance behaviour of considerably more complex *BeSke*s. Just in case the different sub-components of the *BeSke*s cannot be modelled re-using results for other, known parallel patterns, during this second step we need to develop *ad hoc* performance models *ex novo*.

3. The **third step** is the identification of the parameters affecting the performance of models of the *BeSke* at hand. Again, most of times this is a simple activity. The parameters of interest are usually the ones of the inner, well known parallel patterns. They are quite simple[11] parameters, either relative to the target architecture or relative to the *BeSke* sub-components.

4. The **fourth step** is relative to devising the proper optimal performance for the *BeSke*. In particular, we should figure out which is the optimal performance to avoid dedicating further or faster resources to the application execution when this performance has been achieved.

5. The **fifth and last step** is related to devising proper heuristics allowing to decide which are the rules to be inserted into the manager rule engine in order to ensure user (or system) supplied performance contracts.

Overall to the five steps above lead the *BeSke* programmer to a point such that the autonomic manager in the *BeSke* can be completed inserting a proper set of rules. However, steps one to four are needed also in case of programmers that want to exploit parallel patterns but also to keep the control of what's going on in the application as far as application performance is concerned. Actually, step five is also necessary, in this latter case, as programmers implementing their own performance optimisation/tuning should know which rules (strategies) have to be applied when tuning application performance.

5.1 Methodology application sample

We detail the steps of the methodology here by figuring out proper performance model(s) for a particular kind of data parallel *BeSke*. We consider a skeleton that can be derived from the functional replication *BeSke* with a limited set of changes. Given an input stream such as

$$\langle s_1, [x_{11}, \dots, x_{1n}], \dots, s_m, [x_{m1}, \dots, x_{mn}]\rangle$$

[11]from the conceptual viewpoint, not from the viewpoint of how they can be collected from the target architecture and from the application components

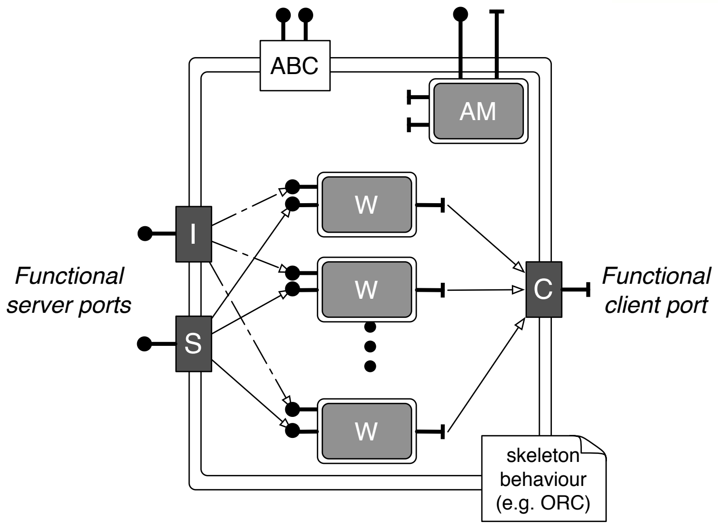


Figure 5: GCM *map with workers specialised with state BeSke.*

the *BeSke* computes the output stream

$$\langle [f(x_{11}, s_1), \ldots, f(x_{1n}, s_1)], \ldots, [f(x_{m1}, s_m), \ldots, f(x_{mn}, s_m)] \rangle$$

where f is a pure function, and, as a consequence, there are no side effects on the "state" values s_i. This *BeSke* models a quite common parallel pattern: a map data parallel where the workers are specialised by a kind of *read only* state parameter.

This *BeSke* can be implemented in GCM as outlined in Fig. 5. In this case, the additional input port I is used to pass the s_i state parameters. The I port scatters the received state value to the workers and in the meanwhile inhibits port S from passing further values to the workers. As soon as the workers receive the new state value, they complete the computation of the current tasks, then update their own variable hosting the state value and eventually signal that computation can be safely restarted. This signal will eventually unblock port S.

This *BeSke* models quite common parallel patterns. As an example, the biometric identification [14] use case within GridCOMP happens to be perfectly modelled by the skeleton. In this case, the input stream will be something as

$$DB_1, fp_{11}, \ldots, fp_{1m_1}, \ldots, DB_k, fp_{k1}, \ldots, fp_{km_k}$$

and the *BeSke* will compute the output stream

$$y_{11}, \ldots, y_{1m_k}, \ldots, y_{k1}, \ldots, y_{km_k}$$

where

$$y_{ij} = f(DB_i, fp_{ij}) = \begin{cases} \langle true, id \rangle & \textbf{iff } \exists \langle fp_t, id \rangle \in DB_i \wedge \textbf{match}(fp_{ij}, fp_t) = \textbf{true} \\ \langle false, \_ \rangle & \textbf{otherwise} \end{cases}$$

Now, let us proceed through the performance modelling derivation methodology steps as defined in Sect. 5.

Step 1 The parallel semantics of this *BeSke* is the following: there is a set of parallel activities corresponding to the worker component instances and to the I, S and C ports. I, S and the workers need to synchronise while receiving and processing one "state" value, otherwise synchronisation is only given by plain data flow. S broadcasts input values (non state ones) to all the workers. Workers compute a partial contribute to the final result (whether or not the given, broadcast fingerprint is present in their data base portion) and the C port "reduces" the input values to a single output value using the following binary, associative and commutative operator:

$$\begin{array}{lcl} \oplus(\langle false, \_\rangle, X) & = & X \\ \oplus(\langle true, id\rangle, \_) & = & \langle true, id\rangle \end{array}$$

Step 2 at the steady state, the performance model of the skeleton can be derived thinking to S, the strand of W and C as three stages of a pipeline, as we did to model task farm *BeSke*. However, in this case, some overhead due to the initialisation of state variables is to be taken into account. If the m_i are big, the overhead can be probably ignored. If not, it may be evaluated adding a

$$(T_I + T_W)/m_k$$

contribute to the T_W[12].

Step 3 Parameters of interest in this case are the same of interest for the task farm *BeSke* (service time of the worker components, of the S and C ports, plus the inter-arrival time, etc.) plus the time spent by port I to scatter state to the workers (this time includes also the synchronisation time needed to block S, and to reactivate it after worker flushing).

Step 4 The workers, in this case, perform a matching of a single fingerprint against all the fingerprints in the portion of database received from port I. Therefore $T_W = g \times T_{match}$, where g is the *dimension* of the data base portion and T_{match} is the time spent to match one fingerprint. The optimal performance for the *BeSke* will be achieved when the time spent to send fingerprints (sets) to the workers is almost the same used to match the fingerprints against the local portion of the data base, assuming that m_k are big and that the other times in the *BeSke* pipeline are negligible. Therefore g turns out to be a parameter clearly affecting the service time of the *BeSke*, an it will be considered to adapt service time in the manager.

Step 5 The rules included in the manager rule engine could include:

- a rule stating that if the *BeSke* performs less than required by the user contract *and* $T_W > T_S$ *then* the number of workers is increased (e.g. by one);
- a rule stating that if the *BeSke* performs less than required by the user contract *and* $T_W \leq T_S$ *and* there are available more powerful resources to host the worker component execution, *then* worker components must be migrated to the faster resources;
- a rule stating that if the *BeSke* performs less than required by the user contract *and* $T_W \leq T_S$ *and* there are not available more powerful resources to host the worker component execution, *then* nothing can be done, and a contract violation should be reported to the user.

6 Performance models for plain components (not behavioural skeletons)

As stated before, we are not interested in deriving performance models for any generic component application and therefore we are not interested in modelling performance of any generic component. However, to be able to model performance of *BeSke*s we need to be able to get some (at least approximated) values for the measures of interest of the components used as parameters of the *BeSke*.

[12]this corresponds to spreading the initial overhead (time spent to sent the state partition to the workers plus the time needed to flush the worker computations relative to the old state value, which is at most T_W) over the computations of all the items logically referring to that state value

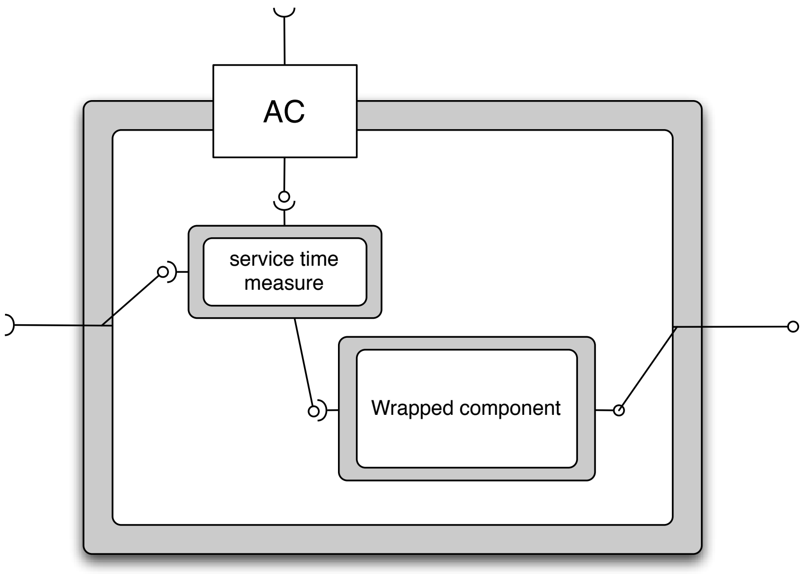


Figure 6: Wrapping components for providing ABC.

One of the parameters of interest is the service time of a component (port). This time can be asked to the component, if the component is a *BeSke*, through the ABC controller. However, if the component is not a *BeSke*, this value should be obtained differently.

To this purpose, we assume to exploit component wrapping technology. Each component is wrapped with a standard component provided by the *BeSke* framework, such that (see Fig. 6):

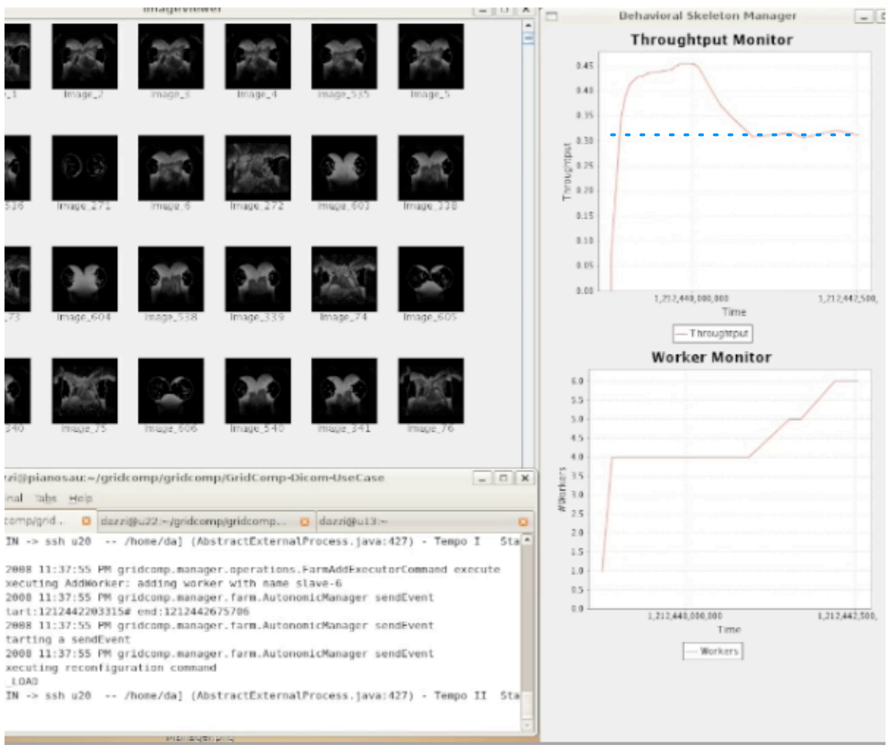
1. all the ports of the component are simply propagated as ports of the wrapping component (both client and server ports, both the ones related to functional and the ones related to non-functional interface)

2. the wrapper component (*service time measure* component in the Figure) measures the times required by the wrapped component to answer to a service request coming onto a server (provides) port

3. the wrapper component implements an ABC controller providing the ports needed to fetch the service time and the other measures gathered in the wrapper component, according to the standard *BeSke* non-functional interface API.

7 Results relative to the performance modelling of behavioural skeletons

We actually implemented *BeSke*s exploiting the performance models described in this work. Deliverable D.NFCF.04 *Early prototype and documentation* describes this implementation. The kind of results achieved can be figured out looking at Fig. 7. The Figure shows the output of a medical image processing application. In the left, top window, the processed medical images thumbnails are shown, in such a way the doctor may click on the thumbnails to see and evaluate the real image. In the right window, throughput and resources used (number of workers used by the application) are plotted as managed by the AM in the *BeSke* used (a task farm *BeSke*, in this case). Initially, a number of processing elements were used to run the application. After half of the total time plotted, additional load was delivered onto these (shared) resources due



```
zi@pianosau:~/gridcomp/gridcomp/GridComp-Dicom-UseCase
IN -> ssh u20  -- /home/da] (AbstractExternalProcess.java:427) - Tempo I   Sta

2008 11:37:55 PM gridcomp.manager.operations.FarmAddExecutorCommand execute
xecuting AddWorker: adding worker with name slave-6
2008 11:37:55 PM gridcomp.manager.farm.AutonomicManager sendEvent
tart:1212442203315# end:1212442675706
2008 11:37:55 PM gridcomp.manager.farm.AutonomicManager sendEvent
tarting a sendEvent
2008 11:37:55 PM gridcomp.manager.farm.AutonomicManager sendEvent
xecuting reconfiguration command
_LOAD
IN -> ssh u20  -- /home/da] (AbstractExternalProcess.java:427) - Tempo II  Sta
```

Figure 7: Adaptation of a *BeSke* exploiting performance models.

to different users and applications. The load caused a sudden decrease in throughput and the manager intervened to prevent throughput falling down the user-supplied value by adding more and more workers to the *BeSke*. This eventually kept the throughput in the range asked by the user.

Observe that:

- adding new workers was considered a useful strategy in the manager as the monitored application values and the *BeSke* performance model indicated we were not at the saturation point (bandwidth of the worker strand equal to either T_A or T_D);
- the time spent adding new workers was sensibly smaller than the time expected remaining time to conclude the computation. If not, adding workers would have resulted in pure overhead.

8 Conclusions

We have discussed a methodology aimed at designing performance models for a particular class of components implemented in GCM: the *BeSke* composite components. These components provide the user with parametric parallel exploitation patterns that can be used/instantiated to build a full range of (possibly complex) grid parallel applications. The methodology allows deriving performance models for these kind of skeletons and the performance models can be exploited within the *BeSke*s to implement efficient autonomic managers taking care of performance aspects in the *BeSke* execution. The development of performance models for *BeSke*s has been already demonstrated in the case of the task farm *BeSke* and in case of plain data parallel *BeSke*s derived from the *BeSke*s in the functional replication family discussed in Sect. 3.

We are currently working to provide other *BeSke*s to the GCM users and all of these new skeletons will require designing proper performance models to be exploited within their autonomic managers.

References

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Patrizio Dazzi, Peter Kilpatrick, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons for component autonomic management on grids. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete, Greece, June 2007.

[2] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Patrizio Dazzi, Domenico Laforenza, Nicola Tonellotto, and Peter Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In Didier El Baz, Julien Bourgeois, and Francois Spies, editors, *Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing*, pages 54–63, Toulouse, France, February 2008. IEEE.

[3] Marco Aldinucci, M. Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, December 2007.

[4] IBM autonomic computing web page, 2008. `http://www.research.ibm.com/autonomic/overview/`.

[5] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible skeletal programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *Proc. of 11th Intl. Euro-Par 2005 Parallel Processing*, volume 3648 of *LNCS*, pages 761–770, Lisboa, Portugal, August 2005. Springer.

[6] M. Cole. Skeletal Parallelism homepage, 2008. `http://homepages.inf.ed.ac.uk/mic/Skeletons/index.html`.

[7] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[8] CoreGRID web site, 2008. `http://www.coregrid.net`.

[9] Charles Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

[10] The Fractal Project homepage, 2008. `http://fractal.objectweb.org/`.

[11] S. Gorlatch and J. Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer, November 2005.

[12] Jboss rules home page, 2008. `http://www.jboss.com/products/rules`.

[13] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

[14] T. Weigold, P. Buhler, J. Thiyagalingam, A. Basukoski, and V. Getov. Advanced grid programming with components: A biometric identication case study. In *Proc. of the 32nd Intl. Computer Software and Applications Conference (COMPSAC)*. IEEE Press, 2008. to appear.