View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Electronic Thesis and Dissertation Archive - Università di Pisa

Autonomic Management of Performance in FastFlow Stream Parallel Patterns

Samson Hailu Tesfay

Master of Science in Computer Science and Networking

University of Pisa and Sant'anna School of Advanced Studies

Supervisor: Professor Marco Danelutto

December 4, 2015

Acknowledgements

I would like to thank my supervisor Prof. Marco Danelutto for his valuable support and patience throughout the thesis. I would never have been able to finish my dissertation without his guidance. I would like to thank my family: my parents and to my brothers and sisters for supporting me spiritually throughout my life.

Contents

1. Introduction..4
   1.2 Thesis objectives..5
   1.3 Structure of the Dissertation...5
2. Background...7
   2.1 Stream Parallel computations...7
   2.3 The FastFlow Algorithmic Skeleton Framework...9
      2.3.1 FastFlow Skeletons and sequential concurrent activities.......................11
   2.3 Related work..16
   2.3.1 Autonomic Computing...16
   2.3.2 Autonomic Management of Non-functional Concerns in Structured Parallel Programming...19
3. Logical Design...31
   3.1 High level view..31
      3.1.2 General Overview of the Autonomic Behavior.......................................32
   3.2 Logical design of the Task Farm Behavioral Skeleton.................................33
   3.3 Logical Design of the Pipeline Behavioral skeleton....................................34
4. Implementation..36
   4.1 Autonomic Manager Implementations..36
   4.2 Autonomic Controller Implementation..37
      4.2.1 Mechanisms provided by lower level FastFlow(Layer 2)......................37
      4.2.2 Monitoring and execution mechanisms in the Task Farm Skeleton.......37
      4.2.3 Monitoring and Execution mechanisms in the pipeline skeleton pattern.............40
   4.3 AM Policies...42
      4.3.1 Policies for the Farm Behavioral skeleton..42
      4.3.2 Policies for the Pipeline Behavioral skeleton...43
5. Experiments...45
   5.1 Experimental Settings..45
   5.2 Experiments on the Farm Behavioral Skeleton..45
      5.2.2 Test cases:...46
   5.3 Experiments on the Pipeline Behavioral Skeleton.......................................48
      5.4 Summary..53
6.Conclusions...54
   6.1 Future Work...54
Bibliography...56
Appendix 58

Chapter 1
Introduction

Over the past few decades, the computer industry has been focused on manufacturing of single central processing units (CPU) with higher frequency and complexity to improve performance. However, this progress has been slowing down due to energy-consumption and heat-dissipation issues that have limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU [22].

The use of multiple, simpler processing elements, or cores has become the main focus of the processor industry, as it was the only viable way to sustain the increase in processor performance. By providing multiple cores, separate parts of the program can be executed in parallel .

However, performance heavily relies on the ability of the program to fully utilize all the cores. This requirement adds additional complexity on development of software applications, making parallel programming mandatory. This led to the design of parallel abstractions focusing on hiding details from the programmer. Parallel design patterns [23] and Algorithmic skeletons introduce such abstractions.

Algorithmic skeletons, introduced by Cole in [19] are high level programming model for parallel and distributed computing. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. While algorithmic skeletons and parallel design patterns provide implementation of well known implementation patters, usually non-functional features such as such as performance, security, fault tolerance, and power management are handled by the user.

Behavioral skeletons have been introduced in early 2000s with the aim of supporting autonomic management of non functional features related to skeleton implementation. A behavioral skeleton is the result of the co-design of a parallelism exploitation pattern and of an autonomic manager, taking care of some non functional feature related to the parallelism exploitation pattern implementation[11].

This thesis presents the implementation of a prototype behavioral skeleton for stream parallel algorithmic skeletons in the FastFlow taking care of service time and efficiency.

1.2 Thesis objectives

The main objective of this thesis focuses on developing behavioral skeletons on top of existing FastFlow algorithmic skeletons that are capable of taking care of non-functional concerns. Though non-functional concerns in skeletal parallel programming include several concerns such as performance, security, fault tolerance, and power management, this thesis only focuses on performance, specifically optimizing service time and/or efficiency by choosing the optimal parallelism degree. Among the skeletons provided by the FastFlow algorithmic skeleton frame work only stream parallel skeletons for multi-core architectures namely the pipeline and task farm skeletons are covered. In the case of task farm skeleton optimal parallelism degree is achieved by increasing and decreasing the number of workers in the task farm. Where as in the case of pipeline skeletons optimal parallelism degree is achieved by merging pipeline stages and splitting previously merged stages.

Reconfiguration mechanisms to increase and decrease number of workers in the task farm skeleton are implemented. For the pipeline skeletons mechanisms that enable to merge consecutive stages and split previously merged stages are introduced. In addition to the reconfiguration, mechanisms that enable to monitor the internal state of the skeletons (i.e. to query the service time of internal components), are also implemented.

Moreover an autonomic manager associated with each skeletons, controlling the parallelism degrees is introduced based on simple hard codded policies is implemented. By choosing optimal parallelism degree (i.e. number of workers in a farm and number of stages in pipeline) the autonomic manager aims to improve the performance(service time/efficiency) of the skeletons,

Experiments conducted to validate and asses the functionalities of the prototype behavioral skeletons are presented and results achieved are discussed.

1.3 Structure of the Dissertation

The rest of this thesis is organized as follows:

- Chapter 2 provides a relevant background material for the thesis including stream parallel computations and algorithmic skeletons. A short introduction to the FastFlow algorithmic skeleton is provided. Then the chapter presents related work on autonomic management non-functional concerns in structured parallel programming and behavioral skeletons.
- Chapter 3 presents high-level view, and logical design of the prototype implemented in this thesis.
- Chapter 4 presents a detailed discussion of the implementation of the prototype behavioral skeleton on top of the existing stream parallel skeletons, including mechanisms

used and policies applied by the autonomic manager.

- Chapter 5 presents the experiments conducted on the prototype implementation of pipeline and task farm behavioral skeletons, and results from the experiments are discussed.
- Chapter 6 discusses the conclusions of the thesis and future works are discussed.

Chapter 2

Background

This chapter explores relevant background material for the thesis, starting with a brief discussion of Stream parallel computations in Section 2.1, followed by the discussion of algorithmic skeletons in Section 2.2 . Then Section 2.2 provides a detailed description of the FastFlow algorithmic skeleton frame work. Finally Section 2.3 presents related works focusing on the concept of autonomic computing and Autonomic Management of Non-functional Concerns in Structured Parallel Programming and introducing Behavioral Skeletons.

2.1 Stream Parallel computations

Stream Parallelism is method for parallelizing the execution of a stream of tasks by segmenting the task into a series of sequential or parallel stages. This method can be also applied when there exists a total or partial order, respectively, in a computation preventing the use of data or task parallelism. This might also come from the successive availability of input data along time (e.g. data flowing from a device). Parallelism is achieved by running each stage simultaneously on subsequent or independent data elements[18].
The following are some properties common to stream parallel computations from [13]

1. Large Streams of data: One of the fundamental properties of streaming computations is that they operate on a large sequence of data items . Data streams generally enter the program from some external source, and each data item is processed for a limited time before being discarded
2. Independent Stream Filters: A streaming computation represents a sequence of transformations on the data streams. The transformations (also referred as *filters*) are generally independent and self-contained, without references to global variables or other filters. A stream program is the composition of filters into a stream graph, in

which the outputs of some filters are connected to the inputs of others.

3. A stable computation pattern: The structure of the stream graph is generally constant during the steady-state operation of a stream program. That is, a certain set of filters are repeatedly applied in a regular, predictable order to produce an output stream that is a given function of the input stream.

4. Occasional modification of stream structure: Even though each arrangement of filters is executed for a long time, there are still dynamic modifications to the stream graph that occur on occasion. For instance, if a wireless network interface is experiencing high noise on an input channel, it might react by adding some filters to clean up the signal.

2.2 Algorithmic skeletons

An algorithmic skeleton is a is parametric, reusable and portable programming abstraction modeling a known, common and efficient parallelism exploitation pattern. The concept was first introduced by Cole In the late 80s and [19,20]. They simplify the task of parallel programming by abstracting commonly-used patterns of parallel computation, communication, and interaction while offering simplicity,portability, re-use, performance, and optimization [21].

An algorithmic skeleton framework provides set of pre-defined patters encapsulating the structure of a parallel computation that are provided to user as building blocks to be used to write applications [11]. Each skeleton corresponded to a single parallelism exploitation pattern.

Provided with an algorithmic skeleton, the programmer is not required to rewrite the code related to parallelism exploitation, when writing parallel applications. He/she can structure his parallel computation by instantiating the skeletons, rather than rewriting the parallelism patterns from scratch.

By using an algorithmic skeleton framework, the programmer gains a range of benefits including[11] simplification of parallel application development (as parallel programming mainly consists of properly instantiating skeletons provided), portability on different target architectures by only recompiling the parallel application on target architectures, simplification of debugging (as only the sequential code has to be debugged).

In section 2.3 algorithmic skeletons provided by the FastFlow algorithmic skeleton are

briefly discussed.

2.3 The FastFlow Algorithmic Skeleton Framework

FastFlow is a C++ algorithmic skeleton framework targeting heterogeneous platforms. It provides programmers a suitable parallel programming patterns which are compiled into networks of parallel activities on target architectures. Conceptually it is designed as a stack of five abstraction layers abstraction layers[16] as shown in Fig 2.1.

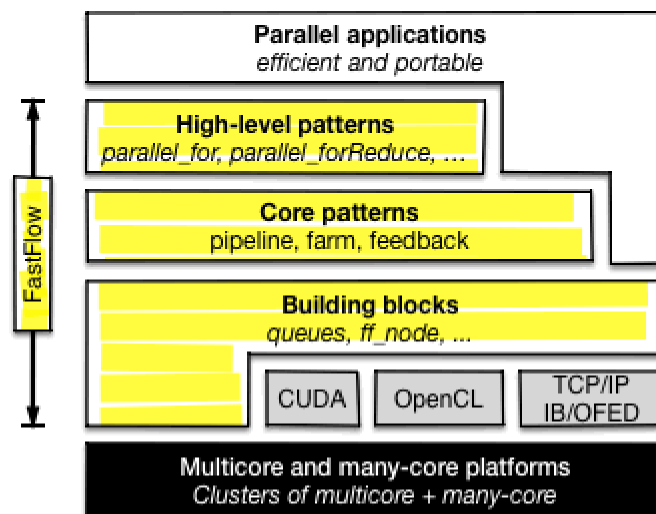


Figure 2.1. FastFlow architecture layers

1. Hardware: At the bottom of the layer are platforms that are targeted by FastFlow such as multi-core, many-core, and clusters of multi-core and many-core possibly equipped with computing accelerators. Initially FastFlow was designed to target multi-core shared memory architectures but has been extended to support distributed

and GPGPU platforms.
GPGPUS are supported through CUDA and OpenCL, where kernel business code is written in those languages . Distributed platforms are build on top of TCP/IP and Infiniband protocols.[16]

2. Building blocks: At this layer the programming model is a hybrid shared-memory/message-passing model; where processes (process containers) are sequential and channels are true dependency between precesses. Processes stream data items to channels and the data items act as synchronization tokens.
FastFlow channels define simple streaming networks whose tun-time support is implemented through lock-free Single-Producer-Single-Consumer queues with non-blocking `push` and `pop` operations [17]. The synchronization overhead of those methods is minimal due to the absence of locks.

   In addition FastFlow provides non-blocking, lock-free Single-Producer-Multiple-Consumer(SPMC), Multiple-Producer-Single-Consumer(MPSC) and Multiple-Producer-Multiple-Consumer(MPMC) queues which can be used to build arbitrary streaming networks. These queues are built on top of the lock free Single Producer Single Consumer queues and an arbiter thread:
Figure 2.2 shows the queues built on top of the SPSC queue

   In addition processes and thread containers are implemented as C++ classes built on top of POSIX threads/processes.

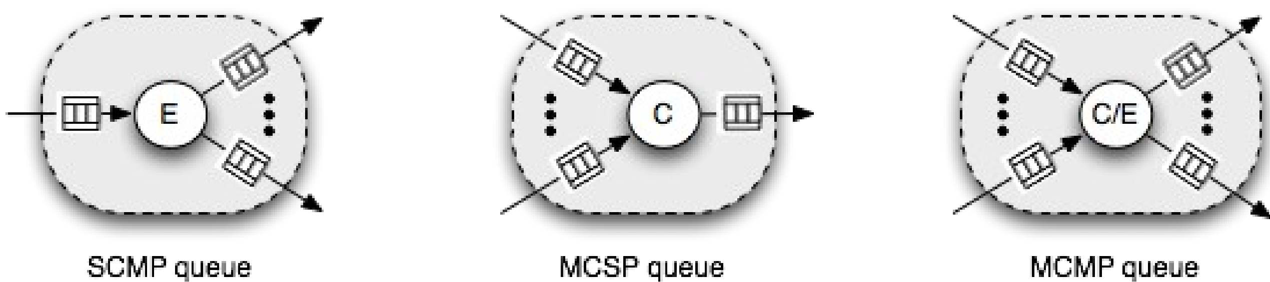


Figure 2.2 SCMP, MCSP and MCMP queues built on top of the SPSC queue

3. Core Patterns: this layer provides a general data-centric parallel programming model with its run-time support, which is designed to be minimal and reduce to the minimum typical sources of overheads in parallel programming. This level provides

two parallel skeletons: the task farm skeleton and the pipeline skeleton, which are built using collection of threads/processes that communicate using queues.

4. High level patterns: this layer provides higher level parallel patterns build on top of the Core Patterns including loop (i.e. *parallel for*), data parallel patterns (such as *map, stencil, stencil reduce*), macro data-flow etc . Those patterns are built on top of he Core Patterns.
5. Parallel Applications: On top of the lies the parallel applications. Parallel application programmers can write efficient applications by directly exploiting the parallel patterns of FastFlow provided at the “Core Patterns” and “High level patterns” layers. This is done by defining sequential concurrent activities for sequential activities, and instantiating the parallel patterns with those activities.

2.3.1 FastFlow Skeletons and sequential concurrent activities

2.3.1.1 Sequential concurrent activities

The *ff_node* sequential activity abstraction provides a way do define a sequential activity that process data items appearing on a single input channel and that delivers the related results to the single output channel. The ff_node is an abstract class where the user is supposed to provide the sequential code by extending this class. This class provides a number of methods, among those the following three methods have a particular importance.

1. virtual void* svc(void* task) = 0;
2. virtual int svc_init(){return 0}
3. virtual void svc_end(){}

The svc method is the one defining the behavior of the node while processing the input stream data items. The svc_init method is invoked automatically by the FastFlow run time support when concurrent activity represented by the node is started, while the svc_end method is invoked right before termination. Both of them are only invoked once during the lifetime of the node. The following code snippet shows the usage of a FastFlow *ff_node*

```
class myNode : public ff_node {
public:
```

```
int svc_init(){
        //initialize the node
     return 0;
         }

void* svc(void*){
        //behavior of the sequential concurrent activity
         }
void svc_end(){
        //finalize the stage,
         }

};
```

2.3.1.2 Skeletons provided in FastFlow

FastFlow provides three types of skeletons, which are Stream-parallel skeletons, Data-parallel skeletons and Data-flow skeletons [14].

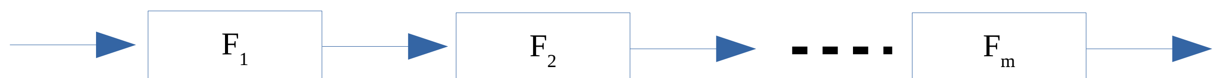
Stream Parallel Skeletons

Stream parallel skeletons exploit parallelism in the computation of input streams, possibly available at different times. Stream parallel skeletons in FastFlow consist of pipeline and task farm skeletons.

Pipeline Skeleton

A pipeline skeleton is used to model computations that can be expressed as stages. Normally a pipeline parallel application can have two or more stages . Given a stream of inputs $x_n, \ldots x_{2,} x_1$ a pipeline with stages $F_{1,} F_{2,} \ldots F_m$ computes the output stream

$$F_1(\ldots F_2(F_1(x_n))), \ldots, F_1(\ldots F_2(F_1(x_1)))$$



Each input passes through each stage in the same order of arrival, F_n processes the output from F_{n-1} while, F_n is processing on other inputs, hence the stages execute in parallel. The latency of the pipeline is equal to the sum of the latencies of all the stages since a single input have to go through all stages. That is:

$$L_{pipe}=\sum_{i=1}^{m} L_i$$

and the service time of the pipeline is the service time of the latest stages. That is:

$$T_{pipeline}(F_1, F_2, \ldots F_n)=max(T_{F_1}, T_{F_2}, \ldots, T_{F_n})$$

FastFlow's ff_pipeline class implements the pipeline pattern with fixed number of stages, which is constructed from a fixed number of threads, each representing one stage, connected by lock-free SPSC queues.

To create a pipeline of n stages in fast flow, one must first create an instance of *ff_pipeline*, then instantiate n different *ff_node* objects and add them to the ff_pipeline by calling the add_stage method.

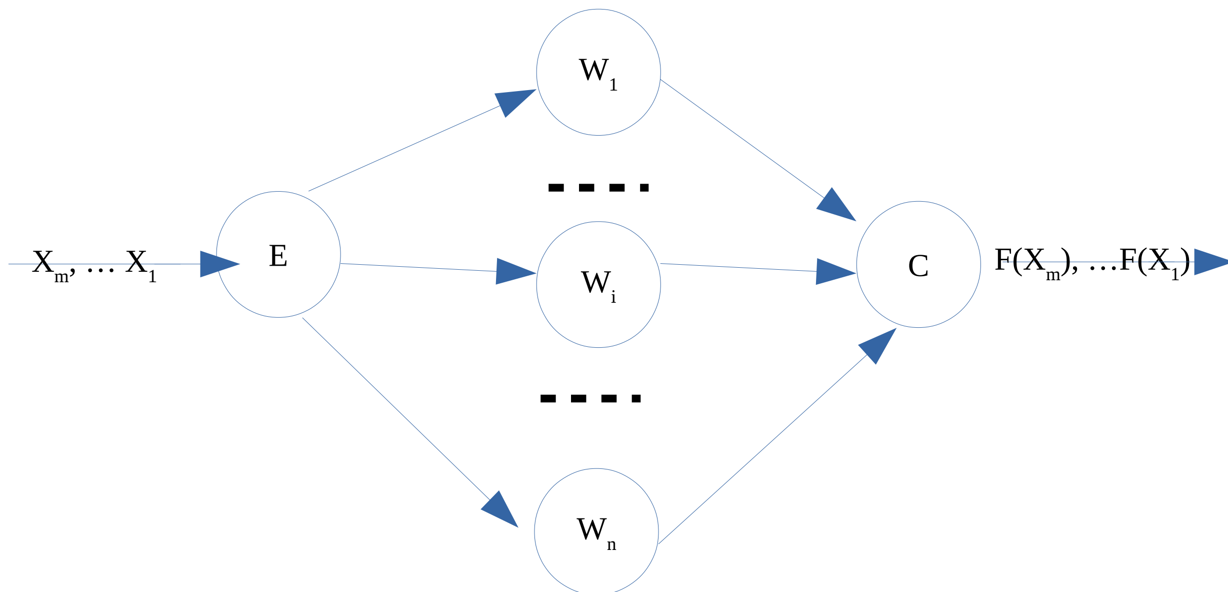
In addition a variant of a pipeline skeleton namely “pipeline-with feed back” is available where the output of the last stage is directed as the input of the first stage, forming a ring like structure. Once after instantiating an *ff_pipeline* object the programmer can set this functionality by calling the “wrap_around” method.

Task Farm Skeleton

The task farm also known as master-worker is a stream parallel paradigm based on the replication of purely functional computation, with out the knowledge of internal structure of the function itself.

An emitter component is used to schedule incoming tasks to the workers. The scheduling policy can be either a static one, such as a round robin scheme or a dynamic one, where tasks are scheduled to the workers on demand. In addition a collector component is used to collect the out put of the workers. If necessary the collector can be implemented in a way that reorders the results computed by the workers. The task farm can also exist without the collector, where the workers consolidate the results in memory or write them to storage. An other variant is where the workers send back the results to the emitter.

The following figure shows the structure of a task farm skeleton:



In FastFlow the *ff_farm* template class provides the implementation of the task farm skeleton. The programmer should first create an instance of an *ff_farm*, then the functional code of the workers, emitter, and collector should be implemented as sub type of the *ff_node* class. And finally their instances can be added by calling their corresponding *add_* methods. The following code snippet shows the usage of the farm skeleton in Fast Flow:

```
#include <ff/farm.hpp>

class Emitter : public ff_node {
public:
void* svc(void* t){
         …
          }
};

class Worker : public ff_node {
public:
void* svc(void*){
```

```
         …
          }
};

class Collector : public ff_node {
public:
void* svc(void*){
         …
          }
};

int main(int argc, char *argv[]) {
 …

ff_farm farm;
farm.add_emitter();
std::vecot<ff_node*> workers;
 for(int i=0; i<nWorkers;++i)
      workers.push_back(new Worker);
farm.add_workers(workers);
 …

}
```

Data parallel Skeletons:

Data parallel skeletons refer to the group of algorithmic skeletons on the computation of different subtasks obtained by spiting a larger input task.

In data parallel applications, a larger input data is partitioned among the number of concurrent resources, each computing the same function on the assigned data partition . Data parallel skeletons may work on single elements coming from a stream input, but in general, those skeletons doesn't consider stream parallelization by them selfs [11,14] . The main goal of data-parallel skeletons is to minimize the completion time of a single task to be computed. Data parallel skeletons in FastFlow are implemented on top of the Farm stream parallel skeletons . FastFlow provides map, ParallelFor, stencil, ParallelForReduce and stencil

reduce. The ParallelFor skeleton is used to parallelize having an independent iterations, its computation is similar to that of a map skeleton. The ParallelForReduce is used to perform a parallel-for computation followed by a reduction operation by allowing the user to provide a combiner function.

Data-flow parallel Skeletons

The data-flow programming model is a general approach to parallelization based upon data dependencies among a program's operations. The computations is expressed by the data-flow graph, i.e. a DAG whose nodes are instructions and arcs are pure data dependencies[14]. When portions of code are used as graph's nodes, the graph is known as a macro data-flow graph (MDF). The resulting MDF program is therefore represented as a graph whose nodes are computational kernels and arcs are data dependencies.
FastFlow provides an MDF skeleton called *ff_mdf* implementing the macro data-flow parallel pattern. The run-time of the FastFlow mdf pattern is responsible for scheduling fireable instructions (i.e. those with all input data dependencies ready) and managing data dependencies.

2.3 Related work

2.3.1 Autonomic Computing

The aim of Autonomic computing is to address the complexity of technology using technology. The term was derived from the human autonomic nervous system. The autonomic nervous system manages our heart rate and body temperature with no conscious effort. In a similar way self managing autonomic capabilities anticipate IT system requirements and resolve problems with minimal human intervention[7] .
The term Autonomic computing is emblematic of a vast hierarchy of natural self governing systems, many of which consist of multiple interacting, self governing components that in turn comprise a number of interacting , self governing components at the next level [8].

Self Management Properties:

IBM cites four aspects of self management, namely: self-configuration, self-healing, self-optimization, and self protection.

<u>Self Configurations</u>: refers to an automated configuration of components and systems

following high-level policies. The rest of the system adjusts automatically and seamlessly.
Self Healing: refers to a setting where the system automatically detects, diagnoses, and repairs localized software and hardware problems.
Self Optimization: refers to a setting where components and systems continually seek opportunities to improve their own performance and efficiency.
Self-Protection: The system automatically defends against malicious attacks or cascading failures using early warning to anticipate and prevent system wide failures.

Building Blocks of an Autonomic System:

An Autonomic system may consist of building blocks that can be composed together to from self-managing systems. The building blocks include:
- Manageability Endpoints: are the components in the system that expose the state and management operations for a resource in the system. The manageability interface for monitoring and controlling a managed resource is organized into a *sensor* and an *effector*, used to obtain data from the resource and perform operations on the resource respectively.
- Autonomic manager: is an implementations that automates some management functions and externalizes this functions based on the behavior defined by the management interfaces. It implements an intelligent control loop consisting of collecting details from the system (known as monitoring), analyzing the details , planning a change and executing it.
- Knowledge Source: is an implementation of a repository that provides access to knowledge according to the interfaces prescribed by the architecture by the architecture. It consists of particular types of management data with syntax and semantics such as symptoms, policies,requests for change, and change plans. It includes data such as topology information, historical logs, metrics, symptoms, and policies.

Figure 2.3: shows the details of an autonomic manager [7]

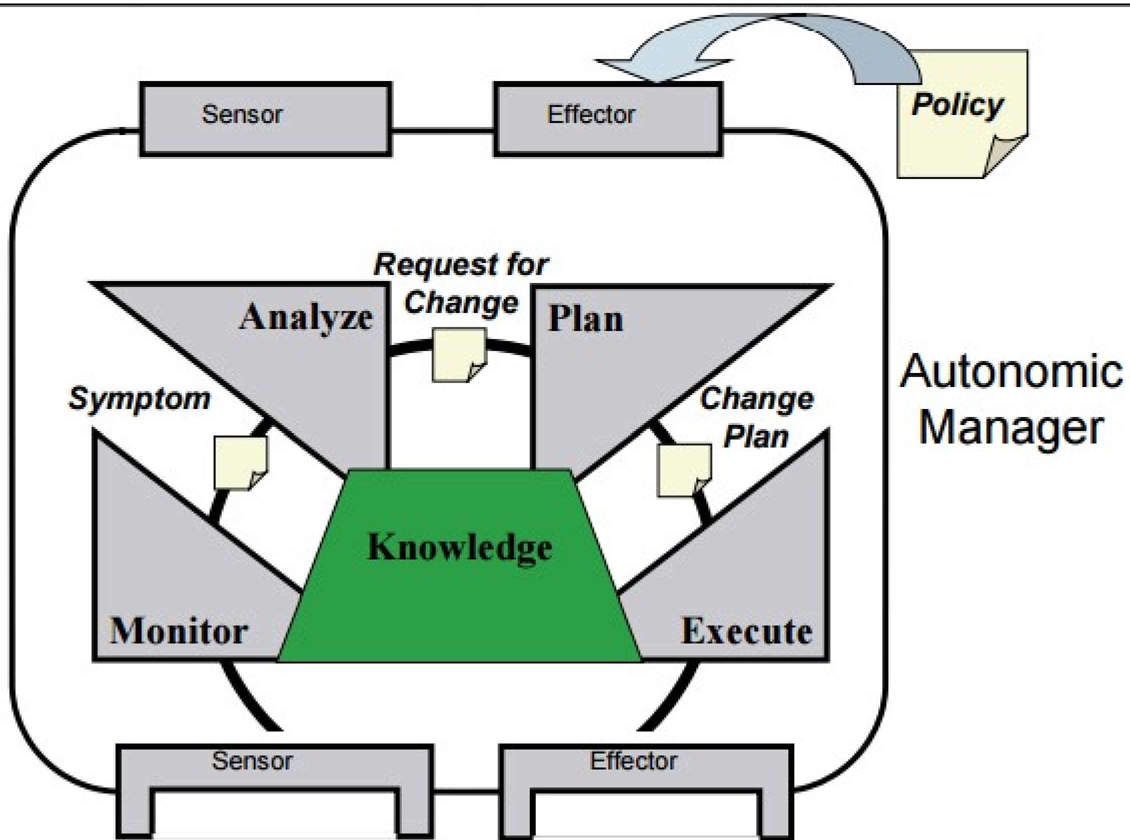


Figure 2.3: Functional details of an autonomic manager with MAPE (Monitor, Analyze, Plan, Execute) loop

The MAPE loop:

As shown in Figure1 an autonomic system executes a loop consisting of four parts known as the MAPE loop, sharing a knowledge source.

- Monitor: during this phase metrics and topologies are collected, filtered and aggregated from the managed resource.

- Analyze: in this phase mechanisms that correlate and model complex situations are provided. These mechanisms allow the AM to learn about the system and predict future situations.

- Plan: In the plan phase mechanisms that construct the actions needed to achieve goals and objectives are provided using policy information.

- Execute : here mechanisms that control the execution plan with considerations for dynamic updates are provided.

2.3.2 Autonomic Management of Non-functional Concerns in Structured Parallel Programming

- In [1] non functional concern management is presented focusing on massively parallel and distributed patterns (focuses on distributed systems and more specifically on grids).
To address the autonomic capability, the ideas from IBM's autonomic computing blue print are adopted. In this case autonomic management of a component is provided by a dedicated autonomic manager which takes care of all activities while interacting with the functional core of the component.

Functional and non-functional application concerns:

- The parallel patterns exploited to implement the application are considered as functional concerns. Where as management issues related to the patterns, such as parallelism degree, load balancing and adaptation of parallelism pattern to different target architectures, are considered as the non-functional concerns.
Usually non-functional features are handled by the user (i.e. the application programmer). But in an ideal programming scenario, functional concerns should be under the user/application programmer control, while non-functional concerns should be handled by the underlying system.

When users/application programmers are in charge of handling the non-functional concerns, the programmer faces several difficulties, mainly:

+ **Code tangling:** usually the user (application programmer) has to mix functional and non-functional code. This makes the programming task difficult (difficult to debug, modularize …) and limits the re-usability of the code.

+**Requires Wide knowledge**: writing non-functional code becomes a burden for the application programmer. This requires wide knowledge of techniques, unrelated to the functional concerns. In addition knowledge of the target architecture, which is only available at run time, might be required.

These problems can be mitigated by moving the non-functional concerns to the run time support (RTS) or the compiler, where the user can provide the non-functional requirements in the form of high level SLA (Service Level Agreements) .

Management of non-functional concerns

Non-functional concerns will be managed by an autonomic-manager (AM), which is an independent activity taking care of all, or some specific non-functional features of the application.

- The AM is characterized by three different dimensions:

i. The concern to be managed

ii. Autonomic policies to be implemented

iii. Degree of cooperation with other managers .

- The AM may concern either single goal or multiple goals. Moreover, the AM being multi-concern or single-concern, it can be a simple AM or hierarchical one. When dealing with more than one goal, the targeted concerns increase. In another dimension, when the AM is hierarchical, it requires more coordination of control. Even when dealing with a single-concern AM, the problem might be difficult, such as mapping of parallel activities to available processors (which is an NP-hard problem). Such a problem gets even more difficult when the AMs are supposed to coordinate with other AMs dealing with different concerns. One way to reduce the complexities is to restrict the kinds of parallel computations, where parallel/distributed computations implement well known parallelism patterns (eg: algorithmic skeletons, Behavioral Skeletons).

Behavioral Skeletons:

A Behavioral skeleton consists of a well known parallelism pattern P along with an AM M_c taking care of a concern on the the computation P. Behavioral Skeletons made it possible to reduce the complexity of performance optimizations into tractable size.

Hierarchical management of a single non-functional concern with BS

When an application is composed of an independent modules, hierarchal management of non-functional concerns can be applied to achieve better results. Inside the hierarchy each software module consists of an AM attached to it. Managers higher in the hierarchy take more autonomous decisions and managers in lower levels of the hierarchy will behave in accordance with the decisions taken in the higher level.

- Two main issues with hierarchal AM

i. A strategy is needed to allow the splitting of a contract C on the top level into subcontracts

$c_1, \dots, c_m$ to be issued by the nested sub-managers.

ii. An AM should be able to act “*passive*” and “*active*” roles. In an “active” mode, the manager automatically tries to issue the contract received (either directly from the user or form a parent manager) by executing the MAPE (Manage, Analise, Plan, Execute) loop .
In “passive” mode, the manager has only to monitor its own execution and execute plans from its parent manager.
A manager enters a passive mode when it can't satisfy the contract at hand, and there are no locally available plans to recover the situation.

Once the two issues are addressed, the user provides the contract as an SLA. The contract is divided into sub-contracts to the children managers in the hierarchy, and the top level manager plays an “active” role.
Each manager in an “active” mode executes the MAPE loop. If an action can't be executed, a contract violation is reported to the corresponding parent and the manager switches into a passive mode, and waits to receive a new contract from it's parent. The problem of “active” and “passive” roles can be solved by organizing the management of non-functional features in two different parts:
- The passive AM implements the mechanisms for “monitoring” its own execution state and executes commands form is parent manager in the hierarchy.
- The active part implements the autonomic policies, in a way the policies maintain the contract received.

Though the strategy of splitting the contracts (the SLA) is a more complex task, it can be achieved by adopting domain specific heuristics associated with a well known parallelism pattern.

In [2] the authors address issues of autonomic management in hierarchical component based distributed systems. A high level view of behavioral skeletons is presented using the ORC notation. In addition a simulation result is is discussed, showing a successful implementation of hierarchical management when service time is autonomically optimized.

Multi-concern management with BS

When considering autonomic managers with multiple goal, more problems arise from structuring of the autonomic management activities. In addition to hierarchal management, there is the issue of how management of different non-functional concerns should be coordinated among the AMs .

In [1] two different scenarios are presented:

1. Single AM taking care of the concerns c1, ... ,cm all together.
2. Multiple hierarchies of autonomic managers, each dealing with a different concern Ci, along with a general super-AM orchestrating the multiple AMs.

In both scenarios the challenge lies in resolving conflicts coming from decisions taken when considering different concerns.

Adopting hierarchical multi-concern AMs is easier due to the complete separation of concerns.

In [3] the authors address the problem of multi-concern autonomic management with independent hierarchical managers and discuss how it can be implemented in a typical use case. The approach presented is based on five steps: coordination of the managers activities, finding a common knowledge by which managers can interact, means of reaching consensus, initialization of of the managers' hierarchy and devising a means of implementing the management.

Coordination: When dealing with multiple autonomic managers taking care of different concerns, there might a case where the decision taken by one autonomic manager is in contradiction with the goal of another autonomic manager taking care of different concern. To resolve such issue there must be a way of *coordination* between the different managers. Two strategies are presented:

1. A super manager AM_0 positioned on top of the hierarchy of managers AM_1-AM_m, coordinating the decisions taken by these managers .
2. The managers AM_1 - AM_m reach agreement with each other before actuating decisions.

Shared knowledge: in order to agree on global application management, a common knowledge is necessary across the different managers. An application graph whose nodes

represent the parallel activities and whose arcs represent communications can be used as a common concept across these managers (the nodes and arcs are labeled with *metadata*[1]).

Reaching Consensus: before committing any decision consensus must be reached on the resulting application graph, i.e. a consensus process must be established and implemented. Consensus can be established as a two phase process: first the proposed reconfiguration is communicated in the from of a new graph *G'* including *metadata* about new resources if any. In the second phase, the manager proposing the change should wait for consensus results. and based on these results, either commits the decision or abort it.

Initialization of the manager hierarchy: The user submits QoS contracts to the AMs ordered according to the priority, such that, the decision of a manager dealing with a contract of higher priority will have a higher precedence. The first contract QoS1 determines which manager is responsible of establishing initial application implementation configuration.

Implementation of autonomic management: The autonomic management can be implemented as a rule-based implementation as in [4], where the rules consist of $preconditions \rightarrow actions$. A classical MAPE rule is implemented by each manager. In the *monitor* phase current values of variables used in the precondition parts are gathered. The *analyze* and *plan* phases correspond to evaluating the preconditions and choosing the corresponding actions respectively, according to the priorities. And the execute phase is implemented as the execution of the actioned planned.

In order to deal with multiple concerns, each rule implemented by AM_i in isolation is transformed into two rules: where the first the rule consists of the proposed action as its precondition and consensus request as its actions, while the second rule consists of the responses from other managers as its precondition and actions consisting of either the original plan or with adjusted plans.

Behavioral Skeletons in GCM

In [4],[5] behavioral skeletons for the Grid Component Model (GCM) are presented. In

1 Metadata of a node may represent mapping information, while metadata of an arc may represent features of corresponding communication channel.

addition, results evaluating the overhead introduced by autonomic management activities are presented in [4].
GCM is a hierarchical component model designed to support component based, autonomic applications in grids. It was defined in the CoreGRID Network of Excellence as an extension of Fractal with grid components. An open source implementation is also available from the EU STREP project GridComp, as an embedded module in the ProActive middle ware.

In the GCM, a component is composed of two major parts. The *membrane*: which is an abstract entity that consists of the control behavior associated with a component.
The *content*, which includes either the code directly implementing functional component behavior, or other composite components.
There are different mechanism for interactions between GCM components including, use-provide ports, stream ports, data ports and event ports. In addition collective interactions are also supported.

Components have two types of interfaces: *functional* and *non-functional*. The non-functional interfaces consist of all the ports needed for a component management activity.
Each GCM component contains an *Autonomic Manager* (AM), interacting with other components' managers through the non-functional interfaces. When the AM executes the classical MAPE loop, the *execution* and *monitoring* are done by a component controller, known as *Autonomic Behavior Controller* (ABC) .
Components consisting of only the ABC are called *passive* components and those with both the ABC and AM are called *active* components.
Behavioral Skeletons
Though programmers are able to write their own AM and ABC in GCM, Behavioral Skeletons abstract those tasks for the user/application programmer.
Behavioral Skeletons as algorithmic represent patterns of parallel computation, along with sound self-management mechanisms,.
On one hand, as an algorithmic skeleton, a behavioral skeleton expose a description of its functional behavior and establishes parametric orchestration schema of inner components.
On the other had it carries constraints that inner components are required with, and encompass a number of pre-defined plans to come with self-management goal.

In [5] the authors introduced simple set of behavioral skeletons modeling f*unctional replication* patterns consisting of a one to many stream server dispatching inputs to workers, number off instances of workers and a many-to-one client stream interface collecting the outputs from the workers. A possible instantiation of this behavioral skeleton can be a task farm or a data parallel .
A behavioral skeleton with *functional replication* pattern (task farm or data parallel) can equipped with a self optimizing policy since the number of workers can be dynamically changed in a sound way. The QoS goal could be to keep a given limit of served requests in a time frame. Then the AM checks whether the average time fulfills the given time limit and react by adding/removing instances of workers.

As the operations implemented by the ABC are more related to the membrane structure, the ABC is implemented as a controller in the membrane. Where as with the AM contractually specified QoS must be enforced. The AM has to decide whether a reconfiguration is needed, an which plan would fulfill the contract. The AM accepts a QOS contract, which consists of set of variables representing the measures to be evaluated, and set of mathematical expressions over these variables. The constraints and goals that the AM should fulfill are encoded in the mathematical expressions. Then the AM checks the validity of the QoS provided and ,if broken, it executes the predefined reconfiguration plans.

LIBERO: a Light Weight Behavioral Skeleton

LIBERO[6] is a prototype behavioral skeleton implemented in Java. It aims to address the problem of multiple Autonomic Managers associated to the same parallel pattern , each taking care of a different non-functional concern. Those managers coordinate themselves in such a way that a global user provided contract can be satisfied.
The implementation of LIBERO relies on the existence of Java and RMI based runtime.

A single behavioral skeleton consists of an algorithmic skeleton (implementing a well known parallelism pattern) along with multiple autonomic managers AM_i, each taking care of a distinct non-functional concern. The algorithmic skeleton also consists of an Autonomic

Controller AC to access its internal state (*monitoring*), and operate on its internal state (*execution*). The AMs periodically execute the classical MAPE control loop.
Coordination among distinct AMs operating on a single algorithmic skeleton is implemented by following a two-phase approach [3].Each action planed by an AM is validated by other AMs in the same behavioral skeleton before being executed. This two-phase approach is realized in such a way where an AM broadcasts its decision to the rest of the AMs, the other managers evaluate the decision to check whether it is in accordance with the concern they are dealing, then finally reply the result of their evaluation. This reply can be an acknowledgement of the decision, an abortion indicating violation of their concern, or a conditional acknowledgement indicating that the plan can executed given a condition is satisfied.
The Autonomic Managers' behavior is expressed in JBoss rules which are compiled and executed by the DROOLS middle-ware library at runtime. The consensus protocol is embedded in the JBoss rules.

Experiments were conducted on farm and pipeline skeletons with autonomic controllers taking care of performance and security, with similar amount of overheads observed in single concern autonomic managers in GCM [4].

Autonomic Management in ASSIST

ASSIST [10] is a parallel and distributed programming environment. It provides the programmer with a structured coordination language that can express parallel programs as an arbitrary graph of software modules, connected by typed data streams. The modules, which can be either sequential (can be written in C,C++, Fortran) or another parallel module (called parmod).
In [9] the ASSIST programming environment is described as a suitable basis to capture all the desired features of QoS control for the Grid.
Components and managers:
A single module or graph of modules can declared as a component. A component consists of

functional and non-functional ports.

Autonomic management is realized in a hierarchal structure.

- Mostly the Autonomic management features in ASSIST focus on performance related issues and load balancing.

Each ASSIST module consists of an application manager (called Module Application Manager: MAM) responsible for configuring and controlling the QoS[2] associated with the module.

Globally, at the component level, the configuration and control of the QoS is implemented by a Component Application Manager (CAM).

Module Application Manager (MAM)

The main task of the Module Application Manager (MAM) is to ensure the contracts provided by the application programmer as QoS. The performance contract can be set up the parent Component Application Manager (CAM).

The ASSIST compiler is responsible for preparing QoS contracts for each parmod and binding them to to the MAM.

The QoS contract consists of a goal and how it should be achieved. Specifically the goal consists of

- Performance features: set of variables evaluated from static module properties, monitored runtime information and performance evaluation.
- Performance model: set of relations between performance features, ranging from simple analytical models to complex models derived though advanced mathematical rechniiques.
- Performance goal: set of inequalities involving performance features.
- Deployment annotations: annotations elaborating the processes resource needs including required hardware, software and other constraints.
- Adaptation policy: refers to a specific adaptation policy among the available ones.

Among the autonomic behaviors of the module application manager, the main one is load balancing of module resources.

The MAPE loop life cycle of the Module application manager looks as follows :

- During the Monitor phase, the VPM's execution times between two consecutive synchronization points are collected. The synchronization points are selected during the

2 QoS is the term referring to the contracts or SLA provided by the application programmer

compile time.
- In the Analyze phase the data collected is used to verify the performance goals. If those goals are violated, possible causes are detected.
- At the Plan phase, if the performance goal is not satisfied, a plan to recover the situation is devised. This consists a sequence of reconfiguration actions, each taking care of a specific cause for the performance degradation. If no effective reconfiguration actions were found to address the problem, an event (i.e. goal violation) is sent to the parent CAM (Component Application Manager).
- At the Execute phase, the VP are redistributed among the VPMs depending on the previous outcome, and resource upgrades are negotiated with the parent CAM(Component Application Manager). The MAM may also receive an event from its parent CAM to apply restructuring due to a global variation of performance degradation.

Component Application Manager (CAM):

Each Component Application manager handles control strategies at a global level for the corresponding component. The CAM might receive requests for restructuring from its MAMs during its *monitor* phase. In such a case it devices a solution by applying global performance model.

In such a case, it individuates a solution at the *analyze* and *plan* phases. It does so by applying a global performance model.

At the *execute* phase, the CAM may receive reconfiguration requests from its corresponding parent.

The root manager (i.e. the *Application Manager AM*) is responsible for final decisions at the global level.

Auto Tuning Parallel Programs

The idea of auto tuning is to automatically adapt the execution of a program to a given software and hardware environment to optimize one or more non-functional objectives such as exection time, energy consumption etc.

Unlike behavioral skeletons, in auto-tuning optimal parameters for specific objectives are predicted at compile time.

In [15] the authors introduce a multi-objective auto-tunning framework consisting of compiler and component featuring a multi-objective optimizer and a runtime system. The multi-objective optimizer derives non-dominated solutions (known as the *preto set*) , each

expressing a trade-off between different conflicting objectives .
The so called pareto set is then made available, in a way where the compiler generates a set of code versions per region, each corresponding to one specific solution. The runtime system then selects a specific solution for each code region based on the context-specific criteria.

In [16] the author presents an interview discussed with Prof. t. Fahringer focusing on the difficulty in predicting the performance of parallel programs, and the popularity of auto-tuning.

Chapter 3

Logical Design

This chapter introduces the logical design of the Autonomic managers implemented in this thesis, for the task farm and pipeline algorithmic skeletons. Section 3.1 discusses the high level view of the behavioral skeletons. Then sections 3.2 and 3.3 discuss the design of the behavioral skeletons for the pipeline and task task farm patterns respectively.

In this chapter and in the rest of the thesis the term Autonomic Controller (AC) is used to refer to the mechanisms inside the skeletons that enable to collect internal execution metrics and execute reconfiguration plans. In addition an Autonomic Manager (at some points simply referred as AM) refers to the part of the component which is responsible to plan reconfigurations based on some kind of policies. In general the term Behavioral Skeleton is used to refer to an Algorithmic skeleton with an Autonomic controller and an Autonomic Manager. In synthesis[1] :

Behavioral Skeleton= Algorithmic Skeleton +Autonomic Controller + Autonomic Manager .

3.1 High level view

The Autonomic manager is designed to work on top of the existing FastFlow algorithmic skeletons . The Farm and Pipeline skeleton patterns are extended, to include Autonomic Controllers. As described in the previous chapter the AC interface is part of the behavioral skeleton . Their main goal is to read internal execution metrics and deliver them to their associated manager. In addition once the delivered metrics are analyzed and reconfiguration plans are prepared by the manager those plans are executed by the AC. Figure 3.1 shows the high level logical design of a behavioral skeleton .

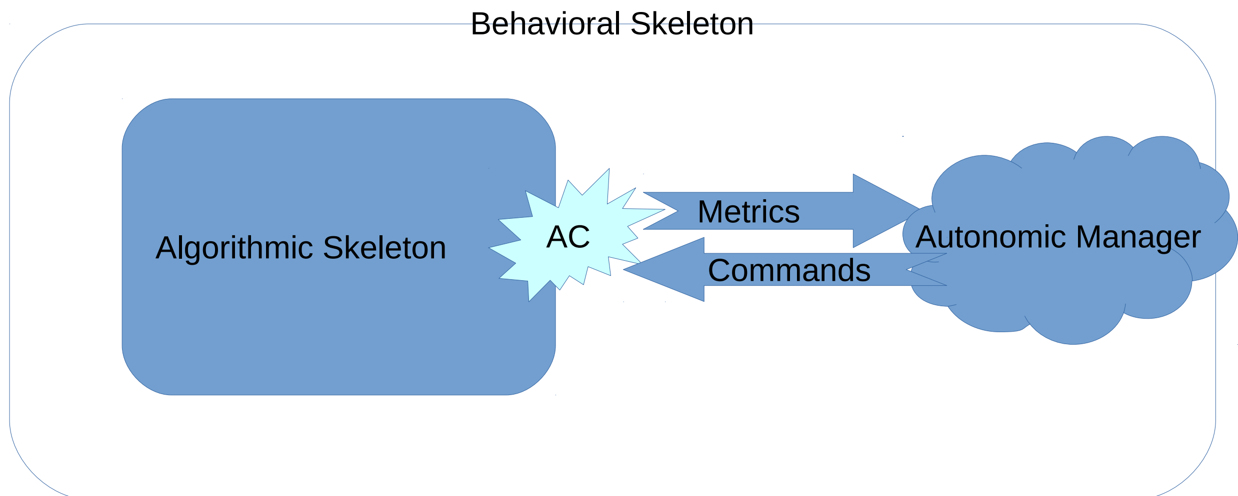


Figure 3.1: high level logical design of a behavioral skeleton

3.1.2 General Overview of the Autonomic Behavior

The existing framework consists of Algorithmic skeletons where parameters (specifically number of stages in pipeline or number of workers in farm) are passed as an argument to the skeletons and they are not changed during the life cycle of the parallel application implemented through the skeletons.

In order to implement the autonomic behavior, the algorithmic skeletons are extended to include an AC which can measure the execution metrics of the skeleton itself and deliver it to the autonomic manager. At the same time it may execute reconfiguration commands planned by the autonomic Manager.

In addition the Behavioral skeleton consists of an Autonomic Manager component which executes the MAPE (Manage, Analyze, Plan, and Execute) loop [8] . During the MAPE loop the AM first receives execution metrics from the AC, then analyzes the metrics and plans reconfigurations based on the analysis of the metrics received plus some simple internal polices and eventually it sends the reconfiguration execution plans to the AC of the algorithmic skeleton.

In order to communicate the execution metrics from the AC to the AM and reconfiguration plans from the AM to the AC communication channels are needed. Logically, two communication channels are needed. One the “Metrics-Channel”, which is used to send the execution metrics from the AC to the AM. Another is the “Command-channel”, which is used to communicate reconfiguration plans from the AM to the AC.

3.2 Logical design of the Task Farm Behavioral Skeleton

A task farm processes a stream of tasks $\{x_0, \dots ,x_m\}$ producing a stream of results $\{f(x_0), \dots ,f(x_m)\}$. The computation of $f(x_i)$ is independent of the computation of of $f(x_j)$ for any $i \neq j$ and the items of the input stream are available at different times[4]. A stream of tasks is absorbed by a server E (known as the emitter) then tasks are sent to workers (one task per worker) is computed)by one instance of worker W) and eventually the result is sent to a collector component C which collects them and deliver to the results to the output stream. To implement the autonomic behavior on the existing task farm skeleton of FastFlow the following components are added:

- The Autonomic controller: as described above this is added as part of the skeleton to collect execution metrics and execute reconfiguration plans. This can be added to the emitter component of the skeleton and to periodically collect the execution metrics as it schedules tasks to the workers .

- The Autonomic Manager: is added as a separate thread . It reads the metrics sent from the AC via the , and decides if reconfigurations are needed. The metrics in this case refer to the service time of the emitter thread and of the workers, while reconfiguration plans can be either addition or removal of a worker based on the metrics if needed. It then communicates the reconfiguration plans to the AC component of the farm through the “Commands Channel”.

Reconfigurations commands in the task farm behavioral skeleton are increasing (activating more) and decreasing (deactivating) of workers. Figure 3.2 shows the design of the task farm behavioral skeleton:

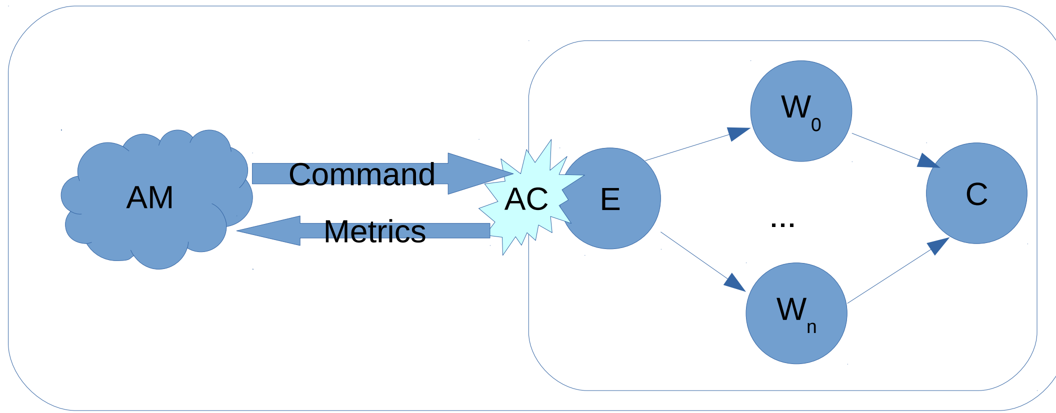


Figure 3.2 Logical Design of the task farm behavioral skeleton

The Autonomic Controller:

The AC for the task farm behavioral skeleton can added in the emitter component of the task farm so that, as the emitter schedules tasks to the workers, it can periodically read the service times of the active workers and its own inter-departure time. Moreover it can execute reconfiguration plans by activating and deactivating workers.

3.3 Logical Design of the Pipeline Behavioral skeleton

The pipeline skeleton, as described in section 2.3.1, is typically used to model computations expressed in stages (usually consisting of two or more stages) [11] .
Given input tasks : $\{x_0, \dots, x_m\}$, the pipeline stages: $S_1, \dots, S_n$ computes
$S_n (\dots S_2 (S_1 (x_m)) \dots), \dots, S_n (\dots S_2 (S_1 (x_0)) \dots)$
- The Autonomic Controller:

Unlike to the task farm, the AC of the pipeline skeleton is added as an additional stage at the end of all the other (functional) stages of the pipeline to collect the metrics, which in this case are the service times of each pipeline stages and send them to the Autonomic Manager. Since the service times of the stages can be measured after each stage has at least executed once the AC can be added as an additional stage in the pipeline, so that it can query the execution time of each stage. It then communicates the metrics to the AM through the "Metrics channel", and execute reconfiguration plans provided by the AM through the "Commands channel", if necessary. The AC is contained inside the pipeline

skeleton itself, thus it can access the mechanisms of the skeleton when collecting metrics and implement reconfiguration plans by merging/splitting stages.

- The Autonomic Manager: is added as a separate thread and, as mentioned above, executes the MAPE loop. It then communicates the reconfiguration plans to the AC component of the pipeline through the "Commands Channel".

Reconfiguration commands in the pipeline behavioral skeleton refer to the merging of consecutive stages and splitting of previously merged stages. The metrics gathered by the AC refer to the service times of the all stages in the pipeline.

The following figure shows the logical design of the pipeline behavioral skeleton:

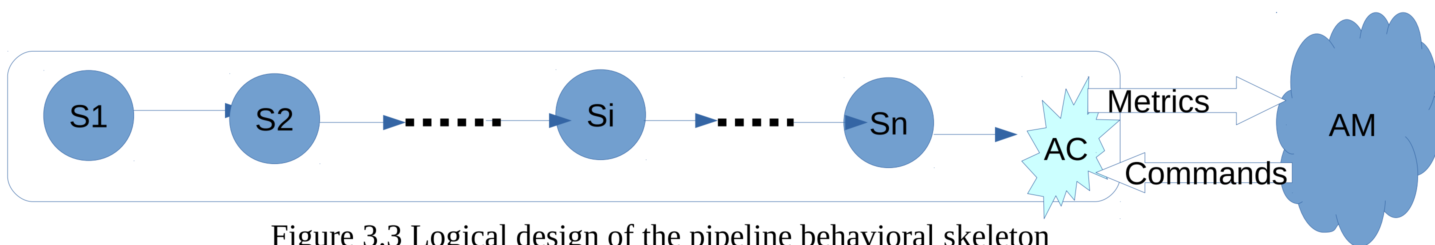


Figure 3.3 Logical design of the pipeline behavioral skeleton

Chapter 4

Implementation

This chapter discusses how the existing algorithmic skeletons in FastFlow were extended into a behavioral skeleton[1] . Section 4.1 discusses how the autonomic manager was implemented and how it plans reconfigurations. In section 4.2 implementation of the autonomic controllers inside the skeletons is discussed, mainly focusing on the monitoring and execution of reconfiguration mechanisms. Finally section 4.3 discusses the autonomic manager policies implemented .

4.1 Autonomic Manager Implementations

As of the logical design discussed in chapter three, the autonomic manager for a skeleton was supposed to be implemented as a separate thread communicating with the skeletons via dedicated communication channels. However, to simplify the implementation and avoid overheads, each skeleton was implemented to contain an instance object of the autonomic manager, so that the communication is simply done via method calls on the contained instance of the autonomic manager object. In other words the autonomic manager was implemented to execute in the same thread with the autonomic controller.

The autonomic manager is implemented as a simple C++ class consisting of *monitor*, *analyze*, *plan* and *execute* methods.

The monitor method receives the metrics collected by the corresponding skeleton and sets the metrics variables so that they can be analyzed. The analyze method of the manager analyzes the metrics received, if necessary. Then the plan method of the manager decides whether reconfigurations are needed, and if so, what type of reconfigurations has to be implemented, based on the policies available to manage the corresponding skeleton. The policies are further discussed later in section 4.3 .

Finally the execute method of the manager sends out the planned reconfiguration

commands to the AC , if there are any, so that the reconfigurations can be executed.

4.2 Autonomic Controller Implementation

Autonomic Controllers are implemented as an extension of FastFlow Farm and Pipeline skeletons adding facilities to query the internal state of the skeletons (monitoring mechanisms) and allow modification of internal states.

The monitoring and execution capablities (modification of internal states) are specific to the type of the underlying skeletons (i.e. Farm or Pipeline skeletons).

4.2.1 Mechanisms provided by lower level FastFlow(Layer 2)

Freezing threads :

FastFlow provides lower level mechanisms to freeze an executing node and thaw previously frozen nodes. In order to freeze an executing node in FastFlow, the `freeze()` method can be used to tell the node that it, should go to sleep (and not terminate) when an "End of Stream (`EOS`)", value is received. Then in order to actually freeze the node the call to the `freeze()` method should be followed by sending an end of stream (`EOS`) value on the node input channel. Alternatively the "`GO_OUT`" value could also be sent to the node to freeze it. While the `EOS` is propagated to other nodes, the `GO_OUT` value is consumed by the node and is not propagated to the output channel.

Thawing a frozen thread:

Once an `ff_node` has been put to sleep by calling the `freeze()` method and sending an end of stream value, it can be thawed back by calling the `thaw()` method, which takes a boolean value indicating whether the node should be frozen or destroyed upon the arrival of an end of steam value.

4.2.2 Monitoring and execution mechanisms in the Task Farm Skeleton

The Farm skeleton skeleton represents functional replication, and it is also

known as master worker. It consists of an Emitter E where incoming tasks are initially processed, a vector of workers W and an optional Collector C where results are aggregated and delivered to the output stream . As stream of incoming tasks arrive they are absorbed by the emitter E and each of the the task is scheduled to one of workers to be computed. The results from the workers my finally be aggregated by a Collector C, in some specific way and sent to the output stream. Alternatively a Farm might exist with out a collector. In this case all the results are consolidated in memory or written to a storage directly by workers producing them.

In FastFlow the farm skeleton consists of a default load balancer to do the task scheduling where tasks can be scheduled either in a round robin or on demand way. The emitter E is also contained in the loadbalancer.

Each time an incoming task arrives to the emitter from input stream or it is generated by the E (eg: reading from disk), the load balancer schedules the task to one of the workers according to the policy of the loadbalancer.

In order to implement Autonomic Controller capabilities in the farm skeleton the default load balancer was extended. The extended load balancer (load balancer with controller) initially starts with only some of the workers activated, and then proceeds by activating or deactivating one worker at a time according to reconfiguration commands from the autonomic manager.

The extended load balancer (load balancer with controller) keeps as a state the average service time of all the workers, the average service time of the emitter (as a simple moving average), initial active workers, and current active_workers. The initial active workers represent the number of workers to activate when the farm starts, if it is less than the total number of workers allocated during the instantiation of the the farm, then the rest of the workers stay frozen. The active_workers state represents how many of the instantiated workers are actively working and how many of them are frozen.

During the monitoring phase, the load balancer queries for the service time of each worker, and computes the average service time of the farm accordingly.

According to [2] the service time of a task farm with n_w workers is given by

$$T_{farm}(n_w)=\sum_{i=1}^{n_w} Tw_i/n_w^2$$

where Tw_i is the service time of worker i .

Since the average service time of a single worker is

$$\sum_{i=1}^{nw} Tw_i/n_w$$

We can simply express the service time as

$$T_{farm}(n_w)=Tw/n_w \qquad (1)$$

Where

$$Tw=\sum_{i=1}^{nw} Tw_i/n_w$$

In addition to the service time of the farm, the average inter-departure time of the emitter is measured as a simple moving average . After the values have been queried, the controller then sends the values to the autonomic manager, so that the manager can analyze them and prepare a reconfiguration plans.

The actuator mechanisms that are required in this case are one that increases the number of active workers and another that decreases the number of active workers.

In order to decrease the total number of workers some of the active workers has to be frozen. Hence, in order to decrease the number of workers in the farm skeleton,first we should ensure that the total number of active workers is more than one, then last active worker is frozen by calling the `freeze()` method, and sending it an end of stream (`EOS`) value. Once this is done the load balancer should wait until that worker consumes the end of stream signal and goes to sleep. Then the number of active workers is set to be one less than

the previous.

On the other hand in order to increase the number of active workers it is simply a matter of activating the last worker frozen previously, which is done by calling the *thaw*() method of the load balancer passing the index of the last worker frozen .

4.2.3 Monitoring and Execution mechanisms in the pipeline skeleton pattern

The pipeline skeleton is composed of sequence of two or more consecutive stages computed one after another, where the input produced by the preceding stage is consumed and processed and the output is delivered as the input to the next stage, and so on.

In the case of a pipeline skeleton, the service time efficiency are optimized by merging and splitting pipeline stages. Specifically, merging stages avoids using extra resources if they do not contribute to improve the service time and splitting previously merged stages to minimize service time (i.e. by using resources if they contribute to improve the service time).

First in order to make the threads (i.e. the nodes) capable of merging with other threads there must be a mechanism to freeze the pipeline only starting from the specific stages that we want to merge. Hence a node that was capable of being frozen with out the need to freeze the entire pipeline had to be implemented. As described in section 4.2.1, in order to deactivate a thread in FastFlow first we must call the *freeze*() method and then send an end of stream value . This means that, if we want to merge a stage somewhere in the pipeline, we must devise a way to send an end of stream to that stage only, without sending it through the input channels. This was implemented by extending the FastFlow node to contain an extra channel in addition to the input and output channels. In this way we can put and end of stream value in the extra channel. In addition if we want the stage to be deactivated as soon as we put the end of stream signal in it, the way it reads its inputs has to be changed. For this reason the *pop*() method which reads tasks from the input stream had to be modified in such a way that it should first check if the extra channel is not empty. If it is not, then the input stream must be read from the extra channel, otherwise it is read from the standard input channel as usual.

Once this is done, another extension of the `ff_node`, which represents an already merged nodes has to be implemented. The merged node consists of references to two nodes each with merging capabilities. The input buffer of the first node becomes the input buffer of the merged node and the output buffer of the of the second node becomes the output buffer of the merged node. In the initialization method i.e. the *svc_init* method, the initialization of the first node is called while in the ending method *svc_end* the ending method of the second node is called. In every FastFlow node the sequential code is provided in the *svc()* method. For the merged node in the *svc* first the sequential code of the first node is called passing the input parameter of the merged node as its input parameter keeping its output in a state variable . Next the sequential code of the second node is called with the output from the first node as its input parameter. This way the two stages which were being executed in a pipeline parallel before getting merged are now executed sequentially. In order to make the merged node capable of being split back in to two stages, getter methods are used to obtain the two merged nodes separately.

In order to implement the Autonomic Controller capabilities, the pipeline skeleton of the FastFlow frame work must also be extended.

The execution mechanisms in the pipeline skeleton with controller are the merging and splitting methods . The merging method takes an index of the pipeline stage as an input parameter and tries to merge it with the next stage. In this case the merging can be accomplished only if the index passed as an input is different from the first stage and the last stage. This is because, FastFlow enables the stream of tasks to be generated from an inner node and if this is the case , the first stage can not be frozen since it will not have an input buffer, which means we cannot send an end of stream signal to freeze it. While for the last stage, it can only be merged with the stage preceding it, and the merge method only assumes a pipeline stage is being merged with the next one.

Merging and splitting of consecutive stages:

In FastFlow the pipeline skeleton stores a reference to each of the stages inside a vector called nodes_list. In order to merge consecutive stages after freezing them, the reference pointing to the first node is replaced by the reference of the merged node, and then the second node is removed from the vector (since its functional code is already contained inside the merged node). Then if the already merged stage needs to be split back to two stages, the merged node is replaced by the first node, and the second node is inserted back

next to the first node. This is done by accessing the internal nodes stored inside the merged node.

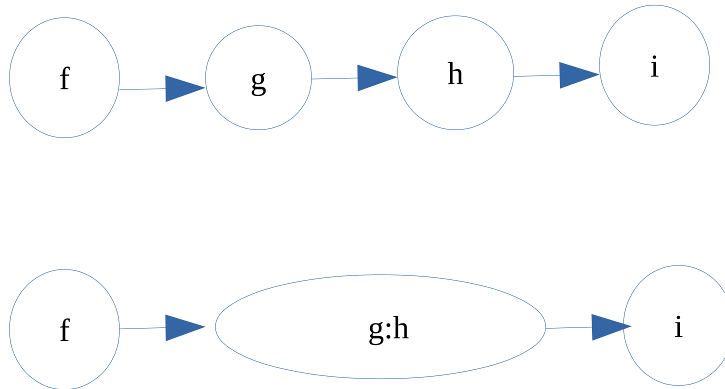


Figure 4.1 pipeline stages before and after merging

4.3 AM Policies

- The Autonomic Manager policies are implemented as *precondition* → *action* rules expressed in plain C++ code. As discussed in chapter 3 each skeleton consists of a corresponding manager, and the policies depend on the type of the manager, that is either a farm manager or a pipeline manager.

4.3.1 Policies for the Farm Behavioral skeleton

Once the metrics indicating the internal state of the Farm skeletons are received from the skeleton , a policy is applied to decide whether to add more (i.e. activate more workers), remove workers or to keep them as they are .

The idea behind the policy of the task farm skeleton is to keep the farm service time approximately equal to the inter-departure time of the emitter. In other words, if the inter-departure time of the emitter is higher than the service time of the task farm, then this means that all the worker in the farm are not utilized, as the utilization factor of the farm is:

$$\rho_{farm} = T_{farm} / T_{p_{emitter}}$$

where T_{farm} is the farm service time and $T_{Pemitter}$ the inter-departure time from the emitter. [12]

Instead if the inter-departure time of the emitter is less than the service time of the farm, the task farm would become a bottleneck.

By balancing the inter-departure of the emitter and service time of the farm, the policy allows to keep optimal number of workers active, even if the behavior of the incoming stream of tasks is dynamic.

Given the farm service time and Emitter inter-departure time, the precondition part of the rule checks if the farm service time is approximately equal to the inter-departure time of the emitter.

The following pseudo-code shows the $precondition \rightarrow action$ rules implemented by the farm Autonomic Manager:

```
if TE > c*Tf then
     action = decreaseWorker
else if c*TE < Tf then
     action = IncreaseWorker
else
     action = do_nothing
```

the constant *c* indicates a parameter that decides the how much difference should be tolerated.

4.3.2 Policies for the Pipeline Behavioral skeleton

The metrics in the pipeline refer to the service times of each pipeline stage. The service time of a pipeline parallel skeleton composed of n stages is given as:

$$T_{pipeline}(s_1, s_2, \ldots s_n) = max(T_{s_1}, T_{s_2}, \ldots, T_{s_n})$$

where T_{si} is the service time of the ith stage

That is, the service time of the slowest stage (which represents the bottleneck of the whole computation) is the service time of the pipeline [11].

Given this the manager tries to balance the stages by merging and splitting the pipeline stages. Initially, the pipeline consists of stages computing a sequential concurrent activities[3] and the possible action is merging consecutive stages if the sum of their service times is not more than the slowest stage, as this would not change the service time as the service time of the pipeline(even after merging consecutive stages, the service time of the pipeline is the service time of the slowest stage).

Later, the behavior of the pipeline might change due to the behavior of the incoming tasks, possibly altering the service times of the stages. In this case if the bottleneck stage becomes the one that was merged previously, that stage has to be split back into two parallel stages, so that the service time of the pipeline could be improved.

The following pseudo code shows the *precondition* → *action* rules implemented by the pipeline skeleton's autonomic manager.

```
BS ← bottleneck_stage
If BS is merged_node then
      split(bottleneck_stage)
end

foreach stage Si in the pipeline
   if servicetime(Si)+servicetime(Si+1) > servicetime(BS) then
          merge(Si,Si+1)
   end
end
```

3 Taking into account that there is no skeleton nesting

Chapter 5

Experiments

This chapter discusses experiments made to validate and asses the prototype implementations discussed in the previous chapter. The goal of the experiments is to verify the functionality of the behavioral skeletons, and to figure out the overheads introduced by the implementation of the autonomic manager.
First the experimental settings are presented in Section 5.1. then Sections 5.2 and 5.3 present the experiments conducted on the task farm and pipeline behavioral skeletons respectively. Then the results are finally summarized in section 5.4.

5.1 Experimental Settings

The experiments were carried out on a 24 core machine with an Ubuntu operating system version 14.04.2 LTS and a kernel version 3.16.0-30-generic. Table 5.1 shows detailed platform on which the experiments were carried.

Processor	CPU Cores	Frequency	L1 Cache	L2 Cache	L3 Cache	Main Memory
AMD 6176 Opteron Processor	24	800 MHz	64 KB	512 KB	5118 KB	32 GB

Table 5.1 Platform used in the experiments

All the test cases have been compiled to native binaries on the platform using g++ version 4.9 . To avoid interference from other users exclusive access to the machine was obtained to conduct the experiments.

All the experimental results for each test case presented are relative to the average of the values obtained in 10 different experiments (runs), in all cases the maximum and minimum (outliers) values were ignored to avoid random errors [15]

5.2 Experiments on the Farm Behavioral Skeleton

Application: Farm Image processing

To test the behavior of the farm behavioral skeleton a simple image processing application was used, where the emitter stage reads image files from the storage, the workers process the images, and the collector writes the processed images back to storage. [14]
In order to come up with tasks whose processing varies over time, the workers execute different processing based on the annotations of the incoming image files. The processing of image files that are annotated takes significantly larger computing than those that are not annotated.

Input Dataset:
The image input datasets have been obtained from the web (from http://www.emt.tugraz.at/~pinz/data/GRAZ_02/) and some of them were duplicated to prepare enough amount of data for the test. Then some of them were properly annotated such

that their processing requires a different amount of time.

Measurements:
For the experiment that verifies the behavior of the task farm skeleton, the program was executed 20 times, among those tests 16 of the tests used a maximum of 15 workers, while the rest used 14,16 and 17 workers. Then the test results that used 15 workers were taken . From the test results, the average of the times where reconfigurations are executed were taken by removing the outliers .

5.2.2 Test cases:

Two types of test cases where prepared:

Test case1:

In the first test case lighter tasks are processed first, followed by the heavier ones, and then followed by lighter tasks. The task farm was started with total of 22 workers, and with two initial active workers.

The following graph shows the results of the first test case:

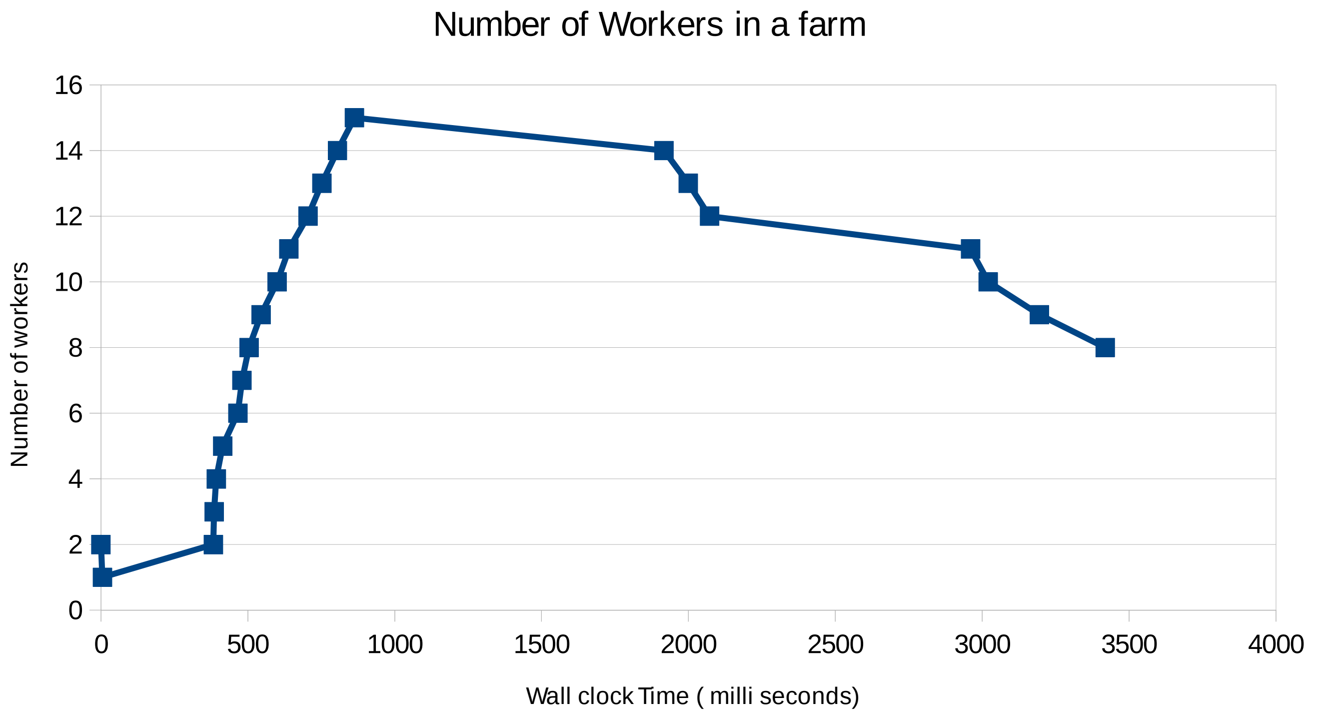


Figure 5.1 Results of test case 1 for the farm behavioral skeleton

The graph shows that the task farm was started with only two active workers. Since the processing time of the lighter tasks takes less processing time than the inter arrival time from the emitter which reads files from disk storage, not more than one worker is required, hence the autonomic manager reduces the number of active workers to one. At the 400th millisecond, the heavier tasks start arriving, and the number of active workers start growing, until the 1st second. Eventually, the lighter tasks start arriving at the 2nd second, and the autonomic manager reacts by deactivating workers. The rate that the number of workers are decreased is different that that of the rate that the number of workers are increased. The main reason comes from the fact that even though the average service time of the workers is reduced by the arrival of lighter tasks, some of the workers have not yet completed processing the heavy tasks, this affects the average service time of the workers.

Test Case 2:

In the second test case, the heavy tasks were generated first, followed by the lighter tasks, and then heavier tasks. Similarly to the first test case, the autonomic manager reacts by increasing the number of active workers when heavy tasks are processed, while decreasing them when lighter tasks arrive.
The graph in Figure 5.2 shows the results of the second test case:

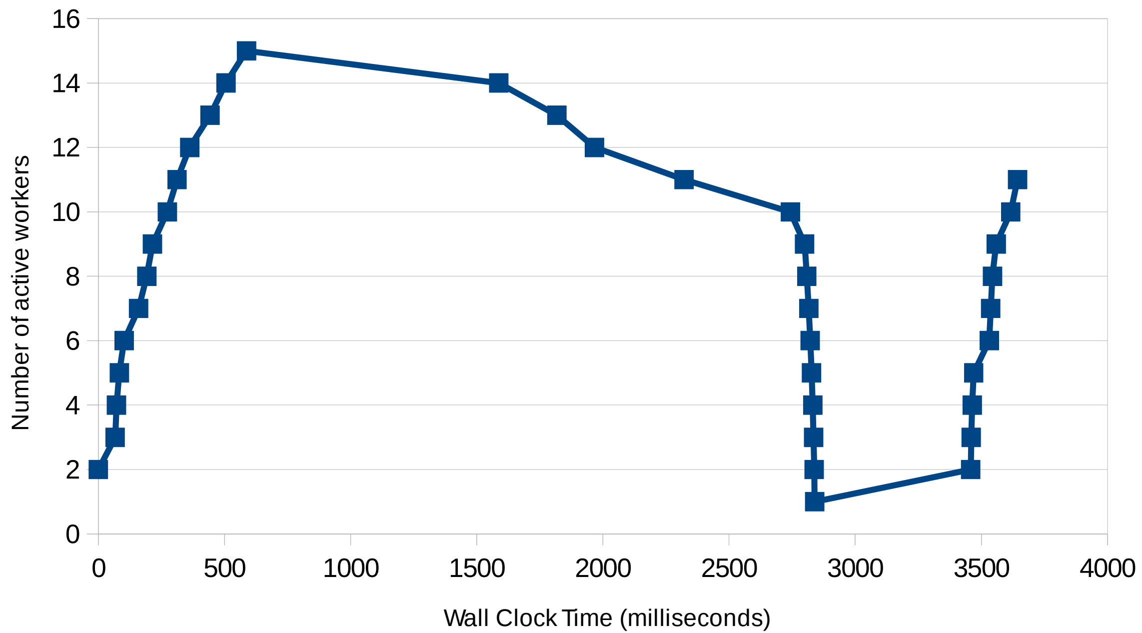


Figure 5.2:Results for the second test case , showing number of active workers during the execution of the test case.

As one can see from the figure above, the task farm was started with only two active workers. the autonomic manager starts activating more workers as the heavy tasks arrive and deactivating workers when lighter tasks arrive. . In this test case, the light tasks are being processed from the 500th millisecond till the 3500th millisecond, but the rate where the number of active workers drops is slow until the 300th millisecond. This is because some of the workers are still processing heavy tasks, affecting the average worker service time.

5.3 Experiments on the Pipeline Behavioral Skeleton

To verify the functionality of the behavioral skeleton a similar image processing application was prepared. The application has five stages. The first stage reads image files from disk, the next three stages do different types of image processing, and the fifth stage writes the processed files back to disk. In a similar way to the experiment for the task farm, the images were annotated and the second stage of the pipeline does two different types of processing based on the annotation. The processing of annotated images in the second stage takes much longer than the processing of unannotated ones . Hence when heavy tasks (i.e. annotated images) arrive in the second stage, its service time becomes higher (even higher than the following two stages), where as when the lighter ones arrive the stage is the service time drops down.

To test the reaction of the autonomic manager, unannotated images were processed first making the service time of the second stage much less than the other stages, and then followed by annotated ones. In this case, the autonomic manager is supposed to react by merging the second stage with the third one. When heavier tasks arrive later, the second stage, which is formed by the merging of the second and third becomes a bottle neck and

hence the autonomic manager is supposed to react by splitting back (i.e. undoing the merge operation performed before) .

Since the first and the last stages work on read/write operations, their service times vary over time due to external factors such as disk locality. To avoid an unexpected behaviors (merging and splitting operations which greatly vary on each test run), the autonomic manager was modified to ignore both the first and the fifth stages of the pipeline.

The graph in Figure 5.3 shows the results obtained for the test case

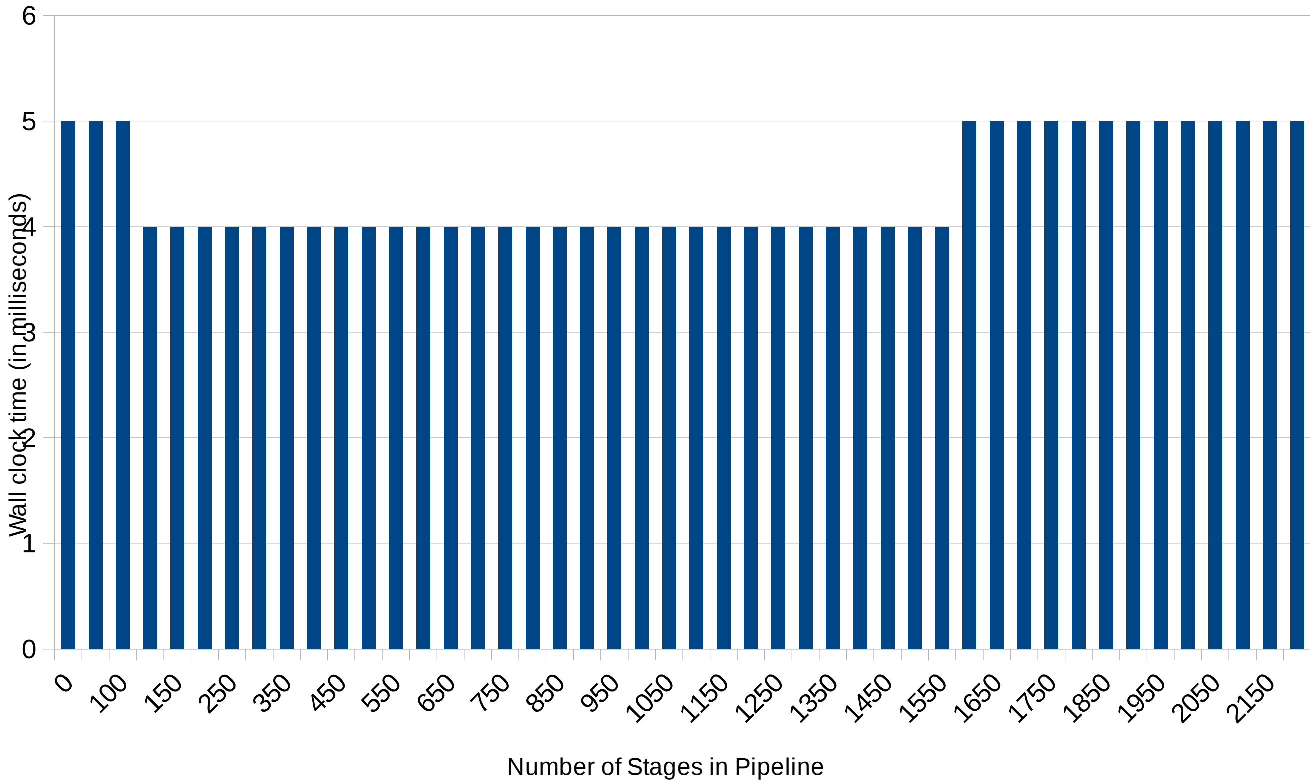


Figure 5.3: number of pipeline stages during the execution of the pipeline behavioral skeleton

As one can observe from the graph in Figure 5.3 the pipeline was started with 5 stages. Then the manager reacts by merging of the second stage with the third one (at the 100th millisecond), as sum of the service times of the second and third stages is less than service time of the (see Figures 5.4 and 5.5). Since the autonomic manager is executed as an additional last stage on the pipeline, a merge reconfiguration is not executed until all the stages process at least one tasks (all the stages has to execute once so that their service times can be measured by the last stage). In addition, to perform the merge operation a freezing signal must be sent to the second stage and the autonomic controller should wait until the freeze signal is consumed by that stage. For those reasons the merging of the stages is not performed until the 100th millisecond. Later when heavy tasks arrive, the second stage, which was formed by merging of the second and third staged is split. This can be seen at the 1650th millisecond of the graph in Figure 5.2.

The following graphs show the service times of the three stages. Figure 5.4 shows the service times of the three stages while processing light tasks, before the second and third stages were merged. Figure 5.5 shows the service times of the stages while processing light tasks after the second and third stages were merged. ,afteri.e when the second stage is processing light (Figure 5.1) , and and later heavy tasks (Figure 5.2)

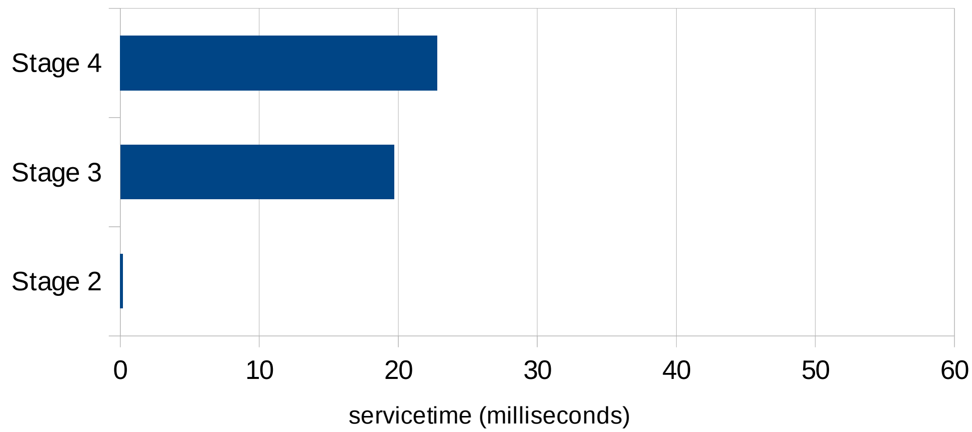


Figure 5.4: service times of the three stages in the pipeline when the second stage is processing lighter tasks (before merging)

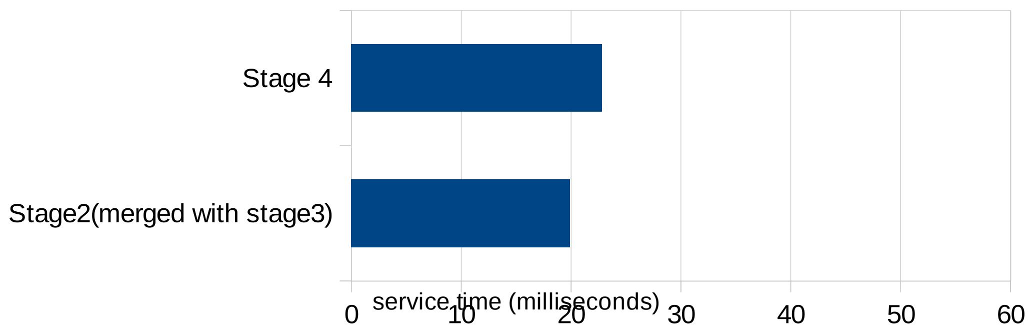


Figure 5.5: Service times of the stages when lighter tasks are processed by the second stage (after the second and third stages are merged)

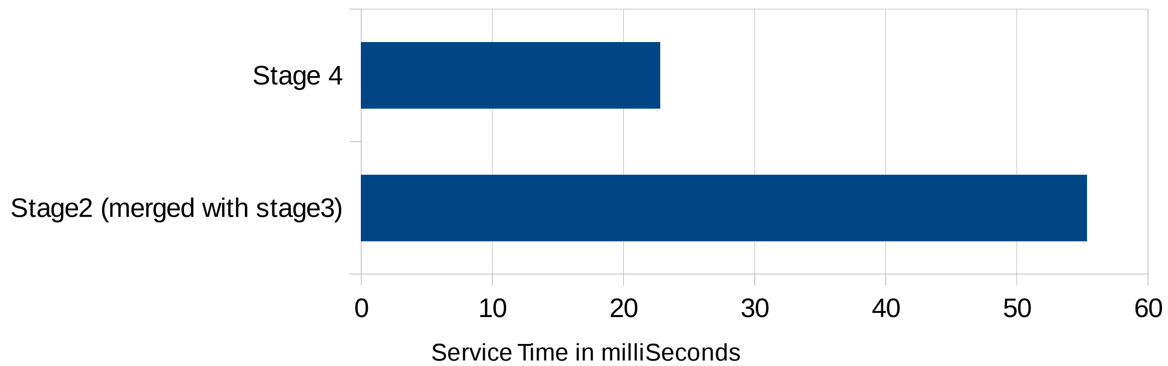


Figure 5.6 Service times of the pipeline stages when the second stage is processing heavy tasks. (stages 2 and 3 merged)

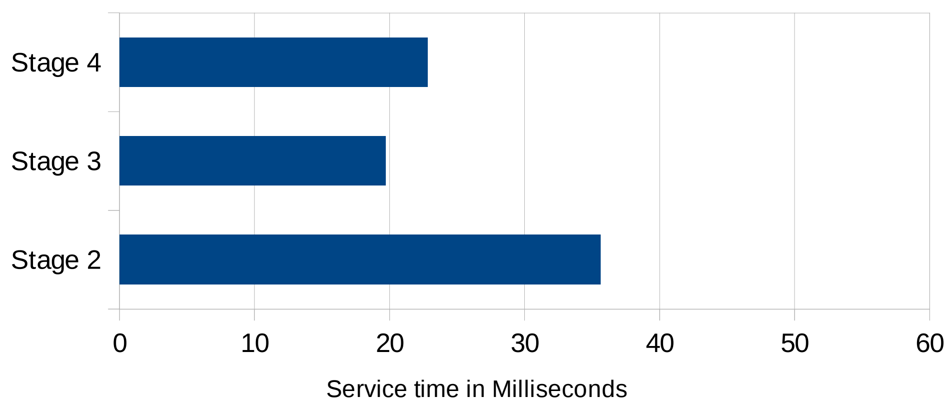


Figure 5.7: service time of the stages when processing annotated images (after stages 2 & 3 has been split).

As shown in the graphs above, the service time of the second stage was very low as compared to the other two stages when processing the lighter tasks in stage two (Figure 5.4. Then the autonomic manager reacts by merging it with the third stage, since the service time of the merged stages is still lower than the service time of the fourth stage, which is the bottleneck of the pipeline at the moment(Figure 5.5). Later when the second stage starts processing heavy tasks, the merged stages become a bottleneck, since the service time of the second stage is now higher (Figure 5.6) the autonomic manager then reacts by splitting the merged node to balance the service times, resulting slightly balanced stages (Figure 5.7).

5.4 Summary

The experiments show that the autonomic manager reacts as expected for both the task farm and pipeline behavioral skeletons when tasks with different behaviors are processed. For the task farm behavioral skeleton, the increase and decrease workers operations are executed in response to the service times of the workers. Similarly for the pipeline merging and splitting of stages is done in response to the service times of the stages and the bottleneck stage.

To measure how fast the Autonomic Controller reacts to do a single reconfiguration the reconfigurations on each test for the farm behavioral skeleton were measured. The measurements were carried out by calling to the “gettimeofday” function inside the reconfiguration methods, specifically, the “gettimeofday” function is called at the beginning and end of the reconfiguration methods and the difference was computed.
d and the result shows that to do a reconfiguration it only takes few microseconds interval (specifically 0.143 milliseconds). While for the pipeline skeleton, it took slightly longer, the average merge/split operations took few milliseconds (specifically 5.156 millisecond on average). This could be due to the implementations of the merge/split operations, which wait for all the stages (except the ones prior to the stage to be merged/split) to freeze after sending a freeze signal.

Chapter 6

Conclusions

This thesis presented the implementation of a prototype behavioral skeleton for stream parallel patterns in the FastFlow algorithmic skeleton framework. Design and implementation of the prototypes was discussed.

For the pipeline skeleton the implementation an autonomic manager that chooses the parallelism degree (i.e. number of pipeline stages) was provided . The autonomic manager works by merging consecutive stages to avoid using extra resources if they do not contribute to the service time, and splitting previously merged stages to improve service time (i.e. by using resources if they contribute to the service time).

For the task farm skeleton an autonomic manager which was able to choose an optimal parallelism degree (by activating and deactivating the number of workers) based the behavior of tasks and execution of the internal components of the skeleton was implemented.

Experiments conducted on the implementations show that the behavioral skeletons were capable to manage performance by choosing the correct parallelism degree for the skeletons at run time. In addition the experiments showed that reconfigurations can be actuated with the introduction of little overheads.

6.1 Future Work

Further work is needed to address some of limitations including:

- One of the major limitations of this thesis is that it doesn't consider the nesting of skeletons. No hierarchy of nesting is assumed during the implementation and the experiments. Those prototypes can be extended (modified to take in to consideration the hierarchical nesting of skeletons)
- The prototype Behavioral Skeletons were implemented on hard coded policies. But a better implementation can be provided in a way where the user can provide some specific contracts which the skeletons must achieve. Then the policies will be set of rules which should take actions when the contracts provided by the user are not fulfilled.

- Only stream parallel skeletons (i.e. farm and pipeline) were covered in this thesis. Other skeletons such as data-parallel skeletons can be extended to include such autonomic behaviors.
- The implementations only focused on skeletons targeting multi-core architectures. The FastFlow skeletal frame work also targets heterogeneous and distributed architectures. Thus behavioral skeletons could be extended so that they can be applied on different architectures.

Bibliography

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick: Autonomic management of non-functional concerns in distributed and parallel application programming, IPDPS, 2009:1-12
[2] M. Aldinucci, M. Danelutto, P. Kilpatrick: Towards Hierarchical Management of Autonomic Components: A case study. PDP 2009:3-10
[3] M. Aldinucci, M. Danelutto, P.Kilpatrick: Autonomic management of multiple non-functional concerns in Behavioral skeletons. CoRR abs/0909.1517 (2009)
[4] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, N. Tonellotto: Behavioral Skeletons in GCM: Autonomic Management of Grid Components . PDP 2008: 54 – 63
[5] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, D. Laforenza, N. Tonellotto: Behavioural Skeletons for Component Autonomic Management on Grids. CoreGRID Workshop – Making Grids Work 2007: 3-15
[6]M. Aldinucci, M. Danelutto, P. Kilpatrick, V. Xhagjika: Libero: A framework for Autonomic Management of Multiple Non-functional Concerns. Euro-Par Workshops 2010: 273- 245
[7] IBM corp. An architectural Blueprint for Autonomic Computing, 2006
[8] J. O Kephart, D. M. Chess. The Vision of Autonomic Computing. IEEE Computer, 36(1): 41-50, 2003
[9] M. Aldinucci, M. Danelluto, M. Vanneschi : Autonomic QoS in ASSIST Grid-aware components. PDP 2006: 221 – 30
[10] Vanneschi, M. (2002). "The programming model of ASSIST, an environment for parallel and distributed portable applications". Parallel Computing 28 (12): 1709–1732.
[11] M. Danelluto “Distributed Systems: Paradigms and Models”. Teaching material: June 2013
[12] M. Vanneschi “High Performance Computing, parallel processing models and architectures” . Pisa University Press 2014
[13] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In Proc. of the 11th Intl. Conference on Compiler Construction (CC), pages 179–196, London, UK, 2002. Springer
[14] M. Torquati : Parallel Programming Using FastFlow, Tutorial on FastFlow programming, September 2015
[15] H Jordan, P Thoman, J Durillo, S Pellegrini, P Gschwandtner, T Fahringer, and H Moritsh: A Multi-objective Auto-tuning Framework for Parallel Codes. International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2012 1-12
[16] M. Torquati, M. Aldinucci, M. Danelutto (2015):The FastFlow website: http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:architecture
[17] M. Torquati (2010). Single-producer/single-consumer queue on shared cache multi-core systems. Technical Report TR-10-20, Universit`a di Pisa, Dipartimento di Informatica, Italy.
[18] Aldinucci, M., Torquati, M., and Meneghin, M. (2009). FastFlow: Efficient paral-lel streaming applications on multi-core. Technical Report TR-09-12, Universit`a di Pisa, Dipartimento di Informatica, Italy.
[19]M Cole : Algorithmic skeletons: A structured approach to the management of parallel computation. PhD Thesis, University of Edinburgh, Computer Science Dpt, Edinburgh 1988.
[20] M Cole: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing, Pitman/MIT Press: London,

1989.
[21] M Leyton: Advanced Features for Algorithmic skeleton Programming. PhD Thesis, University of Nice – Sophia Antipolis, 2008
[22] K Asanovic, R Bodik, J Demmel,T Keaveny,,K Keutzer, J Kubiatowicz, N Morgan, D Patterson, K Sen, J Wawrzynek, D Wessel, and K Yelick,. A view of the parallel computing landscape. Communications of the ACM (2009), 52:56–67.
[23] T Mattson, B Sanders, and Massingill: Patterns for parallel programming. Addison-Wesley professional, 2005

Appendix A

Implementation source Code

```
#ifndef AM_HPP_
#define AM_HPP_

#include <ff/Command.hpp>
#include <iostream>
#include <vector>

namespace ff{

class ff_AM_farm{
public:
        ff_AM_farm():farmCom(DO_NOTHING),workersSvcTime(0.0),
        emitterSvcTime(0.0){}



        void monitor(std::vector<double> svcTimes){
                workersSvcTime = svcTimes[0];
                emitterSvcTime = svcTimes[1];
                svcTimes.clear();

        }


        void analyze(){
                // do nothing in the case of farm
        }



        void plan1(){
                if(emitterSvcTime < workersSvcTime && ((workersSvcTime/emitterSv
cTime) > 3)){
                        farmCom = ADD;
                }

                else if(emitterSvcTime > workersSvcTime && ((emitterSvcTime/work
ersSvcTime) > 3))
                        farmCom = REMOVE;
                else
                        farmCom = DO_NOTHING;

                if(workersSvcTime == 0 || emitterSvcTime == 0) // this happens i
f most of the workers hasn't finished at least one task
                        farmCom = DO_NOTHING;
                #if defined debug_
                if(farmCom == ADD)
                        std::cout << "ADD" << std::endl ;
                else if(farmCom == REMOVE)
                        std::cout << "REMOVE" << std::endl;
                else
                        std::cout << "DO_NOTHING" << std::endl;
                #endif
        }


        farmCommand execute(){
                farmCommand com= farmCom;
                this->farmCom = DO_NOTHING;
                return com;
        }

        farmCommand MAPE(std::vector<double> svcTimes){
                monitor(svcTimes);
                analyze();
                plan1();
                return execute();
        }
private:
        farmCommand farmCom;

        double workersSvcTime;
        double emitterSvcTime;


};

class ff_AM_pipe{
```

```
    public:

            ff_AM_pipe():commandList(*new std::vector<pipeCommand*>()), serviceTimes
    (NULL), maxIndex(0),all_stages_executed(false){

80          }

            void monitor(std::vector<double> * st){
                    this->serviceTimes = st;
                    std::cout << "servicetimes" << std::endl;
85                  for(size_t i=0; i<serviceTimes->size(); ++i)
                            std::cout << (*st)[i] << "\t";
                    std::cout << std::endl;
            }

90



            int analyze(){

95                  int maxIndex = 0;
                    all_stages_executed = true;
                    for(size_t i=0; i < serviceTimes->size(); i++){
                            if((*serviceTimes)[i] > (*serviceTimes)[maxIndex])
                                    maxIndex = i;
100                         if((*serviceTimes)[i]== 0)all_stages_executed =false;

                    }

                    this->maxIndex = maxIndex;
105                 //std::cout << "maxIndex = " << this->maxIndex << std::endl;
                    return maxIndex;
            }


110         void plan(){
                    std::vector<pipeCommand*> cmdList;

                    //try to split the stage with the highest service time
                    commandList.push_back(new pipeCommand(pipeCommand::SPLIT,maxInde
    x+1));
115

                    for(int i=serviceTimes->size()-1; i >1 ; i--){

                            if(((*serviceTimes)[i-1] + (*serviceTimes)[i]) <= (*serv
    iceTimes)[maxIndex]){
120                                 if((*serviceTimes)[i-1]>0 && (*serviceTimes)[i]>
    0){
                                            commandList.push_back(new pipeCommand(pi
    peCommand::MERGE,i)); //to be fixed (make index i rather thatn i-1)
                                            (*serviceTimes)[i-1] +=(*serviceTimes)[i
    ];

                                            return;
125                                 }

                            }



130
                    }




135
            }


            std::vector<pipeCommand*>& execute(){
140                 return commandList;

            }


145         std::vector<pipeCommand*>& MAPE(std::vector<double> * st){
                 monitor(st);
                 analyze();
```

```
                plan();
                return execute();
150         }



    private:
155
            //list of service times of the pipeline stages
            std::vector<double> *serviceTimes;

            //list of commands generated by the autonomic manager
160         std::vector<pipeCommand*> &commandList;

            //the index of the stage with the highest service time (this is the serv
    ice time of the pipeline)
            int maxIndex;
            bool all_stages_executed;
165

    };


170 }


    #endif /* AM_HPP_ */
```

```
/*
 * Command.hpp
 *
 *  Author: nosmas
 */

#ifndef COMMAND_HPP_
#define COMMAND_HPP_

#include <iostream>

namespace ff{


/*class Command{
public:
        virtual void execute() = 0;
        Command(){}
        virtual ~Command(){}
};*/

class pipeCommand{
public:

        enum CommandType{MERGE,SPLIT};

        pipeCommand(CommandType ct, int index): cmdType(ct), index(index){}
        pipeCommand(): cmdType(SPLIT), index(-1){}



        void setCmdType(CommandType cmdType) {
                this->cmdType = cmdType;
        }

        void setIndex(int index) {
                this->index = index;
        }

        CommandType getCmdType() const {
                return cmdType;
        }

        int getIndex() const {
                return index;
        }

        void print(){
                std::cout<< "Command: " ;
                switch(cmdType){
                case MERGE:
                        std::cout << "Merge ";
                        break;
                default :
                        std::cout << "Split";
                }
                std::cout << "index =" << index;
                std::cout << std::endl;
        }

private:
        //ff_pipeWithMerge *pipe;
        CommandType cmdType;
        int index;
};

////////////////////////////////////////////////////////////////////////////////
////////////////////

enum farmCommand{ADD,REMOVE,DO_NOTHING};



}
#endif /* COMMAND_HPP_ */
```

```
/*
 * lb_withControler.hpp
 *
 *
 *      Author: nosmas
 */

#ifndef LB_WITHCONTROLER_HPP_
#define LB_WITHCONTROLER_HPP_

#define FF_BOUNDED_BUFFER true

#include <ff/lb.hpp>
#include <vector>
#include <iostream>
#include <stddef.h>
#include <assert.h>
#include <ff/Command.hpp>
#include <ff/AM.hpp>
#include <ff/SMA.hpp>

namespace ff{


class ff_loadbalancerWithControler:public ff_loadbalancer{
public:

        /*
         * Constructor
         * */
        ff_loadbalancerWithControler(size_t max_num_workers):ff_loadbalancer(max
_num_workers),reconf_times(new std::vector<double>)     {


                nextWorker = 0;
                initial_active_workers=2;
                first=true;
                activeWorkers=ff_loadbalancer::getNWorkers();
                inter_departure_time = new SMA(4);
                autonomicManager =new ff_AM_farm();

        }
        /*
         * Schedule Task: Executes the plan (increase/decrease workers) after sc
heduling the task
         * */
        virtual bool schedule_task(void * task, unsigned int retry=(unsigned)-1,
 unsigned int ticks=0) {


                if(!ff_loadbalancer::schedule_task(task,retry,ticks))
                        return false;

                        gettimeofday(&now,NULL);
                                if(!first){
                                        //std::cout<< diffmsec(now ,prev_call_to
_sched) << ")))))" <<std::endl;

                                        inter_departure_time->add(diffmsec(now ,
prev_call_to_sched));
                                }else{
                                        first =false;
                                }
                                prev_call_to_sched = now;

                nextWorker= (nextWorker+1) % activeWorkers ;
                if(nextWorker == 0){
                        farmCommand com =autonomicManager->MAPE(monitor());
                        execute(com);
                }

                return true;

        }

```

```
        std::vector<double> monitor(){
75              double workersSvcTime=0, emitterSvcTime=0;
                 std::vector<double> result;
                                for(size_t i=0; i<activeWorkers; i++){
                                        workersSvcTime += (workers[i]->svcffTime
()/activeWorkers);
                                }
80                              workersSvcTime /= activeWorkers;
                                emitterSvcTime = inter_departure_time->avg();
                                result.push_back(workersSvcTime);
                                result.push_back(emitterSvcTime);
#if defined debug__
85                              std::cout << " emitter service time = " << emitterSvcTim
e << std::endl;
                                std::cout << "workers service time = " << workersSvcTim
e << std::endl;
#endif
                                                return result;
        }
90



        /*
         * setter method for initial workers
95       * @param i: the number of initial workers in the farm
         *
         * */
        void setInitailWorkers(size_t i){
                initial_active_workers=i;
100     }

        /*
         * SVC method:
         * starts with initial_workers
105      * */

        virtual int svc_init(){
                activeWorkers=ff_loadbalancer::getNWorkers();
                //deactivate all the workers
110             stop_all();
                // then activate only some
                ff_loadbalancer::thawWorkers(true, initial_active_workers);
                activeWorkers=ff_loadbalancer::getnworkers();
                gettimeofday(&startTime,NULL);
115             return ff_loadbalancer::svc_init();
        }


        virtual void* svc(void* task){
120
                return ff_loadbalancer::svc(task);
        }


125     void stop_all(){
                size_t nw = ff_loadbalancer::getnworkers();

                for(size_t i=0; i<nw ; ++i){
                        ff_loadbalancer::freeze(i);
130                     ff_loadbalancer::ff_send_out_to(EOS,(int)i);
                }


                activeWorkers=ff_loadbalancer::getNWorkers();
135     }



        /*
         * simply increases or decreases the number of workers according to the
command received
140      * */
        bool execute(farmCommand farmCom){


                if(farmCom == ADD){
145
                        return increaseWorker();
                }
```

```
                else if(farmCom == REMOVE){
                        return decreaseWorker();
150             }


                return true;
        }
155
        /*
         * decrease number of workers
         */
        bool decreaseWorker(){
160             timeval reconf_start,reconf_end;

                activeWorkers=getnworkers();

                if(activeWorkers <= 1)
165
                        return false;
                gettimeofday(&reconf_start,NULL);

                ff_loadbalancer::freeze(activeWorkers-1);
170             ff_loadbalancer::ff_send_out_to(EOS,activeWorkers-1);

                ff_loadbalancer::wait_freezing(activeWorkers-1);
#if defined debug_
                        std::cout << " Deactivating worker " << activeWorkers-1 << std
::endl;
175 #endif
                --activeWorkers;

                gettimeofday(&reconf_end,NULL);
                reconf_times->push_back(diffmsec(reconf_end,reconf_start));
180             gettimeofday(&now,NULL);
                std::cout << diffmsec(now,startTime) << "\t" << activeWorkers <<
std::endl;

                return true;
        }
185
        /*
         * decrease the number of workers
         */
        bool increaseWorker(){
190
                timeval reconf_start,reconf_end;
                activeWorkers=getnworkers();
                gettimeofday(&reconf_start,NULL);
                if(activeWorkers >= ff_loadbalancer::getNWorkers())
195                     return false;
#if defined debug_
                std::cout << "activating Worker " << activeWorkers << std::endl;
#endif
                ff_loadbalancer::thaw(activeWorkers, true);
200             activeWorkers++;
                gettimeofday(&reconf_end,NULL);
                reconf_times->push_back(diffmsec(reconf_end,reconf_start));
                gettimeofday(&now,NULL);
                std::cout << diffmsec(now,startTime) << "\t" << activeWorkers <<
std::endl;
205
                return true;
        }

        /*
210      * compute the average reconf time
         */
        double average_reconf_time(){
                double sum = 0.0;

215             for(unsigned int i = 0; i< reconf_times->size(); ++i){
                        sum += (*reconf_times)[i];
                }

                if(reconf_times->size() == 0)
220                     return -1.0;
                return sum/reconf_times->size();
```

```
        }

225 private:
        size_t activeWorkers;
        bool first;
        size_t initial_active_workers;
        ff_AM_farm* autonomicManager;
230     size_t nextWorker;

        SMA* inter_departure_time;
        struct timeval prev_call_to_sched,now;
        struct timeval startTime;
235     std::vector<double>* reconf_times;
    };



240



    }

245




250 #endif /* LB_WITHCONTROLER_HPP_ */
```

```
/*
 * managed_farm.hpp
 *
 *      Author: nosmas
 */

#ifndef MANAGED_FARM_HPP_
#define MANAGED_FARM_HPP_
#include <ff/farm.hpp>
#include <ff/lb_withControler.hpp>

namespace ff{

class ff_managed_farm: public ff_farm<ff_loadbalancerWithControler> {
public:
        ff_managed_farm(size_t init_workers = 2, int buffSize=5):ff_farm<ff_load
balancerWithControler>(),initial_activeWorkers(init_workers){

        lb->setInitailWorkers(initial_activeWorkers);

        }

double getReconf_time(){
        return lb->average_reconf_time();
}

private:
    size_t initial_activeWorkers;


};

}

#endif /* MANAGED_FARM_HPP_ */
```

```
/*
 * pipe_withMerge.hpp
 *
 *  Created on: Mar 1, 2015
 *      Author: nosmas
 */

#include <ff/pipeline.hpp>
#include <iostream>
#include <vector>
#include <ff/Command.hpp>
#include <ff/AM.hpp>



namespace ff {




///////////////////////////////////////////////////////////////////////////////
////////////////////////////////////



/*
 *class ff_mnode:
 *represents a node which can be merged
 *
 * */

class ff_mnode:public ff_node {
protected:
        /*
         * constructor
         * */
        ff_mnode():ff_node(){
                eosRecieved=false;
                extra_buffer=new FFBUFFER(1);
                extra_buffer->init();
                splitted =false;
        }

public:

        friend class ff_mergedNode;

        /*
         *
         * */
        void eosnotify(int id=-1) {
                eosRecieved = true;
                //std::cout << " EOS Recieved in stage " << get_my_id() << std::
endl;
        }




        /*
         * checks if there are entries in the extra buffer before popping from t
he input
         * buffer
         */
        inline bool pop(void** ptr){

                if(!extra_buffer->empty()){
                        return extra_buffer->pop(ptr);
                }
                else

                        return ff_node::pop(ptr);
        }



        bool splitted;
        /*
         * put an EOS in the extra buffer so that the node can be frozen
```

```
         * */
        virtual bool signal_freeze(){
                if(get_in_buffer() == NULL)
                        return false;
                freeze();
                return extra_buffer->push(EOS);
        }


        /**
         * \brief Gets extra buffer
         *
         * It returns a pointer to the extra buffer.
         *
         * \return A pointer to the extra buffer
         */
        FFBUFFER* getExtraBuffer(){
                return extra_buffer;
        }

        bool isEosRecieved() const {
                return eosRecieved;
        }

private:
        FFBUFFER *extra_buffer;
        bool eosRecieved;

};

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////

/*
 * class ff_mergedNode:
 * Represents a node consisting of two merged stages
 * */
class ff_mergedNode: public ff_mnode {
public:
        ff_mergedNode(ff_mnode * node_1, ff_node * node_2):ff_mnode(),svc_time(0
){

                node1=node_1;
                node2=node_2;

                if(node2->get_out_buffer() != NULL)
                        //set the output buffer of node2 as the output of the ne
w merged node
                        if(ff_node::set_output_buffer(node2->get_out_buffer()) !
=0){
                                error("ERROR: setting the output of the merged node!");
                                return;
                        }

                if(node1->get_in_buffer() != 0)
                        //set the input buffer of node1 as the input of the new
merged node
                        if(ff_node::set_input_buffer(node1->get_in_buffer()) !=0
){
                                error("ERROR: setting the input of the merged node!");
                                return;
                        }

                ff_node::set_id(node1->get_my_id());

        }

        void eosnotify(int id=-1) {
                        eosRecieved = true;

                }



        inline bool pop(void** ptr){
                //return ff_mnode::pop(ptr);
                if(!extra_buffer->empty()){
                        return extra_buffer->pop(ptr);
```

```
                                }
                        else
150                             return ff_node::pop(ptr);
                }

                int svc_init(){
                        return node1->svc_init();
155                     return 0;
                }

                void svc_end(){
                        node2->svc_end();
160             }


                void * svc(void* task){
                        gettimeofday(&start,NULL);
165                     void *t=node1->svc(task);
                        t= node2->svc(t);
                        gettimeofday(&end,NULL);
                        svc_time = diffmsec(end,start);
                        return t;
170             }




175             virtual double svcffTime(){
                        return svc_time;
                }

                virtual bool signal_freeze(){
180                             if(get_in_buffer() == NULL)
                                        return false;
                                freeze();
                                return extra_buffer->push(EOS);
                        }
185
                ff_mnode * getNode1(){ return node1; }
                ff_node * getNode2(){ return node2;     }

        private:
190             ff_mnode *node1;
                ff_node *node2;
                struct timeval start,end;
                double svc_time;

195     };
        ////////////////////////////////////////////////////////////////////////////////
        ///////////////////////////////

        /*
         * class ff_pipeWithMerge:
200      * a pipe line whose stages can be merged and splitted back
         * */
        class ff_pipeWithMerge: public ff_pipeline {
        public:
                /*
205              * simple constructor
                 * */
                ff_pipeWithMerge():ff_pipeline(){

                        manager=new ff_pipelineManager(this);
210                     first=true;
                        manager_added = false;
                }


215             int run_then_freeze(){

                        if (isfrozen()) {
                                thaw(true);
                                return 0;
220                     }
                        if(!manager_added) add_manager();
                        if(!prepared) if(prepare() < 0) return -1;

                        freeze();
```

```
225
                if (!barrier)  barrier = new BARRIER_T;
                const int nthreads = cardinality(barrier);

                if (nthreads+1 > MAX_NUM_THREADS) {
230                     error("PIPE_WITH_MERGE, too many threads, increase MAX_NUM_THRE
ADS !\n");
                        return -1;
                }
                barrier->barrierSetup(nthreads);

235             int startId = (get_my_id() > 0)? get_my_id():0;


                for(unsigned int i=0; i < nodes_list.size(); ++i){
                        nodes_list[i]->set_id(i+startId);
240                     if(nodes_list[i]->freeze_and_run(true) < 0){
                                error("ERROR: PIPE_WITH_MERGE, (freezing and) running stag
e %d\n", i);
                                return -1;
                        }

245             }

                return 0;
        }

250

        /*
         * merge or split stages according to the commands
         * */
255     bool execute(std::vector<pipeCommand*>&commandList){
                while(!commandList.empty()){
                        pipeCommand *cmd = commandList.front();

                        if(cmd->getCmdType() == pipeCommand::MERGE){
260
                                merge(cmd->getIndex());


                        }else if(cmd->getCmdType() == pipeCommand::SPLIT){
265                             split(cmd->getIndex());
                        }
                        commandList.erase(commandList.begin());

                }
270

                return true;
        }

275

        std::vector <double>* monitor(){
                std::vector <double>* serviceTimes = new std::vector<double>() ;
                serviceTimes->reserve(nodes_list.size());
280

                for(unsigned int i=1; i < nodes_list.size()-2; ++i){

                        serviceTimes->push_back(nodes_list[i]->svcffTime());
285             }

                return serviceTimes;
        }

290
        /*
         * Merges a pipeline stage with the next one, given the index of the fir
st stage to be merged
         *@param index is the id  first one to be merged with its successor
         * */
295     int merge(unsigned int index){
                struct timeval mStart;
                struct timeval mEnd;
                double mergeTime =0;
```

```
300
                    gettimeofday(&now,NULL);
                    mergeTime = diffmsec(now,start_time);

                    gettimeofday(&mStart,NULL);
305
                    //the last stage (nodes_list[size -1]) is the manager, nodes_lis
t[size-2] is the last stage, and both can't be merged
                    if(index >= (nodes_list.size() - 2) || index == 0) return -1;


310                 ff_mnode *firstNode=dynamic_cast<ff_mnode*>(ff_pipeline::nodes_l
ist[index]); // does dynamic_cast cause inefficiency?


                    if(firstNode == NULL) {
                            error("ERROR: PIPE_WITH_MERGE, nodes of a pipeline must be of type ff_
mnode");
315                         return -1;
                    }

                    if(!firstNode->signal_freeze()){
                            error("ERROR: PIPE_WITH_MERGE, Signaling freeze to the first node");
320                 }
                    firstNode->wait_freezing();

                    nodes_list[index]=new ff_mergedNode(firstNode,nodes_list[index+1
]);

325                 nodes_list.erase(nodes_list.begin() + (index + 1));

                    nodes_list[index]->freeze_and_run(true);

                    thaw(true);
330
                    gettimeofday(&mEnd,NULL);
                    //std::cout << "merged stage " << index << " with with *__  " <<
 index+1 << " at " << mergeTime << std::endl;

                    return 0;
335         }




340         bool split(unsigned int index){
                    struct timeval sStart,sEnd;
                    double split_time ;
                    gettimeofday(&sStart,NULL);

345                 if(index < 1)
                            return false;

                    ff_mergedNode *casted_val=dynamic_cast<ff_mergedNode*>(ff_pipeli
ne::nodes_list[index]);

350                 if(!casted_val)
                            return false;

                    gettimeofday(&now,NULL);
                    split_time = diffmsec(now,start_time);
355
                    //std::cout << "\n \n split time = " << split_time <<std::endl;

                    ff_mnode * node1 = casted_val->getNode1();

360                 ff_node * node2 = casted_val->getNode2();

                    if(!casted_val->signal_freeze()){
                                            error("ERROR: PIPE_WITH_MERGE, Signaling freez
e to the merged node");
                                    }
365

                    casted_val->wait_freezing();

                    node1->splitted =true;
370                 nodes_list[index] = node1;//casted_val->getNode1();
```

```
                nodes_list.insert(nodes_list.begin()+(index+1),node2);

                thaw(true);
                gettimeofday(&sEnd,NULL);
375             double elapsed_time = diffmsec(sEnd,sStart);


                return true;
        }
380

private:
        int add_manager(){

385             if(!manager_added){
                        gettimeofday(&start_time,NULL);
                        manager_added=true;
                        return add_stage(manager);
                }
390
                return -1;
        }


395     class ff_pipelineManager:public ff_mnode{
        public:
                ff_pipelineManager(ff_pipeWithMerge *p):ff_mnode(){
                        pipe=p;
                        autonomicManager = new ff_AM_pipe();
400                     currentPhase = MONITOR;
                }

                void * svc(void * task){

405                     std::vector<pipeCommand*>commandList =autonomicManager->
MAPE(pipe->monitor());
                        pipe->execute(commandList);



410                     return task;
                }

                enum phase{MONITOR,ANALYZE,PLAN,EXECUTE};

415
        private:
                ff_pipeWithMerge* pipe;
                phase currentPhase;
                ff_AM_pipe *autonomicManager;
420
        };
        /*end of inner class */



425
        private:
        ff_pipelineManager* manager;
        bool first;
        bool manager_added;
430     struct timeval start_time, now;


};

435


}
```

```
/*
 * ImageFarm.cpp
 *
 *
 */
#include <cassert>
#include <iostream>
#include <string>
#include <algorithm>
#include <ctime>
#include <vector>
#include <sstream>
#include <fstream>


#include <Magick++.h>

#include <ff/pipeline.hpp>
#include <ff/farm.hpp>
#include <ff/managed_farm.hpp>

using namespace Magick;

using namespace ff;


                struct Task {
                    Task(Image *image, const std::string &name, double r=1.0,dou
ble s=0.5):
                        image(image),name(name),radius(r),sigma(s) {};

                    Image             *image;
                    std::string  name;
                    const double       radius;
                    const double       sigma;
                   };

                char* getOption(char **begin, char **end, const std::string &opt
ion) {
                    char **itr = std::find(begin, end, option);
                    if (itr != end && ++itr != end) return *itr;
                    return NULL;
                }



                class Read:public ff_node {
                public:
                        Read(char **images, const long num_images, double r, dou
ble s):
                                images((const char**)images),num_images(num_imag
es),radius(r),sigma(s) {}

                            void *svc(void *) {
                                for(long i=0; i<num_images; ++i) {
                                    const std::string &filepath(images[i]);
                                    std::string filename;
                                    // get only the filename
                                    int n=filepath.find_last_of("/");
                                    if (n>0) filename = filepath.substr(n+1);
                                    else     filename = filepath;



                                    Image *img = new Image;
                                    img->read(filepath);
                                    Task *t = new Task(img, filename,radius,sigm
a);

                                    ff_send_out(t);
                                }
                                return EOS;
                            }

                private:
                        const char **images;
                        const long num_images;
                        const double radius;
                        const double sigma;
```

```
                };
75

                class BlurEmboss:public ff_node{
                public:

80                      void* svc(void* tsk){
                                Task* in=(Task*)tsk;
                                if(in->name.find("Emboss") != std::string::npos)
{
                                        in->image->blur(in->radius, in->sigma);
                                        in->image->emboss(in->radius, in->sigma)
;
85                                      in->image->enhance();
                                }else{

                                        in->image->comment(in->name);;

90



                                                                return in;

95                      }


                };

100


                class Writer: public ff_node{
                public:
                        void* svc(void*tsk){
105                             Task* in=(Task*)tsk;
                                 std::string outfile = "./out/" + in->name;
                                    in->image->write(outfile);
                                    std::cout << "image " << in->name << " has been
written to disk\n";
                                    delete in->image;
110                                 delete in;

                                return GO_ON;
                        }
                private:
115
                };

                int main(int argc, char *argv[]) {

120                     if (argc < 2) {
                                std::cerr << "use: " << argv[0] <<
                                    " [-r radius=1.0] [-s sigma=.5] [ -n Wrks=2] <image-file> [ima
ge-file]\n";
                                return -1;
                            }
125                         double radius=1.0,sigma=0.5;
                            int Wrks = 2;
                            int start = 1;
                            char *r = getOption(argv, argv+argc, "-r");
                            char *s = getOption(argv, argv+argc, "-s");
130                         char *n = getOption(argv, argv+argc, "-n");
                            if (r) { radius    = atof(r); start+=2; argc-=2; }
                            if (s) { sigma     = atof(s); start+=2; argc-=2; }
                            if (n) { Wrks      = atoi(n); start+=2; argc-=2; }

135                         std::freopen("./out/output.tsv", "a", stdout);

                            InitializeMagick(*argv);

                            long num_images = argc-1;
140                         assert(num_images >= 1);

                            std::vector<ff_node*> workers;

                            for(int i=0; i < Wrks;i++)
145                             workers.push_back(new BlurEmboss);
                            Read read(&argv[start], num_images, radius, sigma);
                            Writer writer;
```

```
                              ff_managed_farm farm;
150

                              farm.add_emitter(&read);
                              farm.add_workers(workers);
                              farm.add_collector(&writer);
155
                              if(farm.run_then_freeze()) {
                                  error("running farm \n");
                                  return -1;
                              }
160                           farm.wait_freezing();
                              std::cout <<"\n\n\n\n"<<std::endl;
                              std::cerr << "average reconfiguration time = " << farm.getRec
   onf_time() <<std::endl;

                          return 0;
165                               }
```

```
/*
 * ImagePipe.cpp
 *
 */
#include <cassert>
#include <iostream>
#include <string>
#include <algorithm>
#include <ctime>


#include <Magick++.h>

//#include <ff/pipeline.hpp>
#include <ff/pipe_withMerge.hpp>

using namespace Magick;

using namespace ff;


                struct Task {
                    Task(Image *image,  std::string &name, double r=1.0,double s
=0.5):
                        image(image),name(name),radius(r),sigma(s) {};

                    Image             *image;
                    std::string  name;
                    const double       radius;
                    const double       sigma;

                };

                char* getOption(char **begin, char **end, const std::string &opt
ion) {
                    char **itr = std::find(begin, end, option);
                    if (itr != end && ++itr != end) return *itr;
                    return NULL;
                }



                class Read:public ff_mnode {
                public:
                        Read(char **images, const long num_images, double r, dou
ble s):
                                images((const char**)images),num_images(num_imag
es),radius(r),sigma(s),count(0) {}

                            void *svc(void *tsk) {

                                if(count < num_images){
                                         const std::string &filepath(images[coun
t]);
                                         count++;
                                                                             std:
:string filename;

                                                                             // g
et only the filename
                                                                             int
n=filepath.find_last_of("/");
                                                                             if (
n>0) filename = filepath.substr(n+1);
                                                                             else
     filename = filepath;


                                                                             Imag
e *img = new Image;

                                                                             img-
>read(filepath);
                                                                             Task
 *t = new Task(img, filename,radius,sigma);

                                                                             retu
rn t;
```

```
65                          }



                            return EOS;
70                      }

            private:
                    const char **images;
                    const long num_images;
75                  const double radius;
                    const double sigma;
                    int count;
            };

80

            class Enhance: public ff_mnode{
            public:

                    void* svc(void* tsk){
85
                            Task* in = (Task*)tsk;

                                    in->image->enhance();

90                          return in;
                    }
            };


95          class Blur:public ff_mnode{
            public:

                    void* svc(void* tsk){
                            Task* in=(Task*)tsk;
100                         in->image->emboss(in->radius, in->sigma);

                            return in;
                    }
            private:
105
            };

            class Noise:public ff_mnode{
            public:
110                 void* svc(void* tsk){
                            Task* in=(Task*)tsk;
                            if(in->name.find("Noise") != std::string::npos){
                                    in->image->addNoise(GaussianNoise);
                                    }
115
                            else
                                    in->image->comment(in->name);
                            return in;
                    }
120

            };

            class Write: public ff_mnode{
125         public:
                    void* svc(void*tsk){

                            Task* in=(Task*)tsk;
                             std::string outfile = "./out/" + in->name;
130                             in->image->write(outfile);

                                delete in->image;

                                return in;
135                 }
            private:

            };

140         int main(int argc, char *argv[]) {
                if (argc < 2) {
                    std::cerr << "use: " << argv[0] << " [-r radius=1.0] [-s sigma=.5] <
```

```
    image−file> [image−file]\n";
                        return −1;
                    }
145                 double radius=1.0,sigma=0.5;
                    int start = 1;
                    char *r = getOption(argv, argv+argc, "−r");
                       char *s = getOption(argv, argv+argc, "−s");
                       if (r) { radius = atof(r); start+=2; argc−=2; }
150                    if (s) { sigma  = atof(s); start+=2; argc−=2; }
                       timeval start_time, end_time;
                       gettimeofday(&start_time,NULL);

                       InitializeMagick(*argv);
155
                       long num_images = argc−1;
                       assert(num_images >= 1);



160
                       ff_pipeWithMerge pipe;

                       Read read(&argv[start], num_images, radius, sigma);

165                    Noise noise;
                       Enhance enhance;
                       Blur blur;

                       Write write;
170                   //std::freopen("./out/output.tsv", "a", stdout);
                       pipe.add_stage(&read);
                       pipe.add_stage(&noise);
                       pipe.add_stage(&enhance);
                       pipe.add_stage(&blur);
175                    pipe.add_stage(&write);

                       if(pipe.run_then_freeze()<0){
                                error("running pipeline \n");
                                return −1;
180                     }
                        pipe.wait_freezing();

                        gettimeofday(&end_time,NULL);
                        double execution_time;
185                     execution_time = diffmsec(end_time,start_time);
                        std::cout << "Finished Executing took total of " << execution_time/
    1000 << " seconds "<<std::endl;

                    return 0;
                }
```

Table of Contents


```
 1 AM.hpp.............. sheets   1 to   3 ( 3) pages   1-  3  174 lines
 2 Command.hpp......... sheets   4 to   4 ( 1) pages   4-  4   75 lines
 3 lb_withControler.hpp sheets   5 to   8 ( 4) pages   5-  8  251 lines
 4 managed_farm.hpp.... sheets   9 to   9 ( 1) pages   9-  9   35 lines
 5 pipe_withMerge.hpp.. sheets  10 to  15 ( 6) pages  10- 15  438 lines
 6 ImageFarm.cpp....... sheets  16 to  18 ( 3) pages  16- 18  167 lines
 7 ImagePipe.cpp....... sheets  19 to  21 ( 3) pages  19- 21  190 lines
```