UNIVERSITY OF PISA AND SCUOLA SUPERIORE SANTANNA

Master Degree in Computer Science and Networking

Master Thesis

Structured programming in MPI
A streaming framework experiment

Candidate : Leta Melkamu Jifar

Supervisor : Prof. Marco Danelluto

Academic Year : 2012/2013

To my family, who emphasized the importance of education, who has been my role-model for hard work, persistence and personal sacrifices, and who instilled in me the inspiration to set high goals and the confidence to achieve them. Who has been proud and supportive of my work and who has shared many uncertainties and challenges for completing this thesis and the whole masters program.

To Hatoluf, to Dani, to Basho, to Ebo and to *mom*!

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Marco Danelluto, whose expertise, understanding, and patience, added considerably to my graduate experience.
I appreciate his vast knowledge and skill in many areas (e.g., parallel and distributed computing, structured Skeleton programming, Multi/Many-Core and Heterogeneous Architectures, Autonomic computing), and his assistance in keeping me in the right track both in the implementation and thesis writing parts.
I would also like to mention his humble support in making the experimental machine available and installing all tools that are important for the experiment of this thesis.
And I also want to thank him for understanding my situation and be able to available for discussion; giving me time from his busy schedule to serve many students, attending conferences and conducting research works.

Finally, I would like to thank my family, my classmates and my friends for being supportive and proud.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Structured parallel programming can be seen as a model made of two separate and complementary entities: a computation entity expressing the calculations in a procedural manner, and a coordination entity, which abstracts the interaction and communication. A high-level parallel programming paradigm (aka Algorithmic Skeleton) is a well recognized form of structured parallel programming model based on patterns. A new kind of structured programming, based on patterns, is relevant to parallel programming today.
Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction [1][2]. While computation constructs manage logic, arithmetic, and control flow operations, communication and interaction primitives coordinate inter- and intra-process data exchange, process creation, and synchronization. Structured parallel programs are expressed by interweaving parameterized skeletons similarly to the way in which structured sequential programs are constructed.
This thesis deals with the design implementation and experimentation of structured algorithmic skeletons for stream parallelism. The implementation part of this work will produce a parallel programming skeleton framework/library that will provide a C language API (Application programming interface) to a programmer. The experimentation part will examine this new implementation with different experimental cases to produce cost models that suit stream parallel patterns. In that section we will also compare this implementation with other, existing stream parallel programming frameworks.
The implementation is based on MPI. MPI is a message passing library interface specification. There are different implementations that conform to this library specification. For this thesis work one specific implementation called open-MPI 1.6.3, the last stable version has been used.
Algorithmic skeletons were first introduced by Cole [1] in 1989. Several skeleton based programming frameworks have been proposed by different research groups using different techniques such as functional, imperative, and object oriented languages. There are quite fair amount of skeletons programming frameworks [4]. Each one provides different nesting mechanism, number of parallel patterns supported, degree of access to their source code, host language supported, type of architecture they target and distribution library they are based on.

This work is aimed at implementing a structured skeleton framework for stream parallel problem set. It aims at providing a programmer with a library that can easily be included and used to develop an efficient, scalable, and high performant parallel program. The work of the programmer is simplified with respect to the cases where unstructured programming models are used. It boils down to writing the usual sequential code and to be able to properly call the skeleton library passing the correct parameters. Though our skeleton library is implemented based on MPI, a programmer is not required to know MPI (approximately 0% of MPI knowledge is required) to use this library implementation. This work rather encourages programmer to concentrate on writing the sequential code and think about which skeletons to instantiate to achieve good parallelism exploitation. Discussion on the abstraction of the library implementation is more elaborated in chapter 3.
In this library implementation a programmer can feed the skeleton framework the sequential task that has to be computed along with parallelism degree which indicates *by what speed to accelerate the task computation.* The library will then instantiate the correct skeleton and it may also nest skeletons if it is necessary to achieve what is requested by the programmer. This comes from some programming efforts that have been done once by this work and then can now be reused by programmer, thus conforming to the definitions of structured skeleton programming. This is the aim of skeleton programming in general.
Our library supports full composition of the stream parallel patterns. Namely, pipeline, and farm parallel patterns. With this library it is possible to nest a farm skeleton in pipeline stage or to nest a pipeline skeleton in a farm worker. Though the later is a rare case, still an effort has been spent to make it available for those rare cases. This kind of composition requires that the tasks that have to be computed by a farm worker should be suitable to be computed using pipeline pattern; so that the sequential farm worker can be turned to a parallel pipeline pattern.
For example if a programmer put heavy computation at a farm worker, a sequential farm workers will execute this task taking long time with respect to the case when this worker have been turned to a parallel pipeline. The composition of pipeline skeleton in a farm worker could be used, provided that the computation given to a sequential farm worker is suitable to be decomposed and computed in a pipeline pattern.
A Phd. Thesis [20] shows such nesting usage to be important. In this paper an application requirement is presented, where farm worker is a two stage pipeline, in which the first stage applies a filtering operation to incoming item, and the second stage, apply FFT (Fourier Fast transform) operation on the filtered items. Without nesting, each worker of a farm should have been a heavy task to achieve performance.
So even though it is rare, there are some particular real world applications that makes use of the last form of composition.

The first composition (a farm in pipeline stage) on the other hand is very important in that it is the usual and well known to fit to most stream parallel problem set. Regardless of its correct usage, this type of composition is always possible. This is because, any computation that is given to a pipeline stage can be replicated to a farm workers. Each worker will execute the same code accelerating the computation with respect to the sequential stage.
A programmer can plugin farm whenever there is intense computation at a give pipeline

stage. Turning a pipeline stage to a farm skeleton is easier and will result in some advantages: achieving better completion time, resolving of bottleneck (as far as it is applied to a bottleneck stage), etc. Such a stage will be called a farmed-stage in this thesis writing.

1.1 Motivation

The problem of writing a correct and efficient parallel program is an issue. One has to deal with a low level programming effort, new ideas of parallel programming model, and also hardware advancement towards parallel programming.
For this reason it is convincing to come up with an abstraction mechanism that over comes the difficulties that arise in parallel programming. All issues related to parallel programming can be abstracted as much as possible. This includes abstracting the underlying implementation details such as communication between parallel nodes, avoiding deadlock between communicating nodes, imposing potential parallel execution whenever possible.
On the other hand there is also a need towards abstraction of distribution language used for implementation of the underlying skeleton framework. Programmers should not be left with distribution language used to implement the skeleton framework library. This knowledge requirement otherwise will impose the study of the distribution language prior to the usage of the skeleton library implementation. One motivation of this work is to provide this abstraction as much as possible.
The other motivation of this work is to experiment on structured stream parallel programming based on MPI. The experimentation will lead to the conclusion of in what degree to recommend the use of message passing model compared to shared memory model counterpart. The experiments will compare the library implementation with previous works (Fastflow skeleton library [23]). In addition to this, experiment of this skeleton implementation itself on different multi-core architecture is also considered.

1.2 Related Work

This library implementation (mspp-1.0.0 which stands for MPI Stream Parallel Patterns) is designed and developed from scratch but it has several commonalities with a number of well-known research works. There are fair amount of work on algorithmic skeleton implementations [4]. Algorithmic skeleton programming can be classified based on: a programming language interface they provide, execution language they use underlying, distribution language used to implement the skeleton framework, type safety, skeleton nesting mechanism, file access support, skeleton-set they support, a computer architecture they target (multi/many-core architecture, distributed architecture).

In the following some related work of previous years is presented.

i *eSkel (the Edinburgh Skeleton Library).*
Eskel is a structured parallel programming library developed at the School of Infor-

matics, University of Edinburgh (Scotland, UK). It offers a range of skeletal parallel programming constructs to the experienced C/MPI programmer. The first version of the library was developed by Murray Cole in 2002. This library of C functions, on top of MPI, aimed to address several issues raised by skeletal programming. The new interface and the novelty of the implementation allow addressing more concepts for more flexibility. eSkel's API semantics (as well as its implementation) are based on MPI. In order to make full, concise and effective use of this library, a programmer should first be familiar with the essential concepts of MPI itself.
The skeleton sets supported by eSkel are pipeline, farm, deal, butterfly, hallowSwap. Only the pipeline and Deal skeletons are fully implemented so far.
This skeleton framework implementation has some commonality with the our implementation in that it is based on MPI, it supports only stream parallel skeleton patterns and it targets shared memory.

ii *FastFlow [23] high-level and efficient streaming on multi-core [11].*
Fastflow addresses a programming framework targeting cache coherent shared-memory multi-cores. Fastflow is implemented as a stack of C++ template libraries.
It has a layered architectural design stack where the lowest layer provides an efficient lock-free and memory fence free synchronization base mechanisms. The middle layer provides distinctive communication mechanisms supporting both single producer-multiple consumer and multiple producer single consumer communications. The top layer provides typical streaming patterns exploiting the fast communication/synchronizations provided by the lower layers and supporting efficient implementation of a variety of parallel applications, including but not limited to classical streaming applications. One of principles of Fastflow is having a programming model based on design pattern/algorithmic skeleton concepts, that helps to improve the abstraction level provided to the programmer. FastFlow supports stream parallel patterns. This includes pipeline, farm and divide&Conquer.
This implementation also shares some commonality with our implementation. It supports stream parallel patterns and also targets multi-core architecture (in fact it also support distributed systems). Further more we have used this implementation to compare it with our implementation. It is state-of-the-art skeleton framework that has been used by a lot of academic community and research centers.

1.3 Thesis

This thesis work has a clear statement to address:
The thesis describes the implementation of a structured programming environment providing stream parallel skeletons (pipelines and farms) implemented on top of MPI and targeting shared memory multi-cores. The MPI framework performance is evaluated and the results compared with the ones achieved when executing the same applications using FastFlow.

The objective of this work is to be able to investigate the above goal using a library implementation based on message passing library specification (MPI). The implementation part includes structured parallel skeleton patterns such as pipeline and farm. It also focuses on composition of farm in a pipeline stages and vice versa.
We also give some attention to abstract the skeleton implementation as much as possible. As a result, we believe that the work of programmer is simplified.

The implementation part of the thesis will present all algorithmic detail documentations, explaining the algorithm step and logic behind it. Experiment and result collection part of this thesis work will provide graphical representation of efficiency, scalability, and other specific experimental cases. Comparison of this implementation on different multi-core architecture will be investigated to study how the skeleton implementation will scale with number of core.

1.4 Contribution Of This Work

There are many skeleton implementations, only few of them are based on MPI. MPI as a message passing interface library specification is standardized as the best message passing mechanism. As a result MPI is a portable and efficient library to use. There are plenty of documentation and also implementations towards MPI.
In many previous works skeleton frameworks give a programmer a way to access the source code, compose skeletons, use native programming language and so on. Depending on the different type of those frameworks they may require a programmer to know some more knowledge of the underlying distribution library that is used to implement the skeleton framework. We believe that it is important to abstract skeleton implementation as much possible so that programmers will not deal underlying distribution library.
One issue of message passing programming model is the communication latency that is imposed while communicating messages between processes.
Performance of MPI for shared memory can be tuned using its share memory configuration mechanism or using external tools. With KNEM[17] it is possible to perform a direct copy of data bytes from one process memory address to another process memory address, which otherwise will need two copies (first from sender memory address to shared memory and then from shared memory to receiver address). Detail usage of KNEM is presented in section 2.9.2 of this thesis.
Furthermore, this implementation provide item ordering mechanism. It is a way to order items that may get out of order while computed by concurrently executing nodes. This mechanism gives an efficient and parametric algorithm that can be plugged into any skeleton node where item ordering is required.
Measuring over all elapsed time of parallel executing node is also implemented to properly measure over all execution time of the stream elements. The experimentation part of this work will also study the difference between message passing model and shared- memory model.

1.5 Overview

The remaining part of this thesis is organized as follows:
Chapter 2 will deal with main concepts, where all issues related to structured stream parallel skeletons and terminologies are defined.
The second part of this chapter will present background on the underlying distribution language used and its shared memory configuration.

Chapter 3 will present the architectural design of the skeleton framework implementation. Each layer of the architectural design is explained providing detail information and functionalists of the layer.

Chapter 4 will present details of algorithms implemented for this work, how low level MPI communications mechanism are used and handled, how blocking and non-blocking operations of MPI are used, how item ordering algorithm is implemented, how overall elapsed time of parallel execution is measured, and how skeleton composition is achieved.

Chapter 5 will cover the experiments. It will present issues related to stream parallel skeleton experimentation. Here we present methodologies used for the experimentation, tools used to experiment and the future of hardware machines used for the experimentation.
In the second part of this chapter we will present experimental results along with data collected and diagrammatic representation of the results.

Chapter 6 will conclude the thesis, and also present a future work.

At the last we also present all source code of the library implementation in Appendix A.

Chapter 2

Stream Parallel Skeleton Patterns

Parallel computing is related to the simultaneous use of multiple compute resources to solve a computational problem. It is a way to carry out many calculations in parallel.

The parallel programming paradigm allows executing sequential programs in parallel with other sequential programs. As a result, it achieves a better completion time, and better utilization of available resources with respect to the sequential execution. There are several forms of parallel computing: bit-level, instruction level, data, and task parallelism.
In this work we are interested in the stream/task parallelism.

Stream parallel computations are one of the usual parallel programming paradigms. Such paradigms are designed to be applied to stream of input task items. Parallelism is achieved by applying the defined task computation on different items of the stream at the same time. Basically, at least two computing nodes are required to be able to trigger stream parallel computation. In this case item(i) and item(i+1) of the stream will be computed in parallel. Plugging in more computing nodes will result in more parallelism, and more computation and communication overlap. Depending on the number of items to be processed and on the number of parallel computing nodes available this may be beneficiary.

Stream parallel computation may impose data dependencies on the tasks to be performed. However each computing node, once given a data-item to work on, can compute the task without needing any data from other nodes. Here computation node means concurrent components in a given parallel skeleton patterns that run in parallel with in other concurrent component in same skeleton framework. This can for example be a pipeline stage, a farm worker, farm Emitter, farm collector, etc.
Depending on the patterns used computations may be required to communicate to different node; in which case the order of applying functional operation matters. Pipeline stream parallelism is typical example of this where each stage participate in producer/consumer interaction. A stage accepting an input will need to communicate its output to the next stage or to the external (fictitious) output buffer.

2.1 Pipeline and Farm Parallel Patterns

2.1.1 Pipeline Pattern

A pipeline is composed of a linear series of producer/consumer stages, where each stage depends on the output of its predecessor. The producer and the consumer stages should be compatible in terms of the type of data they communicate.
In pipeline pattern parallelism is achieved by executing a task on different item of a stream. The following figure shows a simple pipeline structure.

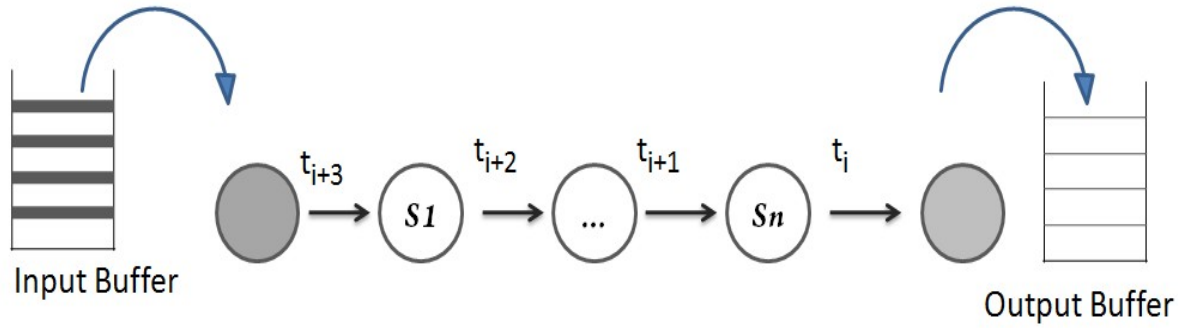

Figure 2.1: A Pipeline pattern

As shown in this figure tasks t_i, t_{i+1}, t_{i+2},etc. will be computed in parallel. Each stage of a pipeline may execute different or similar computation. This depends on the nature of application that will be executed using the pipeline pattern.
The pipeline is an example of a more general category known as a data-flow network. A data-flow network decomposes computation into cooperating components that communicate by sending and receiving messages.
The Pipeline paradigm is a way that lets you to achieve parallelism in cases where there are linear data dependencies.
Pipelines are applicable in many computing areas. We can use a pipeline when data elements are received from some real time item generating entity. For example this can come from a user-generated mouse click events, values on stock ticker tapes, or packets that arrive to the NIC over the network. We can also use pipelines to process elements from a data stream, as is done with compression and encryption, or in image processing where different stage of the pipeline computation adds different effect on the input image item, or to apply transformation operations to streams of video frames. In all of these cases, it's important that computation is carried out by respecting data-dependency constraint.

A pipeline is made of at least two producer/consumer stages. In this implementation the "two stages" requirement are in addition to the external stream generator and stream consumer. As shown in figure 2.1 the stages shown in gray color represent the stream generator and stream consumer. A "stream generator" stage is a node that is responsible to generate

a stream by drawing item from an input buffer and "stream consumer" stage is a node that is responsible to collects the final result and put them in output buffer. Those last skeleton framework nodes are external to the skeleton parallelism pattern. This is preferred for very good reasons; first, it is important in making the skeleton implementation modular. Second, it allows implementing skeleton composition in a simple and structured way. This means that, a skeleton pattern is a standalone entity without the stream generator and stream consumer nodes. For any stream source the skeleton pattern will function with it as long as it is made compatible. As a result while composing skeleton patterns those external nodes will not be included in the composition process. Instead they will be applied to the skeleton framework after the composition processes is done.

A given generic pipeline stage can be a sequential stage or a parallel stage. A sequential stage is a pipeline stage defined by a sequential task. This stage will execute a task in sequential manner in the sense that, it should finish executing a task on a given item before proceeding to the next item in stream.
A parallel stage on the other hand is a stage that executes a task on different items simultaneously. Such stage is made by nesting other skeleton patterns in a given pipeline stage. The nested skeleton will have multiple computing nodes that execute a given task on different items. This issue is further explained in 2.7.

Different stages of a pipeline may execute different or similar functions. In general a pipeline stages need to be compatible in terms of the type of data it exchanges between the paired producer/consumer.
In this implementation however, the requirement is a bit changed because of the underlying distribution language used. MPI requires that processes need to communicate data by specifying a byte size to transfer. The process that sends a given data will specify how many byte to read starting from the address pointed by a send buffer. Similarly the process that receives data will read the specified amount of byte starting from the address pointed by the receive buffer.
As a result, instead of expecting a strict compatible data type, each stage expects a byte size of data specified at skeleton framework topology creation time. This means that a stage sees the data sent to be compatible as long as their byte size is equal to what is expected by that stage. The requirement explicitly expects that a function defined by a programmer should be able to accept the specified byte size and also produce the required amount of byte size. Wrong byte size information will generate error or function erroneously.

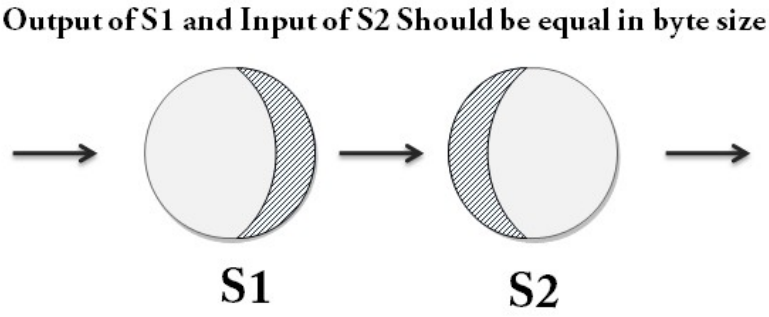

Figure 2.2: Pipeline requirements towards input and output byte-size of each stage

As mentioned above a stage can be sequential or can further be turned to another skeleton to run tasks computation in parallel. The later argument in skeleton terminology is called composition. In this work we have developed a way to compose farm skeleton in pipeline stages. Any stage (or all of them) in a pipeline can be turned to a farm skeleton. The implementation also addresses the composition of a pipeline skeleton in a farm worker.

As in any stream parallel patterns, an important issue to mention here is a stream generation mechanism. A streamer is an external entity that has to feed tasks to the skeleton pattern as fast as possible; so that all stages will compute a task on different items of a stream in parallel. Streamer consumes the input buffer provided by a programmer and each time draws an item from this buffer to create a stream. Please refer to section 3.3.1 for more information on stream generation.
We conclude this section by presenting a pipeline pattern design issues.

Pipeline Pattern Design Issue

When we use the Pipeline pattern to decompose a problem, we need to consider how many pipeline stages to use. This depends on the number of cores we expect to have available at run time, as well as the nature of the application we are trying to implement. Unlike farm, pipeline pattern doesn't automatically scale with the number of cores. This is one of its limitations. To achieve a better degree of parallelism, the stages in the pipeline should be balanced in terms of the amount of time needed compute a task. If it is not balanced however, the slowest stage (aka bottleneck stage) will impose greater completion time; thus affecting the overall performance of the pipeline. The number of stream items with respect to the number of pipeline stage is also an important consideration for the overall performance. We don't want to have a situation in which only fill-in transient phase and emptying transient phase of pipeline pattern exist. Instead we want to have a situation in which the pipeline run in steady state for long duration in the entire computation.
A fill-in transient phase is a phase of pipeline computation that happens when the first item of a stream did not yet reached the last stage of the pipeline. Or in general, it is the time when some nodes of the pipeline are idle and the others are working. Emptying transient phase is a similar issue except that while fill-in transient phase happen when items are start

flowing in a pipeline for the first time, emptying transient happens when the last items of the stream are on processes; in which case some nodes at the beginning of the pipeline stage are idle. All the rest of time, where all the pipeline nodes computing a task is called the steady state.
The fill-in transient and emptying transient should only exist at the start and end of the pipeline execution phase respectively. Of course this is also affected by the stream generator entity.

2.1.2 Farm Pattern

Parallel tasks are asynchronous operations that can run simultaneously. Task Parallelism is a pattern that provides a way to execute the same computation on different input tasks at the same time. Concurrent entities of such pattern execute task in a potentially independent manner, thus in parallel. In farm pattern such concurrent entities are called farm workers.
Parallelism is achieved by assigning different items to available workers which will then execute in parallel.
Figure 2.3 shows a diagram representing farm pattern used for this implementation.

In addition to a workers a farm pattern may also include a node used to schedule task to workers and a node used to collect results from workers. Those nodes are called emitter and collector respectively. In fact the use of emitter and collector nodes is implementation dependent. In this implementation emitter and collector are made to be part of farm pattern. A farm emitter is a module/node that is responsible to schedule a task to farm-workers. Scheduling is an important aspect of parallel tasks.
A scheduler needs to direct the task to specific worker using some scheduling mechanism. There are different mechanisms in which an emitter can schedule a task, round-robin and on-demand. In this implementation we have used a round-robin scheduling strategy. An emitter will have a pool of workers Id where it can perform a round-robin scheduling. Implementation of detail of this strategy is given in chapter 4.

Farm Collector on the other hand is responsible to collect items sent from workers. It will accept an item form any of those workers. One can easily tell that this is a potential point where items can get out of order. There is no guarantee that worker finish their work according to the order in which they were scheduled. Item ordering strategy implemented in this work is made available to be used as farm-collector or at the last stage of the pipeline. Section 3.4.3 presents item ordering strategy. And implementation detail is given in chapter 4.

A very common form of Task parallelism paradigm is a master-worker paradigm. It is a farm skeleton made of Emitter, workers and collector. There are different variant of farm paradigm.

i Decentralized Emitter and Collector (in a tree or ring structure).

ii Emitter and Collector are generalized as one node (aka master-worker).

iii The normal and most used form is a farm where Emitter and Collector are different nodes and workers are arranged to be scheduled by the Emitter either in round robin or on demand scheduling.

In this work, we use the last form of farm paradigm. The following figure shows such farm parallel pattern.

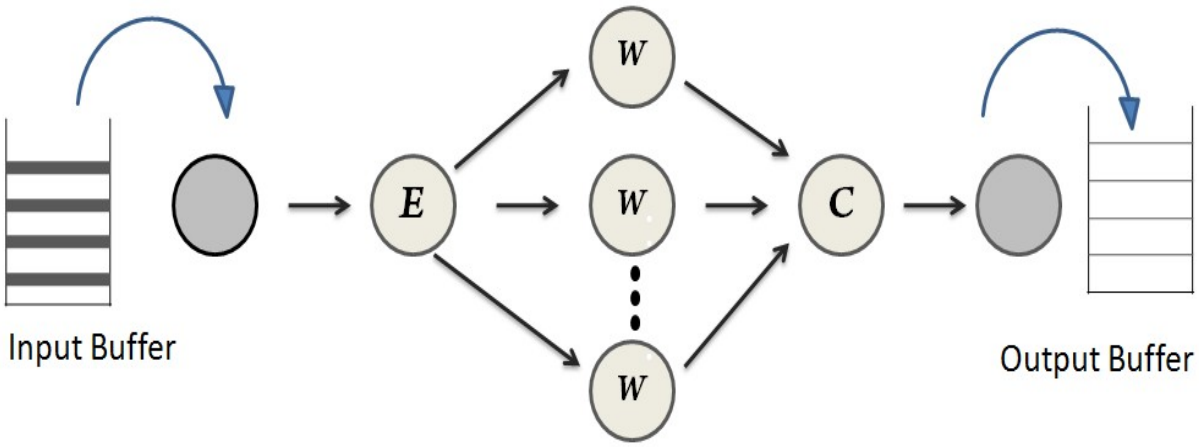

Figure 2.3: A Farm parallel pattern

To be able to consider the case were farm worker can be further turned to a pipeline skeleton, the data structure of a worker accepts either sequential function pointer or a pipeline structure.
Implementation detail of farm worker is given in chapter 4. Here we conclude this section by presenting a farm pattern design issues.

Farm Pattern Design Issue

A farm paradigm is a good parallelism pattern that can scale with number of cores. As long as the computation can be applied to the input items independently farm skeleton is the best stream parallel paradigm. We have to be careful in choosing number of farm worker to use because it will need to have a resource to execute on. Using too much farm workers with respect to available number of cores may degrade performance. The operating system will start performing some context switching between processes that have to use the available CPU resource one after the other. While our aim is to execute computation as parallel as possible, we don't want experience a context switching from the operating system. Using too few farm-workers is also not a good design choice. This will under utilize the available resources. So number of farm-worker is a critical parameter in farm paradigm. In chapter 5, We have experimented how farm paradigm scales with number of cores, and number of farm-workers.
The other issue with farm is that items can easily get out of order while computing. An

ordering farm-collector can be used to order items. Unless required, the use of item ordering farm-collector is not a good design decision. This is because first, the ordering node needs to have a local buffer where out of order item can be stored, second it delays communication of buffered item to be serialized later. This for sure will increase the completion time of the overall computation. It is recommended to avoid the use of item ordering farm. Section 4.5 motivates this recommendation. If a farm is nested in a pipeline stage, ordering can be at farm collector or at the last stage of the pipeline. We strongly recommend that if most of the stages of a pipeline are farmed, it is better to use the item ordering at the last stage of a pipeline instead of imposing it at each farmed stage. This actually is a trade-off. While the use of item ordering at farm level is worse because of the above reasoning; when a farm computes at farmed stages, more items will get out of order at each of such stage, and thus the ordering node presumably overflows its buffer trying to buffer more items.

2.2 Programming Model choice

Shared memory and message passing are two different models of parallel programming. Both are applicable in different computation areas. They have advantage and disadvantage in accordance with their applicability.

Shared Memory

In shared memory programming model, tasks share a common address space, where they can read and write asynchronously. This doesn't come for free; one has to deal with various mechanisms such as locks/semaphores to properly share a memory space between potential tasks running "in-parallel". In this model there is no need to explicit communicate data between tasks. On the other hand it is very difficult to understand and manage data locality. It is not that obvious to keep data local to the processor that works on its conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processors use the same data.

Message Passing

In a message passing model, parallel tasks do not share any address space and therefore they exchange data through explicit passing messages. These communications can be asynchronous or synchronous (or blocking/Non-blocking). In an implicit message passing model, no process interaction is visible to the programmer, instead the compiler and/or run-time is responsible for performing it. This is most common with domain-specific languages where more concurrency within a problem is involved. Message passing model opts for tasks that use their own local memory during computation. It is possible to have tasks that reside on the same physical machine or on an arbitrarily distributed number of machines. Tasks exchange data by explicit message passing. The operation requires to be performed by each process, thus the need for two sided operation. MPI is a de facto industry standard for message passing, replacing virtually all other message passing implementations used for production

work.

For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons. MPI implementations such as "openMPI" use the optimal communication as much as possible. This applies both on a shared memory and also in a distributed system. In distributed system, the option is to choose the fastest networking technology. It detects the available resources automatically and tries to use the better communication channel as much as possible. This is the good decision of MPI implementations. Though this is an optimization towards the traditional message-passing model, it is still not performant with respect to the thread-based implementation on shared-memory. We can see that the shared-memory memory usage by MPI implementation is not a direct copy of data from one processes memory address to the other process memory address. Instead a two phase copy will be performed each time a communication is made; first from the sender buffer to the shared memory and then from the shared-memory to the receive buffer (where the receiving operation can then read to use it as an application data). This multiple copy will hit performance sensibly. There should be a way to copy data from one process memory to the other directly without any intermediate copies. For this a high performance intra-node MPI communication (KNEM) [18] is implemented and released for usage since October, 2008. Open MPI support KNEM since its 1.5 version release. For detail explanation on KNEM please refer to section 2.9.2. With KNEM a zero copy communication is possible between memory address of MPI processes. It is at least feasible to compare MPI-based implementation with shared-memory based implementation with the use of KNEM kernel module.

2.3 Stream generation

Stream parallel paradigm such as pipeline and farm are meaningful only on stream input. Stream generation can be primitive; where real time external entities like user-generated mouse click events, values on stock ticker tapes, or packets that arrive to the NIC over the network. Or a stream can be generated from a program/skeleton-framework. In this last case one specific node is reserved to generate a stream from a given input buffer. And another node is reserved to collect the overall results to an output buffer. In the middle is a node that represents a parallel skeleton pattern that is used to run the computation in parallel. This is the main mechanism used in this work to make composition of skeletons modular. According to [15] this is called "Unpacking-compute-packing". A single data structure can be converted to a stream by using the *"unpacking-compute-packing"* mechanism; provided that the data structure is suitable to do so.

The following figure shows the *"unpacking-compute-packing"* scheme.

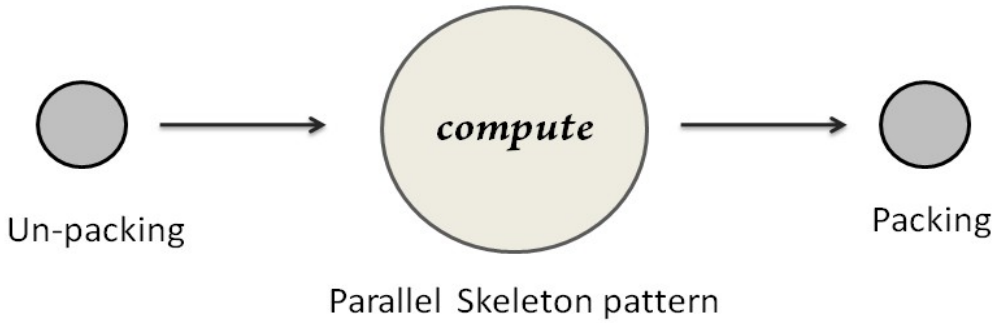

Figure 2.4: unpacking-compute-packing way of stream generation

The *"unpacking-compute-packing"* mechanism is a way to generate stream from the skeleton framework. A stream generator will be given a buffer from where items can be drawn. section 3.3.1 will discuss this issue in detail.

2.4 Function Replication and Partitioning

Stream parallel paradigm can support either function replication or function partitioning based on the specific pattern used. A farm paradigm works with function replication in which each worker of the farm executes the same function. Workers apply this function on different item of the stream in parallel. While pipeline paradigm usually works with function partitioning in which its different stages execute different operation in-order to compute overall computations. [15] Shows that pipeline paradigm can also apply to "function replication". A good example of this is the case where a loop-execution is converted to a pipeline computation. This last case is called loop-unfolding. We know that a loop executes the same task multiple times on the next item of the loop. This can be transformed to a pipeline computation by partitioning some part of the computation to be done by different stage of a pipeline. Each stage will then perform the partial computation of the overall loop computation and communicate it to the next stage. That way the overall computation can be done in a pipeline pattern. However in general the use of pipeline is more meaningful with function partitioning. Mathematical function composition is good candidate of pipeline computation.

2.5 Scalability Of Stream Parallel Skeleton Patterns

Scalability is an important factor that parallel programming frameworks in general have to provide.
Parallel program written and executed in one multi-core machine should be able to also run on another machine, with greater or lesser number of cores, scaling with the number of cores in the new machine.

Different skeleton patterns have different scalability future. A farm skeleton is one of good skeleton patterns that can scale with number of cores available on a machine. The scalability can be achieved by using different number of farm-worker that will make use of the available resource. In farm a function will be replicated to number of workers. So, it is possible to run the application with variable number of workers each time.
As presented in section 2.1.2 under "farm design issue", number of farm workers is a critical parameter that has to be selected carefully. We have presented an experimental case that shows how a farm paradigm can scale with number of core. A programmer can run a given application using farm pattern by passing different number of farm-workers which can be given while running the program from command line.

Pipeline pattern on the other hand is not expected to scale with the number of cores. Scalability can be experimented if some stage of the pipeline is turned to be a nested parallel skeleton such as farm or if the number of pipeline stage is greater than the available cores. In pipeline the number of stage is defined according to the computation that has to be done in a pipeline pattern. In other word a pipeline uses function partitioning as presented in previous section. If possible, it is a good design to decompose a computation in a given single stage to be done in a different multiple stags. This way the number of stages can be made to fit to the number of available cores. In fact making a pipeline stage course grained may be beneficiary in terms of minimizing communication latencies. However, since we have implemented an optimal communication and computation overlap, it is recommended to have fine grained pipeline stages; provided that the resulting number of pipeline stages are less than or equal to the available resource used to handle them. Imposing too many stages compared to available number of core will result in to have a serialized computational stages; that is interrupted by the underlying operating system to context switch processes for proper share of the available resources. Such inefficient decision has to be avoided as much as possible.
It has to be noted that, the above paragraph is not concluding that it is not possible to measure scalability of pipeline pattern, instead it is underlining that, as a number of core (or resources) increases pipeline will not make use of the available resource more than its number of stages. In fact we have experimented and measured scalability of pipeline.

The scalability is defined as a measure of how efficient is a parallel execution with respect to the execution of parallelism degree 1. Formally it can be given as follows:

$$\text{Scalability} = \frac{T^{(1)}}{T^{(n)}}$$

where $T^{(1)}$ is a execution time of parallelism degree 1 and $T^{(n)}$ is execution time of parallel computation with parallelism degree n.

2.6 Efficiency Of Stream Parallel Skeleton Patterns

Efficiency is defined to be the ratio between the ideal completion time and the actual completion time. This parameter provides information of how the parallelism computation time is close to the ideal one. This means that how the parallel execution makes use of the available resource.
Efficiency is calculated as follows

$$\epsilon = \frac{T_{id}^{(n)}}{T^n}$$

where $T_{id}^{(n)}$ is the ideal completion time of a parallel computation and T^n is actual parallel computation time,both with parallelism degree n.

The ideal computation time is calculated with respect to the calculation time of the sequential time as follows:

$$T_{id}^{(n)} = \frac{T_{seq}}{n}$$

Where T_{seq} is sequential completion time and n is parallelism degree used to accelerate the sequential computation

2.7 Algorithmic Skeleton Nesting

Murray Cole, in his Phd thesis [1], [2] proposed algorithmic skeletons that encapsulate a single parallel pattern. The use of those patterns was proposed to be stand alone patterns. This means that skeleton patterns can only be used alone. In this case programmers will have few options to choose a suitable parallel pattern for their application. The need for skeleton framework composition is foreseen by the skeleton community. According to [12] there are two perspectives towards the investigation of composition of skeleton patterns: "CISC"-skeletons and "RISC"-skeletons. The two perspectives are different in the number of skeleton they support and the option of allowing skeleton nesting.
"CISC"-skeletons: In this category the skeleton community come up with the idea to provide a programmer with a list of many[1] skeleton patterns each one addressing different common application problems. As a result the set of skeleton patterns provided to an application are very huge. This last fact imposes the problem of finding a good match of skeleton pattern for the application problem at hand. Besides this "CISC"-skeletons does not allow programmers to nest skeletons. So in case of unsatisfactory result is realized for the current usage of the skeleton framework application programmer has to remove the use of the skeleton pattern and look for another in the set.
"RISC"-skeletons: In this category on the other hand, there are few skeleton patterns

[1] A huge number of skeleton patterns

supported. They are made general and abstract so that their use will be quite usual. Programmers can easily pick a suitable skeleton pattern without much confusion. Furthermore "RISC"-skeletons also supports skeleton nesting.
In skeleton composition a full composition can be supported. However not all composition are useful. [12] presents a composition rule called ***"Two tier model"***. The idea come from the group of Computer science department at University of Pisa, while developing P3L [26]. According to this model there are some skeleton nesting/composition that have to be disallowed. One typical restriction they made was, if a data-parallel skeleton have to be nested in stream parallel skeleton pattern; the stream parallel skeleton should be at higher level in skeleton nesting tree. So there are some nesting that doesn't even make sense. There are also skeletons nesting that are usual less used. Even though they are feasible, those nesting are very rare to be used with respect to a real world application programs.
In this implementation a full composition of pipeline and farm is supported, such composition is ok according to the "Two tier model".

2.8 Computations Suitable For Stream Parallel Patterns

There are quite a lot of applications suitable to the usage of stream parallel skeleton framework. The application type is classified based on weather a stream is generated primitively or programmatically. The following are some of the typical examples of stream computation: the primitives can be real time user mouse click, packet arrive to a network interface, image processing, video frame processing, etc.
Image processing is one of an interesting application area where operations are applied to images in sequential order one after the other, thus conformant to pipeline computation pattern. There are also singleton (and expensive) image operation that can also be applied only once. Those operations will impose a bottleneck behavior on the overall pipeline computation. If they have to be used as a stage of a pipeline looking for composition of farm at that stage is a recommended design decision. Or in case the operation is the only operation to be applied to an image streams then farm pattern will be suitable to accelerate the computation.

The following figure shows a simple pipeline based image processing.

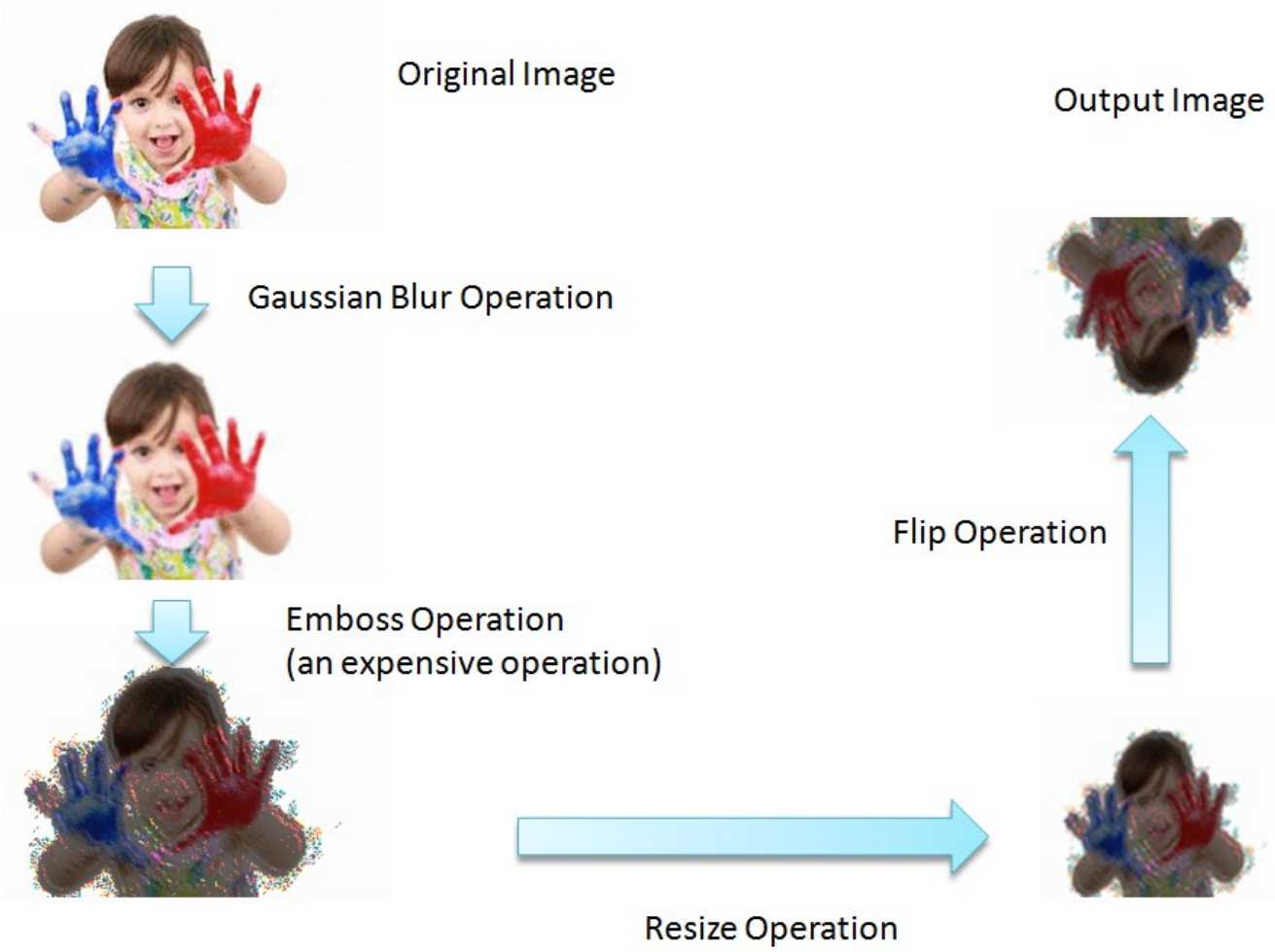

Figure 2.5: Pipelined image processing where the first stage performs "Gaussian Blur" operation,second stages performs "Emboss" operation, the third stage performs resize operation, last stage will flip the resized image.

2.9 Tools used

In this section we present tools used for implementation of this framework. This includes the underlying distribution language used for this implementation and some of its primitives. We have also presented a particular Linux kernel module used for a zero copy communication.

2.9.1 Distribution Language Used For Skeleton Implementation

A previous work shows that there are different skeleton framework implementations that are based on different distribution language. The distribution language used is crucial towards the efficiency and optimization of the underlying skeleton implementation and proper use of its constructs to get advantageous support. On other hand, one has to give attention that

programmers who are going to use the skeleton framework will be exposed to the knowledge of distribution language.
In this implementation MPI (Massage Passing Interface) is used. MPI is a message passing interface library specification that is ready and open to be implemented by different implementers. The different implementation of MPI has to conform to the minimum requirement of the MPI specification. For this specific work open-MPI 1.6.3 is used.
We work hard on the implementation of the skeleton framework, so that an abstraction of the distribution language is achieved. As a consequence we able avoid the requirement of the distribution language (thus almost 0% knowledge of MPI is required by programmers).
While the message passing model can be more difficult to program, more so for irregular applications, its potential advantages are better performance for coarse-grained computation and the fact that once communication is explicitly coordinated with sends and receives, synchronization is implicit in the send-receive pairs.
MPI as a message passing interface specification, it works with processes rather than threads. Processes have their own private virtual memory address which is meaningful for that specific process only. Thus it is expected there should be some mechanism to copy message from one process memory address to the other so that it will be understandable and used in the receiving process memory address. For this the MPI specification uses an explicit message passing where messages are copied from sending memory address and posted on a well-defined channel where the receiving process can read to its receiving buffer and use it for applications data. Most MPI implementations try to overcome this overhead by introducing a mechanism that automatically detects the environment where MPI is running and try to use the fastest way of copying a message between processes. This applies both for shared memory address and distributed memory addresses. In distributed address MPI implementations such as open MPI detect the good network links[2] available and use it. In a shared memory on the other hand open MPI try to use a sm-BTL [8] (A shared memory byte transfer layer) mechanism. In this mechanism a message passing is optimized by copying message from a send memory address to a shared memory and then from a shared memory to the receiving process receive buffer. However there are still multiple copy to pass message between processes. Though it is an optimization with respect to traditional message passing mechanism it still hit performance when compared with shared memory programming model counterpart.
This last problem is not left behind and forgotten by high performance community. The Inria RUN TIME team, a team working on a high performances run-time systems for parallel architectures, come-up with a solution called KNEM [17] (High-Performance Intra-Node MPI Communication mechanism). KNEM is presented in the next section.

There are some important papers comparing different message passing model implementations and also papers that compare message passing with a shared memory model. Ngo and snyder [6] compared several cache-coherent shared address space against MPI versions running on the same platform. They found out that the cache-coherent shared address space programs could perform as well as message passing ones. In fact the program they used was

[2]A switched fabric communications link used in high-performance computing and enterprise data centers such as infiband, Myrine, 10-Gigabit Ethernet.

not written well to take locality into account. And they only examine a single problem. LeBlanc and Markotas [7] on the other hand by comparing message passing with shared memory programming model, they conclude that shared memory is preferable in multiprocessors where communication is relatively cheap. As the cost of communication increases in shared memory multiprocessor, message passing is becoming an increasingly attractive alternative to shared memory.

2.9.2 Configuring MPI For A Shared Memory

As presented in the above section message passing libraries without any external support will be worse in terms of performance. We need a way to avoid multiple copies while communicating processes passes a message to each other. While configuring open MPI for shared memory is explicit during installation, this configuration as presented above will still impose multiple copy of messages. The use of this configuration is at run time. The following shows a sample MPI run command that accepts a parameter to determine shared memory byte transfer layer (sm-btl). The *"self"* flag is used for a communication of a process with itself, and the *"tcp"* flag is used in case not all processes in the job will run on the same single node, which needs to specify a BTL for inter-node communications. The *"mca"* flag means that a modular component architecture. Open MPI is very modular. It has its own component model called *Modular Component Architecture*.

`Open MPI shared memory configuration`

$-- \; mca \; btl \; self, sm, tcp$

KNEM is a Linux kernel module enabling high-performance intra-node MPI communication for large messages. KNEM works on all Linux kernel since 2.6.15 and offers support for asynchronous and vectorial data transfers as well as offloading memory copies on to Intel I/OAT hardware. Almost all MPI implementations support KNEM.

Motivation Behind KNEM

MPI implementations usually offer a user-space double-copy based intra-node communication strategy. It's very good for small message latency, but it wastes many CPU cycles, pollutes the caches, and saturates memory busses. KNEM transfers data from one process to another through a single copy within the Linux kernel. The system call overhead is not good for small message latency but having a single memory copy is very good for large messages (usually starting from dozens of kilobytes). Some vendor-specific MPI stacks (such as Myricom MX, Qlogic PSM, etc.) offer similar abilities but they may only run on specific hardware interconnect while KNEM is generic (and open-source). Also, none of these competitors offers

asynchronous completion models, I/OAT copy offload and/or vectorial memory buffers support as KNEM does. Once KNEM is installed in a Linux machine it can be made as part of a Linux kernel by using the command *"modprobe knem"*.

With KNEM support, Open MPI able to perform "zero-copy" transfers between processes on the same node - the copy is done by the kernel device and it is a direct copy from the memory of the first process to the memory of the second process. This dramatically improves the transfer of large messages between processes that reside on the same node.
Open MPI integrated with KNEM provides a different flag that can be used when an MPI program is started. Below, some commands with description is given.

`KNEM run time flags for openMPI`

1: $--$ *mca btl_sm_eager_limit* 32768. ▷ Change eager limit to 32768byte
2: $--$ *mca btl_sm_use_knem* 0 ▷ Disable KNEM at tun time
3: $--$ *mca btl_sm_knem_dma_min* 1048576 ▷ offload copies to DMA engine, starting from 1MB

We have installed and used this kernel module. Experiment results collected after the installation of this kernel module are much better than the ones when it is not used. After this installation a comparison of message passing implementation with shared-memory implementation is promising. Although KNEM is optimized for large size message passing, we use it in all experiment. In fact we use it with proper parameter tuning in large vector of data communication such as *Gaussian elimination* computation that work on stream of matrices.

Chapter 3

Architectural Design

In this chapter we will present architectural design of our skeleton frame work.
Architectural design of this implementation is simple and spontaneous from the nature of stream parallel pattern and the nature of the underlying distribution language (message passing programming model) used.

The following figure shows a simple architectural design.

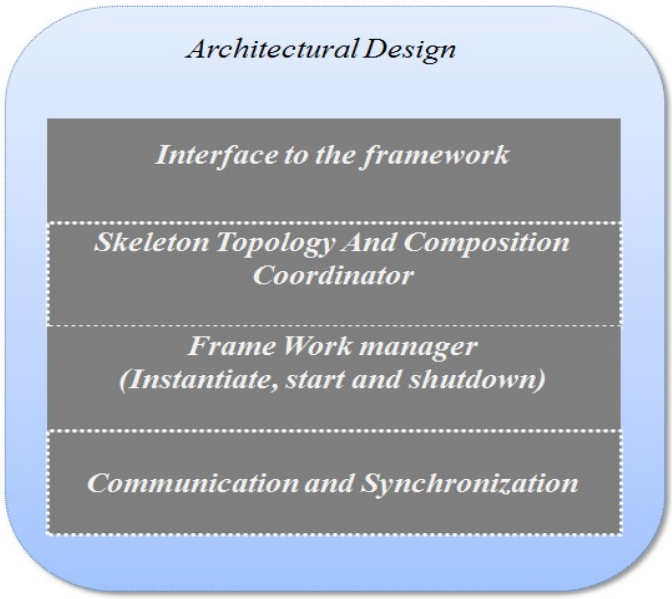

Figure 3.1: Architectural design of the implementation

This figure shows the conceptual layer of the design of this implementation.
The remaining part of this chapter will discuss on each of the architectural design layers.

3.1 Interface To The Framework

The top layer of the architecture is the *"interface to the framework"*. API of the framework which a programmer may use to invoke the library services.
A programmer can feed the skeleton with different parametric settings including the sequential code that needed to be executed by the skeleton framework. Parameter setting is a good way to tune the skeleton framework to better suited the application under consideration. Skeleton composition is one of such parameter that a programmer can specify to achieve better fit of the computation under consideration.
In general this layer is responsible to accept a programmers parameters and pass to the next layer.
The details of the library implementation are abstracted by using this interface. As a result the use of the library is simplified.
A programmer can easily write a parallel program by the support of this library interface. After a design decision is made based on the application to be computed; the remaining task is to call the library interface by passing proper parameters. The parameters that have to be passed to the library interface vary according to the chosen skeleton pattern. Different skeleton pattern requires different parameter. Furthermore parameters can be used to form composition of skeleton patterns. A programmer can specify the composition of skeleton patterns by passing correct parameters.
The library interface can also be used to choose between different functionality of the skeleton patterns. For example a farm pattern can be chosen to be *"ordering-farm"* or *'non-ordering-farm"*. Again, such choice will be made by passing the correct option to the library interface.

3.2 Skeleton Topology And Composition Coordinator

The next layer in the architecture is *"skeleton Topology And Composition Coordinator"* this layer is responsible to take into account the programmer parameters passed to it from the above layer. It is a very critical layer which takes care of creation of process topology and proper composition of skeletons.
The following are the main task of this layer.

i It calculates the optimal number of MPI processes that has to be started.

ii It assigns MPI processes into different computing nodes putting enough amount of processes in each of them.

iii Creates communicator for the overall skeleton

iv Creates skeleton topology by also considering skeleton composition.

3.2.1 Restricting Number Of MPI Processes That Have To Be Started

A skeleton framework abstraction of this implementation, in addition to other objectives, it aims to avoid requirement of knowledge of the distribution language (MPI). As a result the library will need to guide in how many MPI processes a programmer has to instantiate while running the application using the skeleton framework. The library will automatically calculate and check if the exact expected number of MPI processes is instantiated, if not it will inform a programmer the expected number of MPI processes and exit the program. A programmer will be expected to run the application again with the number of processes informed in the notification message. This restriction is important first, to avoid extra MPI processes started which otherwise consume resource for nothing, or to avoid the situation where insufficient MPI processes are started which otherwise leads to inefficient computation. Second, this is important for the library to create the correct MPI process topology (thus the skeleton topology).
The calculation of number of processes is done by considering all skeleton nodes to be a separate MPI processes. For example In addition to all potentially parallel computing nodes, each of the external stream generator, the external stream consumer, farm emitter, farm collector will be counted as a separate MPI processes.
A design of this restriction comes from the fact that, all MPI processes have a pre-assigned task which is identified by their process Id. No MPI process can enter a block of code defined for other processes and do whatever it wants. And it is because of this last reason that we need to build the skeleton topology from MPI processes Id. Under this light now, it is not important at all to start MPI program with too many processes than expected number. If exist those processes that got processes Id of greater than the expected one will be alive in MPI process pool but will do nothing else than simply consuming resource. In this implementation such a situation is not allowed. Only expected number of process will be acceptable by the library.
In relation to this is how to measure scalability with number of cores. Different skeleton patterns have different scalability future. Section 2.5 presents this issue.

3.2.2 Skeleton Topology Creation

Many early skeleton implementations focus on the abstraction of the underlying implementation of the skeleton framework. This is very important in sense that a programmer using this skeleton implementation will not be exposed to details of the implementation. Abstracting the distribution language used for the skeleton framework implementation is one of such interest.
In eSkel [5] library, the implementation of the skeleton is simplified but at the same time it exposes the distribution language to a programmer. In this implementation a programmer is expected to know quite deep knowledge of MPI. This comes from the fact that the library is not creating MPI processes topology by itself instead it gives this task to a programmer. In eSkel it is a programmer who has to put certain range of MPI processes in each specific

skeleton node. This decision will then be considered by the library to execute the skeleton framework. For example if a programmer want to instantiate a 3 stage pipeline where the second stage compose a farm skeleton of "nw" workers. A programmer is expected to assign one MPI process in the first stage, again one MPI processes to the last stage, and "nw+2" MPI processes in the second stage. We found this implementation is difficult to use for a programmer and also error prone.

In this implementation, we work hard to abstract this fact from a programmers deal. An abstraction of almost 100% is achieved. This basically means that programmers are not required to know MPI.

The creation of MPI process topology, creation of communicators when necessary and partitioning of the MPI process over skeleton node is all handled by the skeleton implementation.
The library will accept only parametric value from a user. Those parameters are then further analyzed to create a suitable skeleton topology. This includes coloring each node of the skeleton, putting enough number of MPI processes per node, creating a communicator per grouped node, translating of process ranks when added to new communicator, giving special rank for some nodes in the skeleton. For example, Farm-Emitter, Farm-collector, an external streamer generator and external stream consumer nodes need to have special rank. In fact for farm-Collector and farm-Emitter this rank can change depending on the skeleton type instantiated. They will have different special rank when a farm alone is executed and when farm is nested in pipeline stage for example. While the stream generator and stream collector will have always common special ranks.
The library will also take care of any nesting requested by programmer while creating the topology. The parent skeleton P, that is nesting another skeleton S, will offload the task of creating the topology for nodes of S to be done by skeleton S "helper class". This helper class will accept the number of MPI processes used by P until now, and the color given to skeleton S by P. Skeleton S will then create the topology for its nodes and send back the number of MPI processes used after last call, and color of each process identified by skeleton S. This information will then be used by P to create a communicator for each node (including the nested node). The Communicator handle will be a useful information for S later. Skeleton S will perform a correct MPI communication only if it posts MPI operations in the correct MPI communicator. This valuable information will be communicated to skeleton S while calling it for execution start.
The parent skeleton P will also handle the instantiation and starting of skeleton S. While instantiation phase means to create skeleton topology (coloring and setting up proper communicator), starting of the skeleton means to call a function that will start MPI operation for communication and that computes task given to that particular node.
The way skeleton topology is created also makes it easier to report which node is started, which node is executing and which one shuts down.
This last issue is handled by the next architectural layer.

3.3 Framework Manager

The other layer is the *"Framework manager"*. This layer will handle the instantiation, start and shutdown of the skeleton framework starting from the already setup topology and compositions. Once the framework is setup, it is easy to make the stream item flow in it, where stream item can carry different type of data. It can for example be a time-stamp data, a shutdown notification data or a normal data. This layer is responsible to generate those different data at the correct time and in a proper order.
The following subsection will further elaborate this.

3.3.1 Stream Generator and Consumer

Stream parallel skeleton patterns work on an item of a stream. For this reason there is a need to convert the problem set to a stream. In this implementation stream generator is external to the skeleton framework construct. For this reason it is made to be reused for any skeleton that is plugged in to the framework.

An MPI process that is chosen to be stream generator will post an MPI send operation as fast as possible. The speed of the streamer is very much determined by how fast the first stage able to consume items and give to the next stage, etc. this goes on transitively until we reach a bottleneck stage (if any).
A bottleneck stage will hit performance as it does not accept incoming items in the required rate and also does not produce items at satisfactory rate. A bottleneck stage is a pipeline stage characterized by a longer task computation than the other stages. Such stage has to be turned to a parallel counterpart by using skeleton composition.
The way a composition is achieved makes the stream generator to function at the maximum rate as much as possible. In this implementation composition is achieved by calling the skeleton nodes (thus MPI prcesses) from left to right. For example, in pipeline patter this means that first the MPI process chosen for external stream consumer task will be started, then the process chosen to be the last stage of a pipeline will be started, and then the second last, etc. all the way to a process chosen for external stream generator task. See Section 4.2 for detail on this topic.

At this point a stream generator will see an already created skeleton topology. Because, all MPI processes are ready by posting the correct MPI primitive used for communication.

A programmer will pass to the skeleton framework a pointer to an input-buffer holding an input item, followed by the size of a single item that will define part of item of a stream. The skeleton stream generator will then generate a stream by drawing an item (thus reading specified amount of byte size) at a time and send it to the first node[1] of the skeleton framework. In this implementation the generator will repeatedly post MPI_Send operation. Those operation will need to be matched with the corresponding receive operation (MPI_Recv) of the first stage of the skeleton framework.
We can see that if any bottleneck node exists in the skeleton it will transitively impose the

[1]First node in this implementation can be a pipeline stage or a farm-Emitter

generation of stream to be slower. Such node has to be parallelized by further nesting a suitable skeleton framework. Otherwise this will affect the performance of the overall skeleton.

On the other hand stream consumer is an external (fictitious node) that collects the final output items and put into the specified output buffer. Both the stream generator and consumer are external to the skeleton framework this support easy composition of skeletons while using the same stream generate and consumer.
In relation to the stream generator and stream consumer is the three main phases of pipeline parallel paradigm: a fill-in transient phase, a steady state phase and emptying transient phase. It is important to consider number of items to be computed, the intense degree of the function that has to be executed at each stage and the time needed to finish the computation. All this considerations can be added up to increase the steady state phase of the computation. Fill-in transient phase and emptying transient phase can't be avoided in general.
An interesting optimization is then to look for a way to not impose the existence of those two phases without the steady state phase. There are particular cases where this can happen for example if the number of input items is smaller than the number of pipeline stages, or if the stream source feeds the skeleton in a very slow speed that there is sensible gap between items of a stream.

3.3.2 Measuring The Overall Completion Time

Measuring over all completion time is another issue in parallel programming. If it is not done properly it may lead to erroneous result. The erroneous factor arise from the fact that each parallel node run in parallel and there is no guarantee that they are all participate in a balanced computation time. For this reason it is not easy to get the global completion time of parallel executing nodes. In this work we measure completion time by considering the time stamp when the first item is sent from the stream generator all the way to the time the last item is received at the stream consumer/collector. This will for sure give us the exhaustive and correct completion time as it is measured for all items that cross the skeleton framework. At each execution the skeleton framework will inform a programmer the elapsed time to execute the given application program.

3.3.3 Skeletons Framework Shutdown

The skeleton framework will execute for the duration of time until which an input item is available in the receive channel. That is until a special type of TAG is received. This tag is generated by stream generator at the end of a stream generation processes. A SHUTDOWN_TAG is chosen to be the first TAG that is supported by MPI. All items received with any other tag other than the SHUTDOWN_TAG will not cause the skeleton framework to shutdown. Each node of the skeleton will decide to shutdown and send a shutdown tag to the next node whenever enough SHUTDOWN_TAGs are received. Some nodes like farm-collector need to receive some number of SHUTDOWN_TAGs to be able to shutdown the skeleton properly. Farm-collector will count "nw" SHUTDOWN_TAGs; where "nw" is number of farm-workers. This ensures that all workers shutdown properly and it is safe to

signal a SHUTDOWN_TAG to the reset of skeleton framework. The requirement towards this is that Farm-Emitter has to send a SHUTDOWN_TAG to all the workers of a Farm to shut them down after finishing the computation of what they have at hand. All other nodes of a skeleton framework both in pipeline and in Farm will need to receive the shutdown tag, send the shutdown tag to next node, and stop executing. In this way the overall skeleton framework will shutdown properly.

3.4 Communication and synchronization

The last layer is the *"communication and synchronization"* layer. It is a low level layer that is abstracted from a programmer as the above layers do. In this layer communication of skeleton nodes, passing the input item to the programmer defined function, accepting what is returned from the programmer function, ordering an out of order items and applying different actions depending on the signal passed from the above layer is handled. For example some data are not supposed to be passed to the user-defined function instead they only meant to convey non-functional information. In this layer the framework also handles the communication and computation overlap by using different MPI primitives.

3.4.1 Communication and Computation Overlap

Communication and computation overlap is very important factor in achieving better completion time. We don't want to pay the communication time entirely. Communication time is a time taken to communicate stream item between different nodes of the skeleton framework. In this implementation the communication is between MPI processes; thus message-passing mechanism is used. A process communicate with other process by posting MPI send primitive which has to be matched with MPI receive posted by the receiving end. Messages are drawn from sender buffer and sent over a channel which itself may be exposed to buffering (depending on the amount of data-packet to be transferred and the MPI operation used). The messages will then be put in receive buffer where the receiver can read the corresponding message and use it as application data. At this point configuration of the MPI implementation used is very crucial; in this case open-MPI-1.6.3 is configured to be used in shared-memory architecture. Leaving it with default one imposes unnecessary communication overhead.

To be able to implement communication and computation overlap we have used non-blocking MPI primitive operation. Non blocking operations help to go ahead with other computation while the communication is on progress. What MPI imposes on non-blocking operation is that the buffer pointed by the operation cannot be used until the operation really completes. However other computation can still be done in parallel with the communication. This helps us to do a useful task while the communication is in progress.
The following figure depicts four possible cases of Communication and computation overlap.

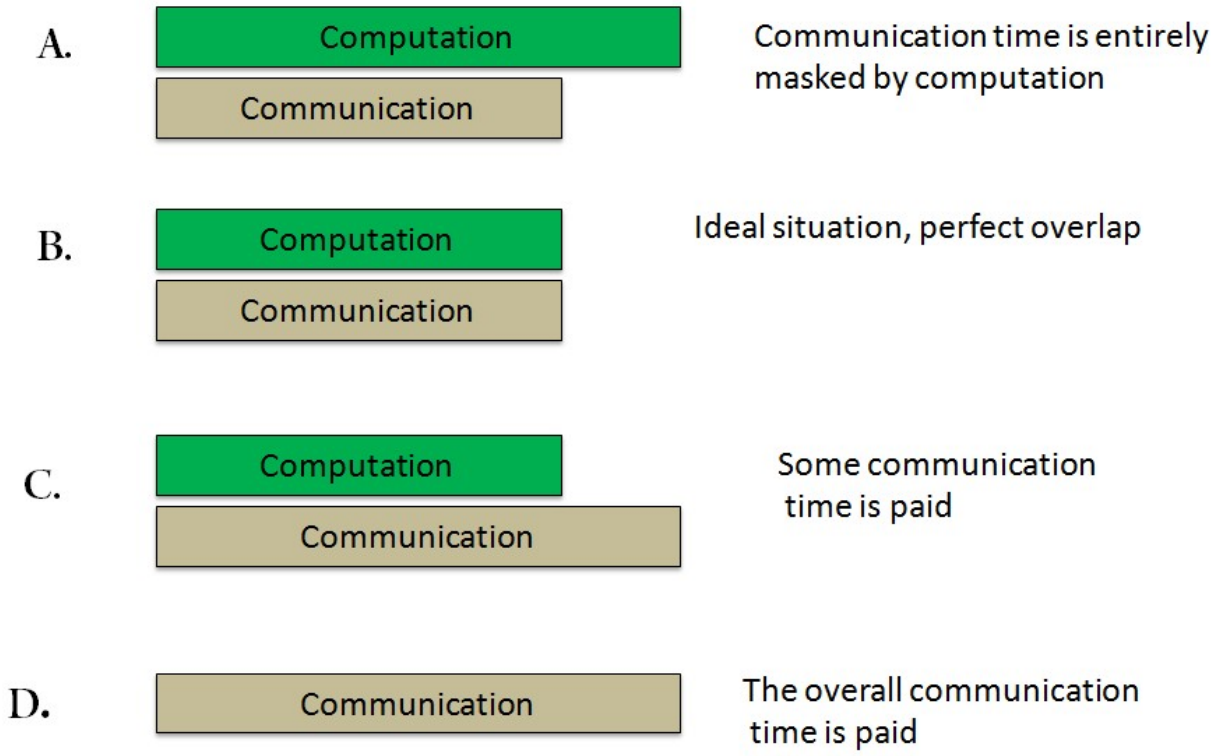

Figure 3.2: Possible Communication computation overlap

In this figure case A) show a situation in which the communication time is entire masked by the computation. This case happens when we have a good computation that can be done at communication time and proceed even after the communication ends. In contrary if the communication is lighter it can also be masked by the computation.
Case B) is an ideal situation which can happen if a programmer carefully chooses an operation that will execute while certain amount of data-byte are in-flight. Case C) happens when communication takes longer time and we don't have much computation to overlap. In any case remember that in this implementation we will do all our computation while the send communication is in progress unless we are computing first item in the stream; or if the computation takes longer time than communication (see case A) above. Case D) can't happen in this implementation. There is no way that the entire communication time is paid. This happens only if a blocking communication version of this implementation is used. We don't recommend the use of blocking communication unless some application requirements enforce it.

The following show a simple pseudo code of non-blocking operation construct.

Algorithm 1 Non-blocking MPI Send Operation

1: $MPI\_Isend(....\&request)$;
2: ▷ Do useful work, (for example receive the next item, and perform a computation on it)
3: ▷ Now, Before next send is started check if the send operation completes successfully
4: $MPI\_wait(..., \&request)$; ▷ this operation will complete the send operation

We can see that most of the tasks are done while the send communication is in progress. So we expect that almost all the communication time are masked by useful work.

3.4.2 Blocking and Non-blocking Operation Of MPI Primitives

MPI supports both blocking and non blocking operation. MPI Blocking operation does not allow the process to continue working until the blocking operation issued is completed. Such process cannot also reuse the resources pointed by the blocking operation until itself completed. Blocking operation should only be used if the requirement is to strictly ensure that the communication must complete before proceeding to the next operation. With a blocking operation it is not possible (or at least not trivial) to implement a communication and computation overlap.
Non-blocking operation on the other hand is an interesting operation that is supported by MPI specification. Processes issuing non-blocking operation can still progress computing some useful work. The only restriction made by MPI specification is that such process are not allowed to reuse resources pointed by the non-blocking operation. Non-blocking operation is a perfect much for our requirement in the implementation of the skeleton framework. We want to start a communication between nodes of a skeleton and perform some other useful task while the communication is on progress. There is no need to re-use the resource pointed by the MPI_Isend operation. In fact we want to make sure that the previous send operation successfully completes before issuing the next send operation. So what is the deal of Non-blocking operation here? Well, while the send operation is in progress we can do useful works; receive next item, perform a computation on it and may be also other book keeping operations (for example determine the next farm worker in round-robin scheduling). The goal that can be achieved here is that all the computation will mask all/some part of the communication. The computation that perfectly masks the communication is ideal one. In any way we will not pay the entire communication time. As a consequence, the time it take for a generic item to cross the pipeline like pattern will be:

$$\sum_{i=1}^{n} max(node_i Comm\_Time, node_i Calc\_Time).$$

where i runs from 1 to number of nodes in the skeleton.
$node_i Comm\_Time$, is a time needed to communicate a message from $node_i$ to node $node_{i+1}$, and $node_i Calc\_Time$ is a time needed to compute a task at $node_i$

We have implemented the framework using both operations. We strongly recommend the use of Non-blocking version unless there is specific requirement towards the use of blocking operations.

3.4.3 Item Ordering In Stream Parallel Computation

Item ordering in stream parallel pattern is another issue to address. Items may get out of order in a point where they can be processed (in parallel) independently of each other. Farm parallel paradigm is one of the known examples where items can get out of order. A Farm-worker[i] execute a task on stream item independently of what the other Farm-worker [i+1] is executing. Even though a round-robin scheduling strategy is used; there is no guarantee that worker[i] will finish the task computation before worker[i+1] or before other workers in the pool. So item ordering for farm parallel paradigm is important. Items cannot get out of order in pipeline parallel pattern. This however is true only if all pipeline stages are sequential (or at least doesn't nest/compose any skeleton that can impose out of order problem).
If it has to be used, we can see the relevance of item ordering both at a farm collector and/or at the end of a pipeline stage. In this implementation a programmer is provided with an enum type definition shown below, from which type of item ordering of a skeleton is chosen.

`Skeleton type enum`

$typedefenum\{FARM\_Odr, FARM\_Non\_Odr\}farm\_type;$
$typedefenum\{PIPE\_Odr, PIPE\_Non\_Odr\}pipe\_type;$

The implementation of item ordering it optimized because of two main reasons.
First, thanks to the support of MPI_TAG there is no need to explicitly create data structure to handle item tag. Second, the algorithm that orders item make use of local buffer at the ordering node only. The size of the ordering buffer is made to be a constant size that can be read from a header file of the library. This is important because one can re-size this buffer making it to handle more out of order items.
If more items get out order than expected it means that the local buffer is already full holding the enough out of order items. All items that arrive after this event will be ignored and sent out to the next node on the fly. This event will be reported to the programmer by printing a warning message. The message goes like this: "Warning: more items are getting out of order, the library is not ordering them any more". One solution to this problem is to increase the local buffer at the ordering node.
Obviously, this is a tread-off. The trade-off is between ordering all items as much as possible versus not imposing performance hit on the whole skeleton execution time.
Item ordering node will impose delay to the whole skeleton structure. Consider that an out of order item arrives; in this case the ordering algorithm will store it in the local buffer. This action makes the communication and computation overlap to be missed and instead it serializes the communication of out of order items to be used later in time.

In fact the algorithm will do its best to make use of such time slot. When an item arrived weather in-order or out of order, the algorithm will look for any items that can fill the gap (i,e., an item with the missing tag). As long as there are items in buffer that can be sent in order the algorithm will stream them out to the next node of the skeleton topology. This is a twofold; first it allows making use of the current time slot in the execution. Second, it frees more slots in the buffer so that items that may get out of order later in time will have a place to be buffered.

3.4.4 Generic Function Signature

Stream parallel computation requires that a function that will be used to compute a task should be a pure function. This function has to be defined by a programmer and should be known prior to skeleton framework execution. In doing so programmers will both be able specify functionality they want, thus supporting generics as long as the computation is suitable for stream parallel computation and also be aware of the input and output type (in fact the byte size of item in this implementation).
Unfortunately the signature of a function that has to be passed as parameter is constant and well defined by the library implementation. Basically all functions have to be defined as follows:

`Generic Function Signature`

$$Void * functionName(Void*)$$

Twofold reasons behind this requirement. First, using this signature any type of function can be defined. This means that any combination of data-type of both the parameter and the return type of a function is supported. It is also possible to pass a variable amount of data-value as long as it is defined in terms of byte, and be able to read from a contiguous memory address. Second, the distribution library used for this implementation, MPI, requires communicating data in byte between processes. Message passing between different MPI processes is done by reading a byte specified by the send buffer and posting on a channel whereas the receiving end process read byte from a channel and put it on a buffer specified by receive buffer.

What a programmer has to be aware of is the amount data-byte that a function accepts and produces. In each function a programmer is allowed to cast the void* type to any type required to fulfill the computation requirement. The return statement of the function can also be a pointer of any data type. In C language any data type pointer can be assigned to void pointer without an explicit casting. The casting is done by the C-compiler itself, it is implicit.

In case stream parallel programming pattern such as pipeline is used, the definition of input and output size of a function has be made; so that the first function output size will

be compatible with the input size of the next[2] function.

[2]The function that is computed by a consumer node in a pipeline pattern

Chapter 4

Implementation

In this section we will present implementation issues of the skeleton framework. Design decision presented in chapter 3 will be referred from here to present their implementation details. Algorithms implemented for this work will also be presented along with the algorithm logic and pseudo codes.

4.1 Round-robin Scheduling Algorithm

Round-robin scheduling is a scheduling mechanism used by farm skeleton in which farm-workers are scheduled to execute a task in a circular mode (i,e., wrapping-around when the last worker in the workers pool is scheduled). The scheduling is done by a farm-Emitter. Farm-Emitter will be given a pool of workers Id; to which it will schedule tasks in a round robin manner. The following is a simple and straight forward pseudo code showing a round-robin algorithm.

Algorithm 2 Round-robin Scheduling

1: *Return* $(nex\_worker++)\ \%\ nw$; ▷ nw is number of workers

4.2 Skeleton Composition Implementation

Skeleton composition is one of the important futures of structured skeleton programming. With composition it will be easier to tune the skeleton framework for a particular application. As presented in section 2.7, "RISC" skeletons are better with compared to "CISC" skeletons. The "RISC" mechanism is better for the following important reasons: first it is compose-able (that is nesting is possible between supported skeletons) and second there are only few common skeleton patterns supported. This results in the choice of the right skeleton pattern without confusion. We stick to the "RISC" skeleton implementation mechanism.

This section presents how skeleton composition is implemented and how the different type of skeleton composition possibilities are supported.
Composition of skeletons is achieved by calling the skeleton nodes from left to right. "Left to right" is in terms of their execution order. The MPI process that triggers the communication primitive is the right most one. In this implementation it is the stream generator. The stream generator process will start the MPI communication primitive and all other processes will start after an an MPI send operation from the stream generator is matched.
Which means in pipeline pattern first, the MPI process chosen for external stream consumer task will be started, then the process chosen to be the last stage of a pipeline will be started, and then the second last, etc. all the way to a process chosen for external stream generator task.
In farm pattern similarly, first, the MPI process chosen for external stream consumer task will be started, followed by farm collector, farm workers, farm emitter and finally Stream generator process will be started. The following figure depicts this process for pipeline pattern. In this figure each stage of a pipeline is shown as a generic stage. However we can think of those stage as a sequential or parallel stages.

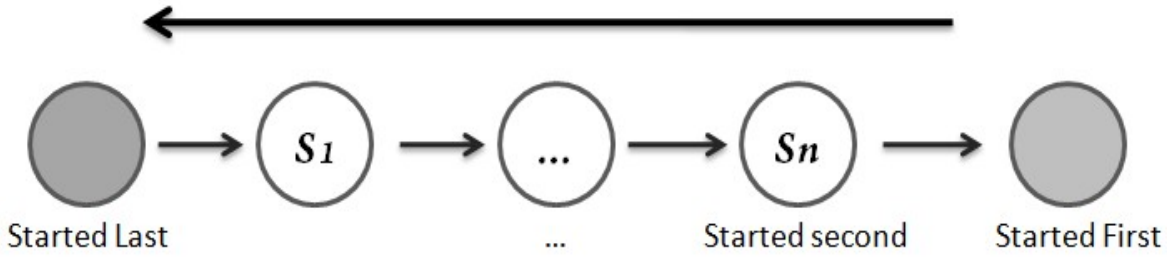

Figure 4.1: "Right to left" way of composing pipeline skeleton

For each skeleton that has to be nested in a given node, the "left to right" way of nesting rule will still apply.

This way of composing skeleton patterns has advantage for the following two reasons: first, the the communication channels will be named while composition is performed. Second, at the point the stream generator starts it will have an already created skeleton nodes thus it can post the communication primitives as fast as possible. Please refer to section 3.3.1 on how stream generator make use of this advantage.

Different Skeleton Pattern Compositions

In this section we will present a composition of different skeleton patterns. Basically we will address the implementation behind the composition of a farm skeleton in pipeline stage and composition of pipeline skeleton in farm workers.

The first possible composition is the composition of farm skeleton in a pipeline stage. This composition is the usual and important form of composition. Specially, it is a way of eliminating a bottleneck from a pipeline skeleton. The slowest stage of a pipeline can be accelerated by nesting a farm skeleton at that stage, thus balancing the pipeline stages. Nesting a farm skeleton in an already balanced pipeline stages will not be beneficiary. So, the choice of the right form of composition has to be made by a programmer in accordance with the application that has to be parallelized.
The following Figure shows such a composition. Assuming that the stage S_{x+1} is a bottleneck with respect to the other stage of the pipeline, a composition can be depicted as follows.

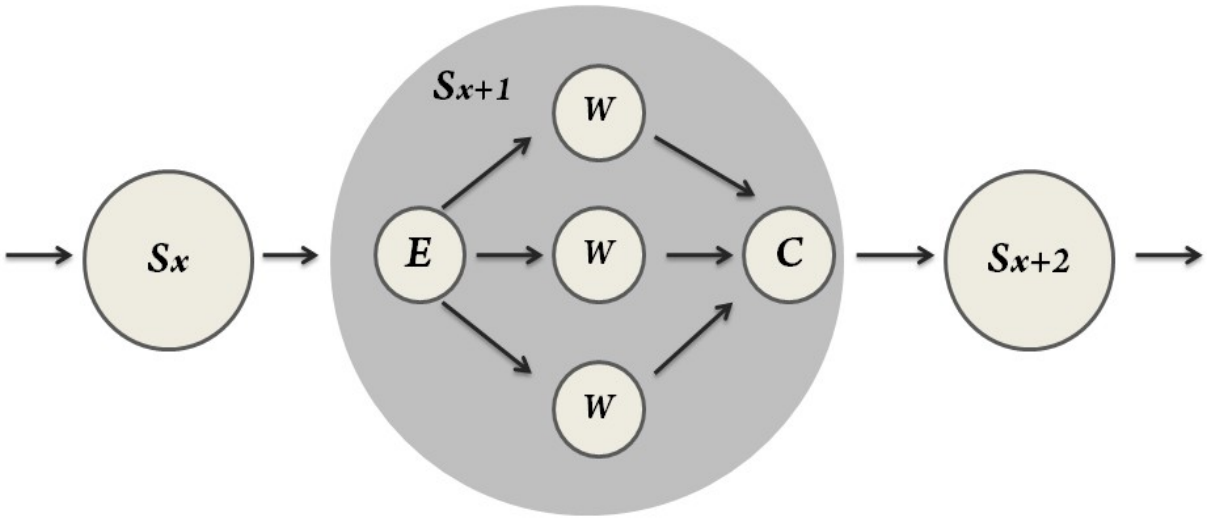

Figure 4.2: A Pipeline skeleton where the stage S_{x+1} is a farm

The composition of farm in a pipeline stage may be used without any restriction. This does not mean that it is always beneficiary to nest a farm in a given pipeline-stage, it just means that it is possible to nest a farm in any stage of the pipeline skeleton (as long as its need is confirmed by a design decision). For example all stages can be turned to be a farm, or only some of them.
In what follows we will present the implementation behind this composition. In this implementation the data structure to represent the skeleton composition is very important towards implementing a modular skeleton composition. Both farm and pipeline skeletons have their own data structure that makes them modular. Both have a list of properties that are partially mandatory and have to be passed by a programmer and other properties that will be filled by the library at run time. The important thing to mention about the data structure is how the pipeline stage and a farm-worker structure are defined to handle a skeleton nesting (composition) issues. Those are important nodes that are candidate for skeleton composition. The composition is supported by defining the data structure of a node to be either a sequential or of a parallel node.
If a node is a sequential node, among the mandatory parameter that has to be passed by the programmer the important one is the function pointer that a node will use to compute a task. With just a pointer to a function that is defined by a programmer it is easy to create

the sequential node.
A parallel node on the other hand is more complex node. Instead of directly accepting a function pointer it will accept a skeleton structure. The skeleton nesting structure has to be filled properly by a programmer. As presented in chapter 3, a programmer is provided with an interface to the skeleton framework. This interface will allow a programmer to pass parameters that may be used to choose between different compositions.

4.3 A Pipeline Stage Implementation

A pipeline stage can be a sequential stage or a parallel stage (it can be a farm skeleton in this particular implementation). For this the data structure of a pipeline-stage is design to have the following form (a pseudo code is presented here).

pipeline stage structure

```
1: TYPEDEF STRUCT (pstage_t) :
2: int in_byte_size;                          ▷ Number of input and output item in byte
3: int out_byte_size;
4: void * (*f_ptr)(void*);                    ▷ function Pointer
5: int in_channel;
6: int out_channel;
7: END STRUCT
8:                                            ▷ A Stage can be a sequential stage or a Farm
9: TYPEDEF UNION (stage) :
10: pstage_t pipe;
11: farm_t *farm;
12: END UNION
```

With this data structure it a decision of a programmer to define a stage to be a sequential or a farm. Since the definition of a pipeline stage is done separate structure, it is possible to turn any of those stages to be a farm. The above figure 4.2 shows a simple composition of a farm skeleton in stage S_{x+1}.

4.4 A Farm Worker Implementation

A farm worker is another parallelism candidate in skeleton framework. A farm-worker can be sequential or it can further nest a pipeline structure. The following pseudo code shows a data structure defined to handle this case.

Farm worker structure

1: ▷ A sequential Worker
2: $TYPEDEF\ STRUCT\ (seq\_worker)$:
3: $void*\ (*task)(void*)$; ▷ function pointer
4: $END\ STRUCT$
5: ▷ A generic farm worker
6: $TYPEDEF\ \ UNION\ (generic\_worker)$:
7: $pipe\_t\ p\_worker$;
8: $seq\_worker\ f\_worker$;
9: $END\ UNION$

Filling the "generic_worker" with the correct parameter a farm-worker can be made to be sequential or a parallel node.
The following figure 4.3 depicts what a composition of pipeline skeleton in a farm-worker looks like.
In general a farm-worker executes a replicated function; this is also true when a farm worker is nested to be a parallel node. A programmer has to define the structure of a skeleton to be nested only once. The library will replicate this structure number-of-workers times ("nw" times) to populate all farm-workers.

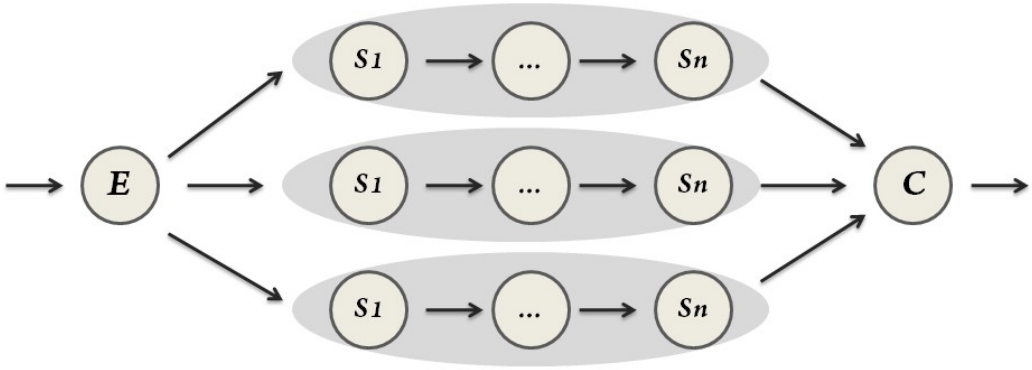

Figure 4.3: A Farm skeleton where its workers are a parallel nodes that execute in a pipeline pattern.

Further Deep Nesting

Nesting can even be made to be a deep nesting. This means that the skeleton that is nested in a given node can itself nest another skeleton. In the following figure 4.4, we show the case where a pipeline skeleton nests a farm skeleton in one of its stage (let say stage S_i). This is a typical and most used composition pattern. And then we further nest the farmed stage S_i to be pipelined pattern worker.

Deep nesting is achieved in a similar way. Each time we take an action transitively according to the mechanism presented previous chapter. The parent skeleton P will be the higher point in the composition hierarchy; the nested skeleton S, will communicate with Skeleton P to form its topology and the new information itself produced. In a similar way any skeleton K, that going to be nested in skeleton S, will communicate with S to form its own proper skeleton topology.
Obviously, this mechanism will need a support from the programmer. Specifically a programmer has to define a given node to be a deep nested node by picking the right skeleton composition and by properly passing parameters expected/needed for such a composition.

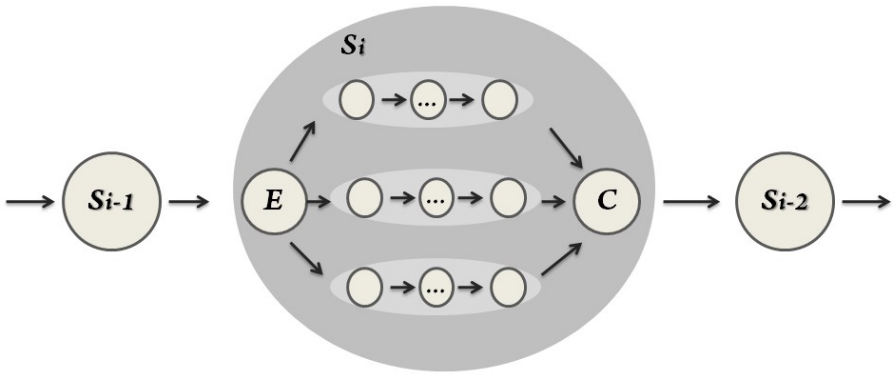

Figure 4.4: Deep nesting: A pipeline skeleton where stage S_i is a farm (which itself nests a pipeline skeleton in its workers)

4.5 Item Ordering Implementation

Item ordering algorithm is a little bit tricky algorithm. It is also an inefficient operation that has to be avoided if possible. The in-efficiency is simplified by this implementation as much as possible.
This algorithm will be used based on the parametric preference of a programmer. A programmer is given to choose between ordering or nonordering skeleton framework.

Item ordering is used either at farm collector or at the last stage of a pipeline skeleton. In fact in a pipeline where all its stages are sequential, it is not possible to have an out of order item, so no need of choosing ordering pipeline skeleton in this case. However in the cases where other skeletons such as farm are nested in any of those pipeline stages an out of order item may occur. In most application item ordering is not a strict requirement or at least it can be offloaded to be handled by the final destination. For example application such as image processing need not impose restriction on the order the input images are finally produced, instead the order an operations are applied to a single image is a restrict requirement (which implicitly handled by the nature/topology of skeleton framework). The output images can be written to a file in any order, this will be enough without the use of item ordering node. Or item ordering task can also be given to the final destination that may

put the output items in an output buffer accessing the buffer index by the value indicated as tag of the output items.
In what follows we will explain how the item ordering is implemented and how it hits performance.
The following figure shows the conceptual depiction of the algorithm. This figure shows the case when item with tag 2 arrives before item with tag 1 and then buffered in the data-buffer. When item with tag 1 arrives it will be used on fly as it will fill the missing gap, this will also trigger that item with tag 2 can now be sent in order. The algorithm is further explained in the following paragraphs.

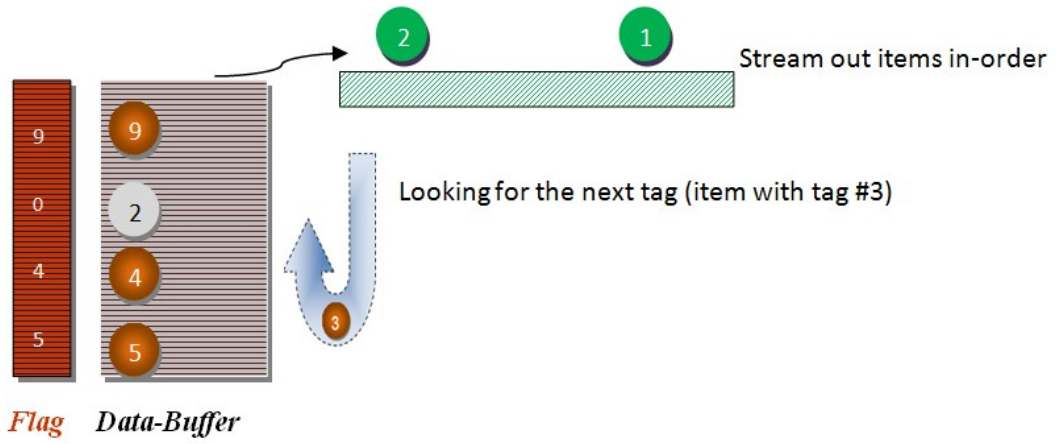

Figure 4.5: Ordering algorithm data-structures. The item ordering algorithm consist of a data buffer to hold items that get out of order during computation and a flag buffer registering an item tag-value or a zero(0) for each slot of the data buffer.

Choice Of The Data Structure

The choice of the above data structure is strongly related to optimization of the algorithm and also to the number of out of order items that the ordering algorithm has to handle. As presented previously in chapter 3, the algorithm will only order some constant number of out of order items. This basically means that the data structures have a finite constant size. When more than expected number of item gets out of order the data-buffer will be full and the flag data structure will have "all-non-zero" values. In such case the library cannot any more take care of the out of order items. So it will leave them un-ordered and stream the items in the order they arrived in. This event will be informed to a programmer as a warning message.

4.5.1 Design Of The Algorithm

The algorithm is designed to be as efficient as possible. First, the algorithm will check if the buffer has currently any buffered item; and it will check if any of those buffered items can fill

the missing tag gap. If so it will stream them out *in-order* until the next item in order is not found in the data buffer. As shown in the above figure this last action is very important in preparing a free buffer slot for next item that may arrive out of order. The figure shows how the slot allocated for item with tag 2 is freed. A free slot is a data-buffer memory space that have never been used by the algorithm or that have been used but marked to be free later. The pseudo code of the algorithm is shown as follows:

Algorithm 3 Item ordering

```
1:  current_item ← INITIA_TAG
2:  while not SHUTDOWN_TAG do
3:      if in-order item is received then
4:          Stream out the item directly;
5:      else
6:          if (an item with the missing gap is in the buffer) then
7:              Stream the item from buffer;
8:              Mark the buffer slot free;
9:              Increment the current tag value;
10:         else if (Buffer is not full) then
11:             Buffer the out of order item in the data buffer;
12:             Mark that slot used;
13:         else
14:             Print warning message;
15:             Deactivate the ordering algorithm;
16:             stream-out all items in the buffer regardless of their Order;
17:         ▷ Items that arrive after this event will be streamed out in the order they arrive;
18:         end if
19:     end if
20: end while
21: if some items are still in buffer then
22:     if item ordering is still active then
23:         Stream item in order from the buffer until the buffer is empty;
24:     else
25:         Stream out item from buffer regardless of their order;
26:     end if
27: end if
```

This implementation is made optimal in a sense that first, line 8 of the above pseudo code shows that the algorithm will free the buffer as much as possible, second, line 7 shows that

all items that can be sent in order will be streamed out as long as they are available in the buffer or in the input channel. Third, the algorithm is made parametric in that it can be used in any node where item ordering is used.

The algorithm will be started whenever an item ordering skeleton is selected. The algorithm accepts the following important parameter that makes it generic to be used at any feasible node of the skeleton. Feasible in sense that it should be a proper node to execute the ordering algorithm, for example even though it is possible to use the item ordering at a farm-Emitter, it doesn't make sense to do so as the potential out-order point is at farm worker.

`parameters for Item ordering node`

INITIAL_TAG , *MAX_BUF_SIZE*, *Source process ID* , *destination process ID*, *input item byte Size*, *output item byte Size*, *MPI Communicator*

INITIAL_TAG, and MAX_BUF_SIZE are constant values given in header file of the library. It can be changed to support different buffer size either to handle the case where more items have to be ordered (i,e., by increasing the MAX_BUF_SIZE), or to reduce the over-head of the ordering node (i.e., by decreasing the MAX_BUF_SIZE). INITIAL_TAG is used to indicate the starting tag number of the first item. This can be changed as this work can be expanded more to include other futures. Tagging an item is supported by MPI itself, items will be tagged by MPI_TAG, which basically start from zero. However, we use some special tags to indicate different signal arrival at execution time of the skeleton framework. For example tag number zero (0) is used to indicate a skeleton SHUTDOWN signal, tag number one (1) is used to indicate the time-stamp used to measure over all time elapsed by the skeleton to finish the computation. Other need may still come into play, which will need to take the tag number 2, 3, etc. For this reason it is good practice to put the INITIAL_TAG to be a parametric.
One limitation of MPI_TAG is that it has limited upper bound. MPI specification puts that the minimum bound that has to be supported by MPI implementers is 32767. On the other hand, the valid upper bound that implementers can use is specified to be MPI_UB $2^{32} - 1$ which is quite a big numbers. Most MPI implementations, including open MPI tends to support a tag number very close to the upper bound.
The item ordering algorithm overcomes the problem of fixed Upper bound limit by using the following mechanism, under one specific condition though. Assuming that the item ordering buffer size can't be made close to $2^{32} - 1$, the algorithm will wrap-around and start reusing the item tags starting from INITIAL_TAG once again. This assumption is feasible in sense that first, an application that requires a buffer size of close to 2^{32} should avoid the use of item ordering otherwise it will impose unnecessarily high performance degradation. Second, it is highly improbable that 2^{32} items can get out of order.
With the above assumption, by the time the algorithm start wrapping-around items with tag close to INITIAL_TAG are already served and streamed out from the buffer. So it is safe to restart the tagging from INITIAL_TAG.

The following pseudo code shows how this is implemented:

Algorithm 4 Wrapping around TAG Value to resolve the MPI_UB limitation

```
1:                                   ▷ In Item Tagging node and also in item ordering node
2: Void * tagmax ← MPI_UB;           ▷ TRY: read open-MPI MPI_UB value
3: if not Succeed then
4:     tag_max_size ← 32767;         ▷ the lower limit that every MPI implementation has to
   support is used
5: else
6:     tag_max_size ← *(int*)tagmax;
7: end if
8: tag ← INITIAL_TAG
9: while item exist in input buffer do
10:                                  ▷ Use tag to send an item.
11:    ++tag;
12:    if tag ≥ tag_max_size then
13:        tag ← INITIAL_TAG         ▷ wrap around
14:    end if
15: end while
16:                                  ▷ The following is in item ordering node only
17: while in-order item arrive or in-order item is obtained from local buffer do
18:    if Next_expected ≥ tag_max_size then
19:        Next_expected ← INITIAL_TAG;
20:    end if
21: end while
```

4.6 Measuring Overall Completion Time

Measuring the overall completion time (Elapsed time) is an important part or parallel programming, as discussed in earlier chapter, it is not simply correct if we try to measure elapsed time by calculating it per parallel executing node and divide the result by the number of nodes. We have no guarantee that those nodes are balanced and take the same (or even closer) amount of time to finish executing. The correct and exhaustive elapsed time is the time from when the first item is sent by stream generator all the way to the time the last item is received by stream consumer. For this a time stamp measured by the stream consumer (when the first item is sent) is recorded to be sent at the end of streaming. The

stream consumer on the other hand is responsible to read the time stamp when the last item is received; which will then calculate the overall elapsed time based on the time stamp itself reads and the time stamp received from the stream generator. This is possible implementation, in a sense that there is no need to synchronize the time the streamer reads and the one that stream consumer reads. Both nodes will run on a single multi-core machine.
We can see that the time measure is a concern of the external stream generator and stream consumer nodes only.
One possible (also usual) implementation of this mechanism is to propagate the time-stamp from the stream generator all the way to stream consumer; where the time-stamp data is tagged by special tag, so that it will not be confused with other data.
As long as this implementation is concerned this is not necessary, and impose in-efficiency in that, time-stamp signal have to cross the overall the skeleton nodes. Instead we can directly communicate the time-stamp from stream-generator to the stream consumer. This is preferable first, for performance reason. Second, since we are dealing with MPI processes, communicating from stream generator to stream consumer will be done as any regular point-to-point communication primitives. This last consideration results a correct way of measuring elapsed time because the initial time stamp is read and recorded early when the first Item is about to sent, then stream consumer will read a time after the very last item (or the shutdown signal) is received.
The following shows a simple pseudo code of this mechanism:

Algorithm 5 Completion time measurement

1: ▷ Stream generator.
2: $begin \leftarrow gettimeofday()$; ▷ Read initial Time stamp
3: **while** item exist in input Buffer **do**
4: $MPI\_Send(item, ...,)$;
5: **end while**
6: $MPI\_Send(Time\_stamp,\ TIME\_STAMP\_TAG)$; ▷ Send the previously registered time stamp
7: ▷ Stream Consumer.
8: **while** tag $\neq$ SHUTDOWN_SIGNAL **do**
9: $MPI\_Recv(item,)$;
10: **end while**
11: $end \leftarrow gettimeofday()$; ▷ Read current time stamp.
12: $MPI\_Recv(Time\_stamp,)$ ▷ Receive initial time stamp from streamer
13: $PRINT\ (diff(end,\ Time\_stamp))$; ▷ Calculate and display Elapsed time

4.7 Non-blocking Operation Implementation

As discussed in section 3.4.2, MPI provides blocking and non-blocking operations. Both implementations are supported by this work. The blocking operation are straight forward, here we focus on the use of non-blocking operation version of this work. The detail implementation is provided by MPI itself, here we will present how we make use of this support of MPI. The fact that "Non-blocking operations allows a program to continue executing, regardless of the state of the operation" is a very important and core point we want to present here. Whenever a non-blocking MPI send primitive is started, all useful tasks that can be done regardless of the state of the send operation will be started in parallel to overlap it with the communication. The common operation candidate to overlap with the communication are receiving the next item, and performing a task on it if necessary. In some nodes other useful task can also be done at this time, in farm-emitter for example, the scheduling algorithm will determine the next worker to be used, and in item-ordering-node, the arranging of the data-buffer and tag-buffer is all done at communication time to overlap or mask the communication time. The following pseudo code shows how this is done. We show for a generic node.

Algorithm 6 Non-blocking operation (Double Buffering)

1: **while** NOT SHUTDOWN_SIGNAL **do** ▷ do computation-communication overlapping operation.
2: $MPI\_Isend(snd\_buffer, ..., \&request)$; ▷ Start Non-blocking send operation
3: $Compute()$ ▷ Now start other useful task to overlap it with computation.
4: $MPI\_Recv(rcv\_buffer, , status)$;
5: $next\_worker \leftarrow getNextworker()$; ▷ In node such as farm
6: $Perform\ item\ ordering\ algorithm\ at\ this\ time$ ▷ In item ordering node
7: $MPI\_Wait(\&request, \&snd\_status)$; ▷ At this point make sure that the send operation completes
8: **end while**

The algorithm shows that communication will always be overlapped (either entirely masked or at least partially) with communication. As presented in chapter 3, there is no way that a communication time is entirely paid. A perfect overlap can result from a good design decision.
We can see that this algorithm uses a *double buffering* mechanism. There are two buffers used to handle non-blocking operation. First, the buffer used to receive incoming items and second, the buffer used to hold an outcome of a computation. Line 2 and 4 of the above algorithm shows this case. Using these two buffers we can do independent tasks in parallel. While the communication is in progress the non-blocking MPI primitive will not allow us to use the *send buffer*; instead we are free to perform important tasks on the received buffer.

4.8 Skeleton Topology Creation

Skeleton topology creation is handled by the library implementation automatically. The implementation starts from the parameters passed by a programmer. The creation of topology makes the implementation optimal and abstracted. The optimality comes from the restriction this implementation imposes on the programmer. For example, as presented previously, restricting the number of MPI processes that have to be used by the programmer when running the application is a good decision that guides a programmer to avoid in-efficiency. In addition to this, when a library make use of the parameters passed from a programmer it group them to the corresponding nodes only. This will make the other nodes to not deal with parameters that does not concern them. For example, think of an input-buffer that is only used by stream-generator. Now, if we copy this buffer to all MPI processes in the pool, we unnecessarily consume more memory space. In a similar way, not only data structures copy but also operations that have to be computed by a given node will be made private to that node. This extremely contributes to the optimality of the implementation.

On the other hand the abstraction achieved is obvious, a programmer need not deal with MPI processes. This is a twofold; first, the usage of the library will be easier. Second, the programmer will not impose inefficiency by bad implementation decision.

Conclusion

In this chapter we discussed implementation details and algorithms of the skeleton framework.
The main topics covered are: implementation of skeleton nesting, we presented how composition is achieved, possible composition supported by this implementation and how the pipeline stage and farm workers are implemented so that a modular and arbitrary nesting of the two skeletons will be possible.
We have also discussed item ordering algorithm that can be used to order an items that may get out of order in a computation like farm pattern. Design issue of this algorithm and its usage is also presented.
The other main topic covered is a non-blocking operation used for communication computation overlap. we provide an algorithm that uses "double buffering" mechanism to make use of non-blocking operation provided by MPI.
In this chapter we also present how skeleton topology is created and how it helps in achieving a good abstraction of the skeleton framework implementation.
last but not least, we also present a mechanism used to measure the overall completion time of the computation carried out by the skeleton framework.

Chapter 5

Experiments

This chapter presents result of experiments using the skeleton framework implementation. We present comparison of different combinations of experimental cases. Each combination is chosen to give good and full information about choices that has been theoretical presented in previous chapters. The Experimentation results, meanings of collected statistics and diagrammatic depiction of those results will be given in the next section of this chapter.
In the first section of this chapter we will present what has been done, what is used and what has been assumed to prepare the test bed for the experiment.

5.1 Environment Used For The Experiment

For these experimentation we have used state-of-the-art multi-core architectures. Two Linux machines where the the first machine is 8 cores machine and the other one is 24 cores machine. The 8 core machine is called "Andromeda" has 2 CPUs each has 4 cores with hardware multithreading support, the 24 core machine is called Titanic has 2 CPUs each has 12 cores and no mechanisms for hardware multithreading is available.

The main future of these machines are summerized as follows.
Andromeda: Intel(R) Xeon(R) CPU E5520 @2.27GHz multi-core machine with 8 cores. All cpus run at 1.6 GHz and have a cache size of 8MB.

Titanic: AMD Opteron(tm) Processor 6176 machine with 24 cores. All cpus run at 2.3GHz and cache size of 512KB .

For the experiment OpenMPI 1.6.3 is used as the MPI library implementation in all cases.

5.2 Methodologies

In this experimentation different experiments have been performed using a simple stream parallel application.
Image processing is one of the interesting applications we have used in this experimentation. We have implemented a pipelined image processing application where each stage adds some effect to the image and pass it to the next stage. The next stage will add another effect, and pass it to a stage next to itself. The output of the application is an image that has been transformed by those pipelined operations. This application uses the imageMagick [14] library. We have used both the C-language API and the C++ language API of this library. This is required to be able to compare this work (which is based on C language) with previous works (a FastFlow skeleton library which is based on a C++ language).
A farm stream parallel pattern may also be used/exploited to implement image processing application. The imageMagick library also has some image transformations that are very expensive to compute. As shown in figure 4.1 operation like "Image Emboss" is expensive operation that may generate bottlenecks behavior if used with other less expensive operation. Farm skeleton can be used to compute such operation either nested in another skeleton or stand alone; and therefore to metigate the effect of bottleneck our applications will be executed on multi-core/many-core machines presented previously. The analysis of the results will be given in the next section of this chapter along with the statistics collected after executing the experimental cases.

Using the imageMagick library we implement the following pipelined image operations.

Pipeline image operation
$Gaussian\ blur -> Emboss -> resize -> flip$

A second synthetic application used for experimentation is a simple mathematical function composition. A function composition such as multiple nested sin() is used. This application is implemented for both farm and pipeline patterns. In pipeline each stage will execute the same code, thus will be a balanced stages. Also in farm, each worker will execute the nested sin() function.

We also give attention to a real world application such as numeric computations. It is the area of computing that researched on for long time and and still needs more attention. It is applicable in a many subjects of our life and technological advancements. For this we used an application kernel called *Gaussian Elimination*. Gaussian elimination is an algorithm for solving systems of linear equations. It can also be used to find the rank of a matrix, to calculate the determinant of a matrix, and to calculate the inverse of an invertible square matrix.
For this experiment we implemented an application that accepts matrix representation of the

problem (i.e., linear equations), and runs a code on it to get the result of Gaussian Elimination (i.e., solution to the linear equation). In this case each input stream item is a matrix, each matrix represents a system of linear equations; an example is presented below:

Linear Equations

$$
\begin{aligned}
2x + \ y - z &= 8 \\
-3x - y + 2z &= -11 \\
-2x + y + 2z &= -3
\end{aligned}
$$

Matrix representation of the above Linear equation

$$
\begin{bmatrix}
2 & 1 & -1 & 8 \\
-3 & -1 & 2 & -11 \\
-2 & 1 & 2 & 3
\end{bmatrix}
$$

The last column of the matrix is a vector/constant part of the linear equation.

The process of Gaussian elimination has two parts. The first part *"forward Elimination"* reduces the given system to either triangular or echelon form, or results in a degenerate equation, indicating the system has no unique solution but may have multiple solutions (rank<order). Forward elimination is computed using elementary row operations.
The second step uses *"back substitution"* to find the solution of the system above.
This tells us that this computation is suitable to be parallelized by using both pipeline and farm patterns.
In farm the "forward elimination" and the "back substitution" operations will be computed by a given worker, a number of workers can be used depending of the stream length and the resource available.
In pipeline pattern, two stages can be used; one to compute the elimination and the other will do back substitution on the eliminated matrix. The first stage of the pipeline will execute an expensive operation than the second one. In-fact the computation of both the "forward elimination" and the "back substitution" will take some more time as they work on matrix than a single element. For this reason we have used a farm skeleton in both stages with different number of worker so that we will have balanced pipeline stages.
Experimentation result of this computation is given in the next section of this chapter.

We have used these different[1] kind of applications to evaluate how both implementations (this work, and Fastflow library) perform towards those application so that we can draw a

[1]Image processing, trigonometric function computation, and numeric computation

good conclusion from the result.

5.3 Tools

We have used different libraries and resources that are used to implement experimental application, and to collect experiment results.
To mention some of the important tools we have used: we used imageMagick [14] library, this library has a rich image transformation operation. It supports different languages interface. We have used the C and C++ interfaces. The library supports operations that are both course grained to compute and operations that are fine grained to compute. We make use of both flavors to experiment the use skeleton composition.

The other resource we have used is a bash scripting language to run the different applications with different parameters and collect experimental result. The script runs the application multiple times. Each time it will pass different parameters and it will append the results of the computation to a file prepared to collect experimental results. The resulting data is collected to have an X, Y dimension so that we can easily plot them later using the tool "gnuplot".

5.4 Experimental Cases

In the next part of this chapter we presented different experimental cases. First we will present the benefit of using the KNEM kernel module by comparing it with an experiment that is done without using KNEM. Then we will focus on the scalability and efficiency of our library by computing the three different applications presented above. Finally we will give comparison of or our library with previous work (Fastflow). This is further explained as follows.

Experiment of mspp and Comparison With Previous Work
In this experimentation we present a comparison of this implementation here after called mspp-1.0.0 (or simply "mspp", to mean MPI Stream Parallel Patterns/Paradigms) with Fastflow [23] skeleton library. The comparison is made to be fair in sense that the same library, application, and computing resources are used. Besides their programming model difference, the only difference they have is that Fastflow is based on C++ and mspp is based on C; however since all the applications we using can be implemented using C and C++, we able to implement the very same application independently of language differences.
Comparison of both farm and pipeline skeleton performances while running different application is experimented.
It is worth noting that even though the two libraries target multi-core architecture, they are

based on different programming model, namely message passing vs shared memory programming models.
Fastflow is a pattern-based programming framework supporting the implementation of streaming applications. It provides pipeline, farm, divide and conquer, and their compositions, as well as generic streaming networks. It is specifically designed to support the development and the seamless porting of existing applications on multi-core. The layered template-based C++ design ensures flexibility and extendibility. Its lock-free/fence-free run-time support minimizes cache invalidation traffic and enforces the development of high-performance (high-throughput, low-latency) scalable applications. It has been proven faster than TBB, OpenMP, and Cilk on several micro-benchmarcks and real-world applications, especially when dealing with fine-grained parallelism and high-throughput applications.
On the other hand mspp is based on MPI, the standard message passing interface. Since this implementation also targets multi-core architecture, we have configured open-MPI to use a shared memory back end. This configuration does not save us from having a duplicate/-multiple copy when passing messages between processes. See section 2.9.2 for detail of how open-MPI shared memory configuration is done and how the Linux kernel module KNEM is used to obtain a zero copy communication.

5.5 Experimental Results

In this section, we will present the experimental results. The results are analyzed from data collected by running different applications presented in the above section.
The collected data is then shown by a diagrammatical representation; this data is also used to drive some important cost model measurements such as efficiency and scalability of the skeleton patterns.

The remaining part of the thesis is arranged as follows: first, we will present the effect of using KNEM, and then we will analyze the scalability and efficiency of mspp, finally we will present comparison of mspp with Fastflow.

5.5.1 Using KNEM vs Without KNEM

Fist we would like to show the effect of using KNEM. As presented in section 2.9.2, KNEM allows to optimizing the communication between MPI processes. The following result shows an experiment that is carried out using KNEM and without KNEM. The later case means that MPI will perform message communication between processes using a double copy. First, from sender buffer to shared memory and then from shared memory to receiver buffer.
The following two figures show the result of executing the *Gaussian elimination* computation using farm pattern. For both figures two curves are depicted one for execution using KNEM and the other without using KNEM.

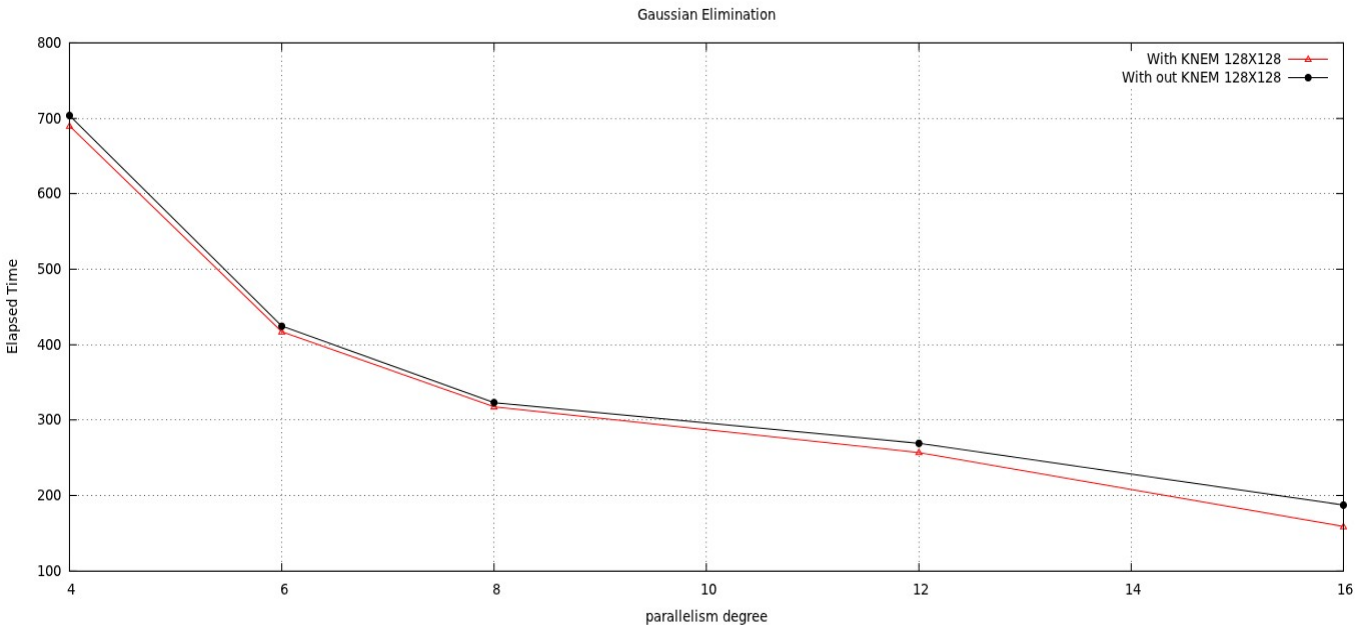

Figure 5.1: Gaussian Elimination of a 128X128 matrix

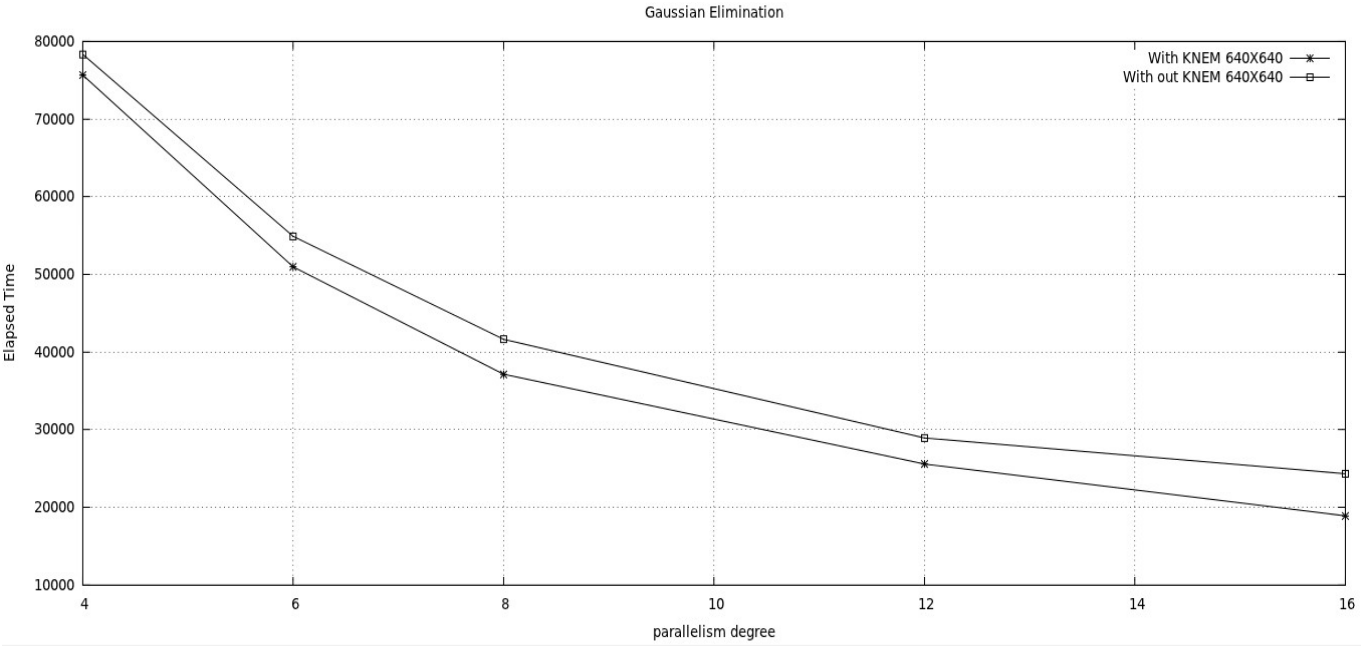

Figure 5.2: Gaussian Elimination of a 640X640 matrix

The above to figures show the effect of using KNEM. It is clear that in both cases a better completion time is achieved by using KNEM. The gap between curves in figure 5.1 is not that much significant. This is because the communication of matrix element of 128X128 can be overlapped with the computation of this matrix. However as the matrix size increases the communication will have significant effect than the computation. Figure 5.2 shows this

effect.
The other important point to notice here is the effect of parallelism degree. As parallelism degree increases the gap between the two curve increases. This comes from the fact that the communication from the worker to a collector is a bottleneck. Each worker has to wait for the time a collector is free to serve them. If KNEM is not used in this case each worker will take much time to communication the matrix result to the collector; thus imposing more latency.

5.5.2 Scalability And Efficiency Of mspp

In this section we present scalability and efficiency of mspp by experimenting on the three applications presented in previous section.

Pipeline Image Processing

Here we will present the result of executing a pipelined image operation. 4 stages of pipeline is used each performing different image transformation on the given input image. In this case a 100X100 image pixel is used.
Both efficiency and scalability o executing of this application is shown:

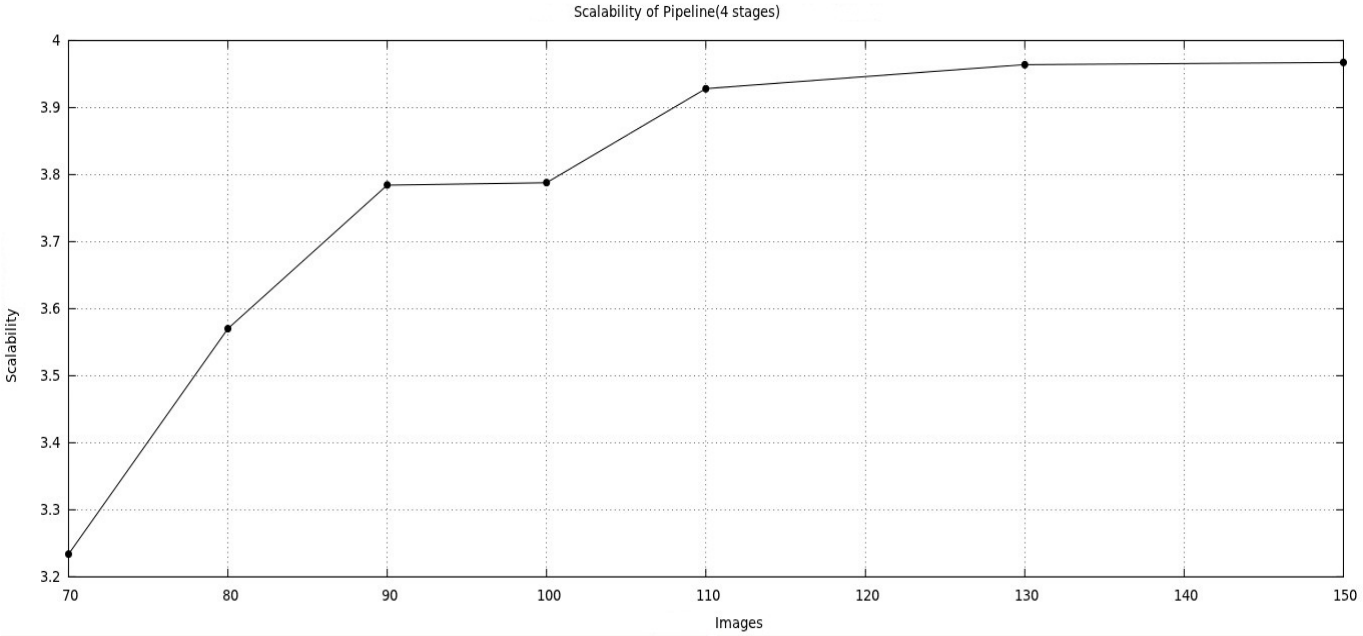

Figure 5.3: Pipeline scalability: Image transformation

The diagram clearly shows that as stream length increases scalability of almost 4 is achieved. For small number of stream length the effect of fill-in transient and emptying transient phases are a bit sensible. As the stream length increases this effect is appear to be

insignificant as presence of steady state will revel.

Similarly the following figure show efficiency of the above application.

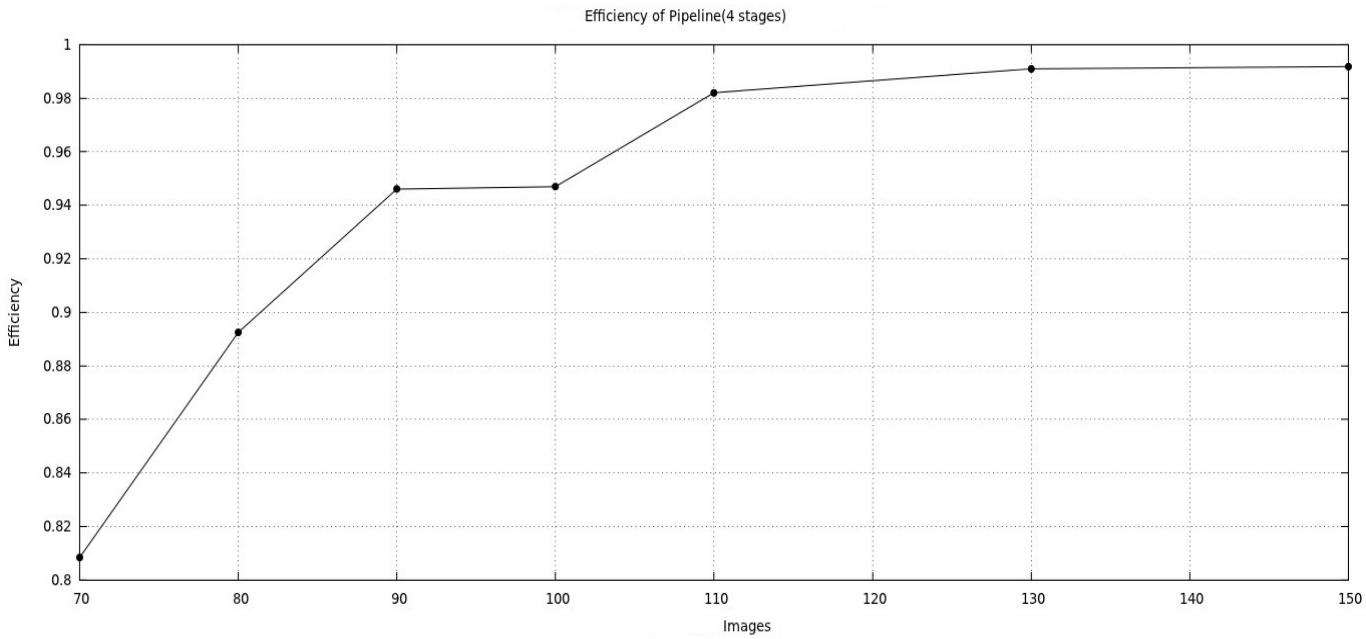

Figure 5.4: Pipeline efficiency: Image transformation

The curve of this result is similar to the above result except that they convey different information. In this result a good efficiency is achieved. Specially, as stream length increases a better and better efficiency is achieved.

Farm Image Embossing

The following figure shows scalability of farm pattern as number of worker increases. This result is for the image embossing application. In this case we have multiple curve, each belonging to different stream length.

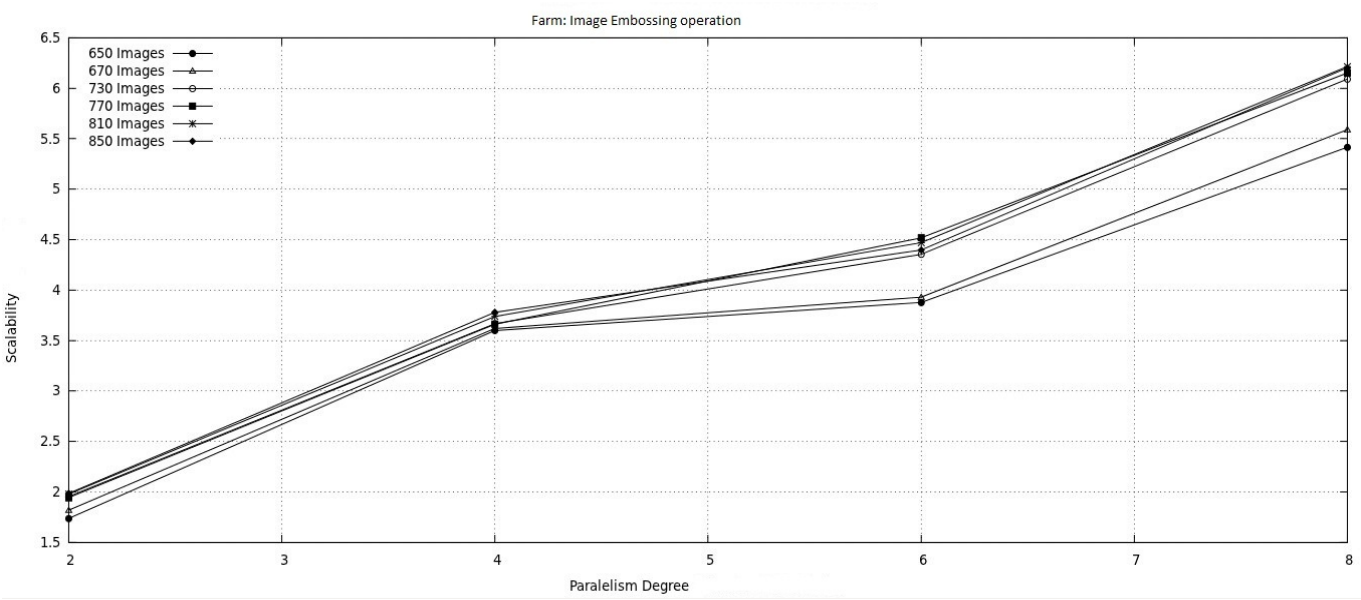

Figure 5.5: Farm Scalability: Image Embossing

From the result, we can see that as the number of farm-worker increases, scalability also increases. In fact some communication latency has to be paid for communication from farm-emitter to schedule a task to workers and also to collect results from workers by a farm-collector; this effect hinders the achievement of potential scalability. Specially, as number of worker increases this effect is more sensible.

In this subsection we also present efficiency of farm skeleton. Again we use the image embossing application to measure efficiency.
The following figure shows Efficiency of farm paradigm as number of worker increases. Also in this case we have multiple curves, each belonging to different stream length.

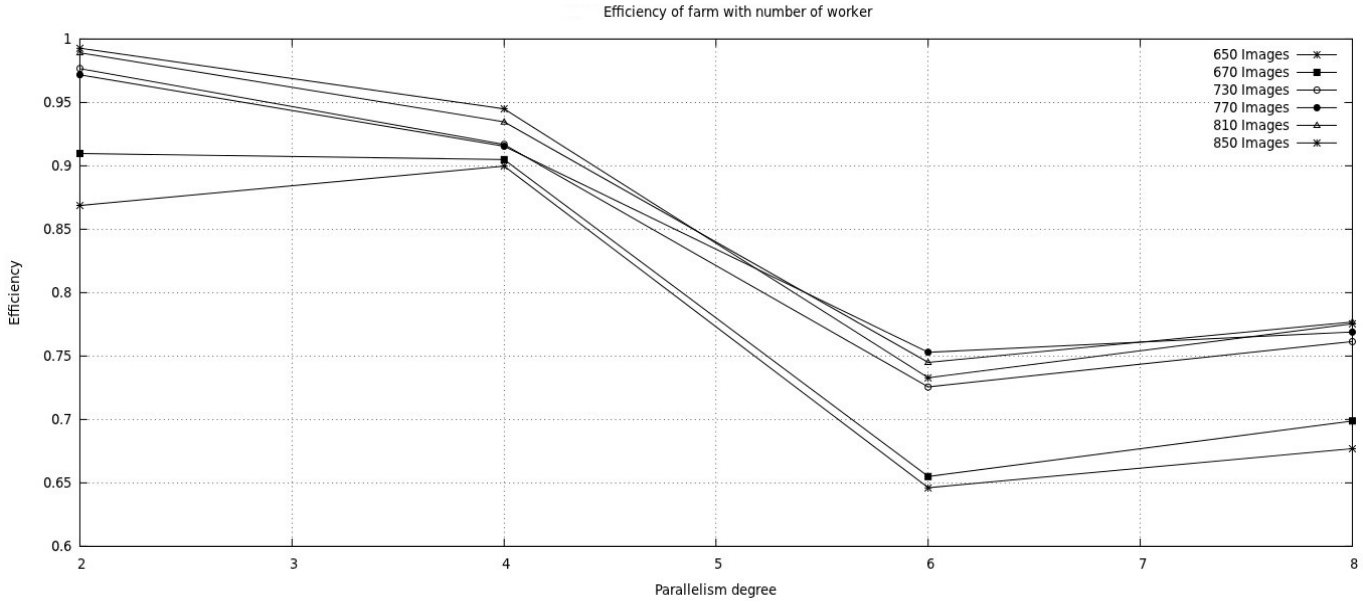

Figure 5.6: Farm Efficiency: Image Embossing

This diagram shows efficiency of farm for different input size. Note that this diagram is not showing an incorrect curve, it is in fact conformant to the scalability measurement presented in previous section. As number of farm-worker increases we can see that the efficiency graph declines, this means that, for a bigger farm-workers the gap between the ideal completion time and the actual completion time is more than expected.
Just like a good scalability is achieved for farm-worker 2 and 4, a good efficiency revels for those cases here too.
Actually, if we draw one diagram for each farm-worker as *"Efficiency vs input size"* we can see the same diagram as the one depicted for pipeline above.

Pipeline Trigonometric Function Computation

Following the same mechanism above, here we present result of trigonometric function computation using pipeline pattern. The pipeline has 4 stages, each stage performing a nested sin() function, as a result we will have a balanced pipeline stages. Each stage will then communicate the result of the computation to the next stage and the last stage will deliver result to the stream collector.
The following figure shows scalability of this computation.

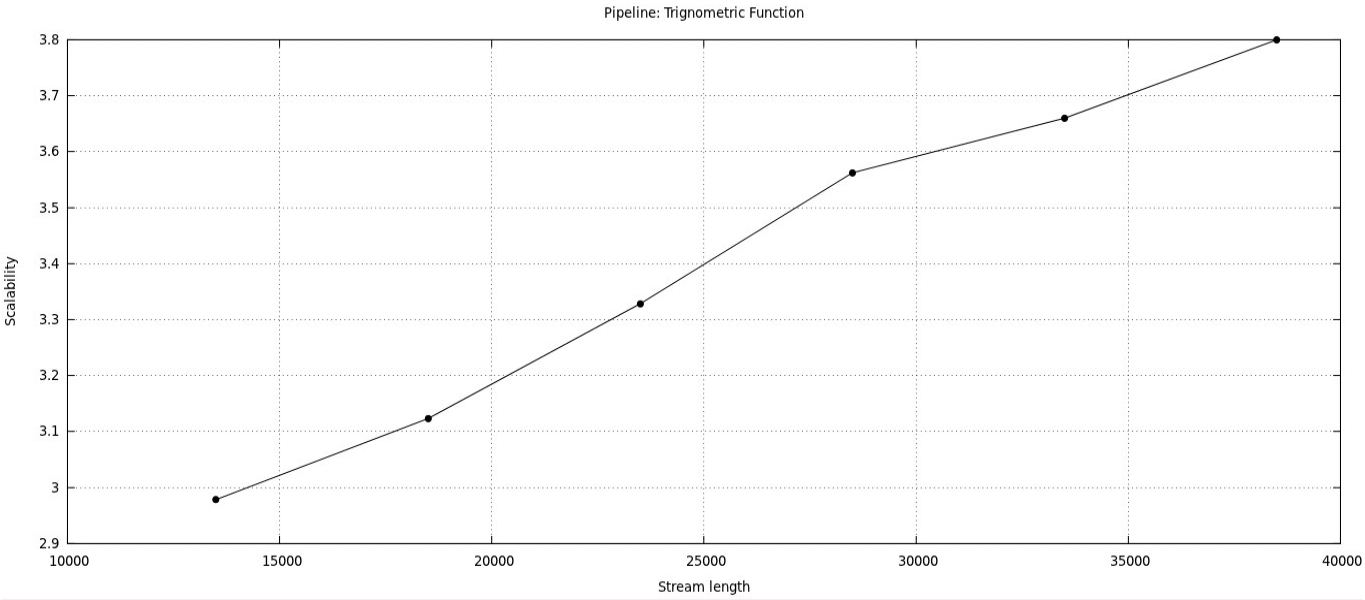

Figure 5.7: Pipeline Scalability: Trigonometric Function computation

This result shows the effect of trigonometric function computation as stream length increases. We can see that for few thousands of stream length the scalability of the computation is not that much significant. This is because the computation is too fine grain to scale as expected. As stream length increases we see the scalability raise in a quire fair rate.

The next result is efficiency of computation of trigonometric function.
Again a better efficiency is achieved as stream length increases.

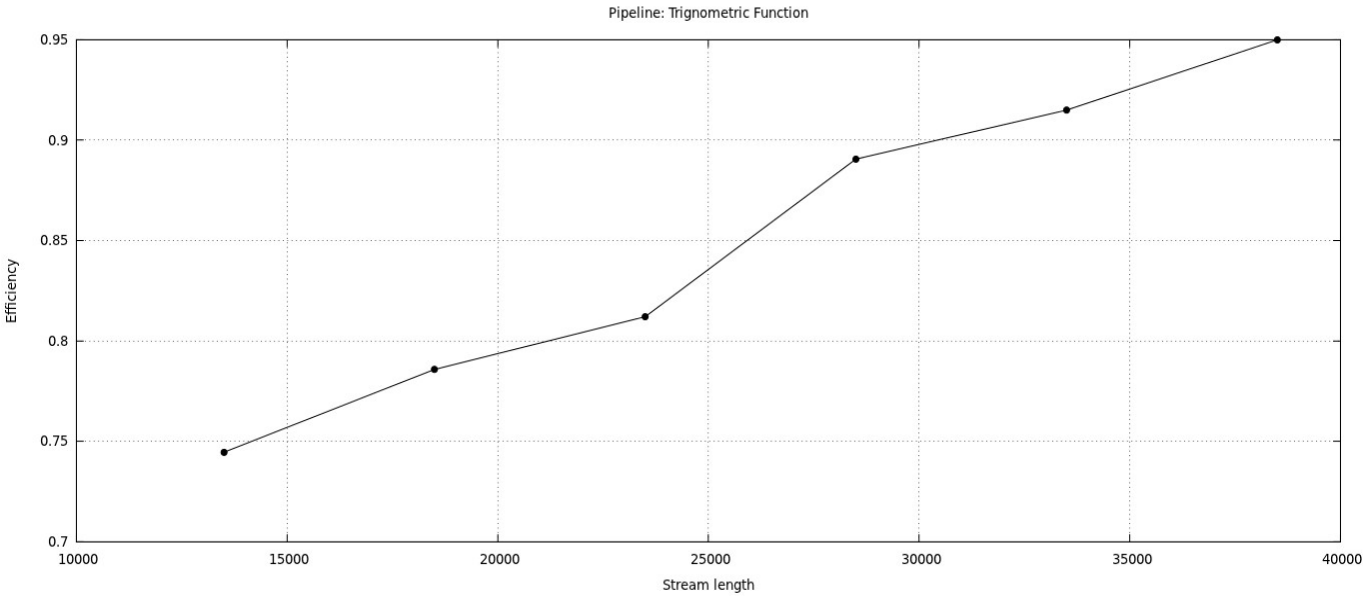

Figure 5.8: Pipeline Efficiency: Trigonometric Function computation

Farm Trigonometric Function Computation

In this case we use farm pattern to compute a nested sin() function. Each worker of a farm will compute this function and deliver the result to the collector.
In the following figures we will present the scalability and efficiency of mspp computing this application.

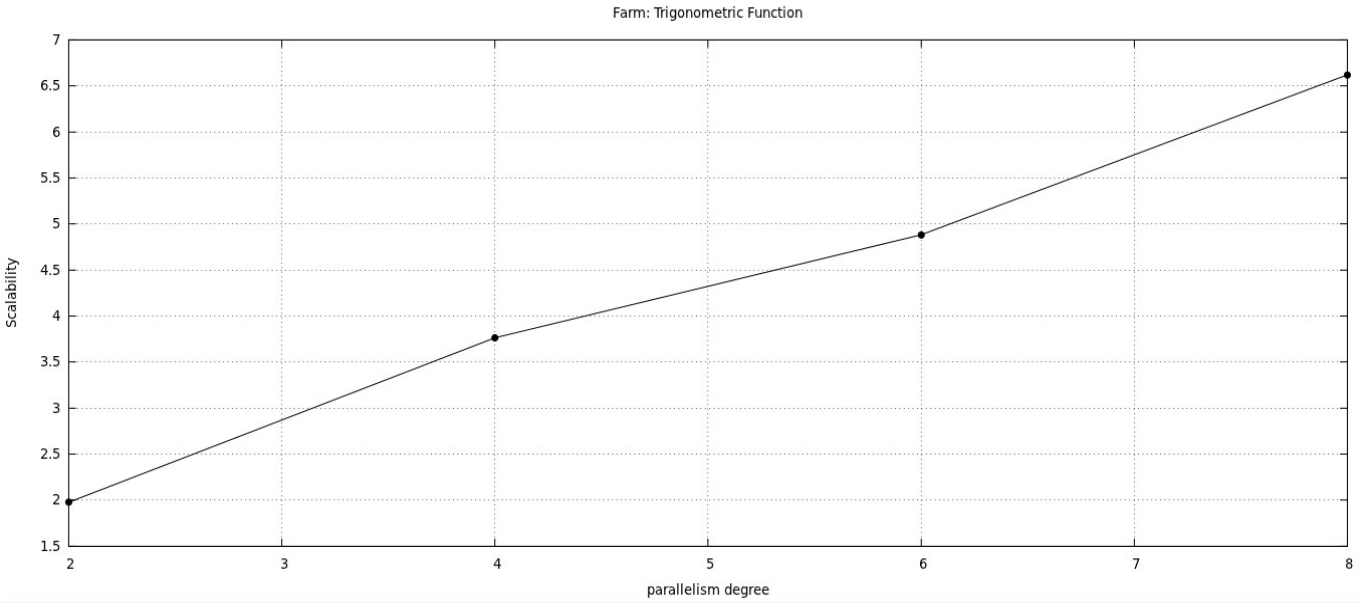

Figure 5.9: Farm Scalability: Trigonometric Function computation

The result is consistent with what we achieved previously; as number of worker increases the scalailbity increases. Again for the same reason presented above, for few farm workers a good scalability is achieved; while for larger farm worker, though it still keep scaling, the scalaibity is not as expected. As a result we have the following declining efficiency graph.

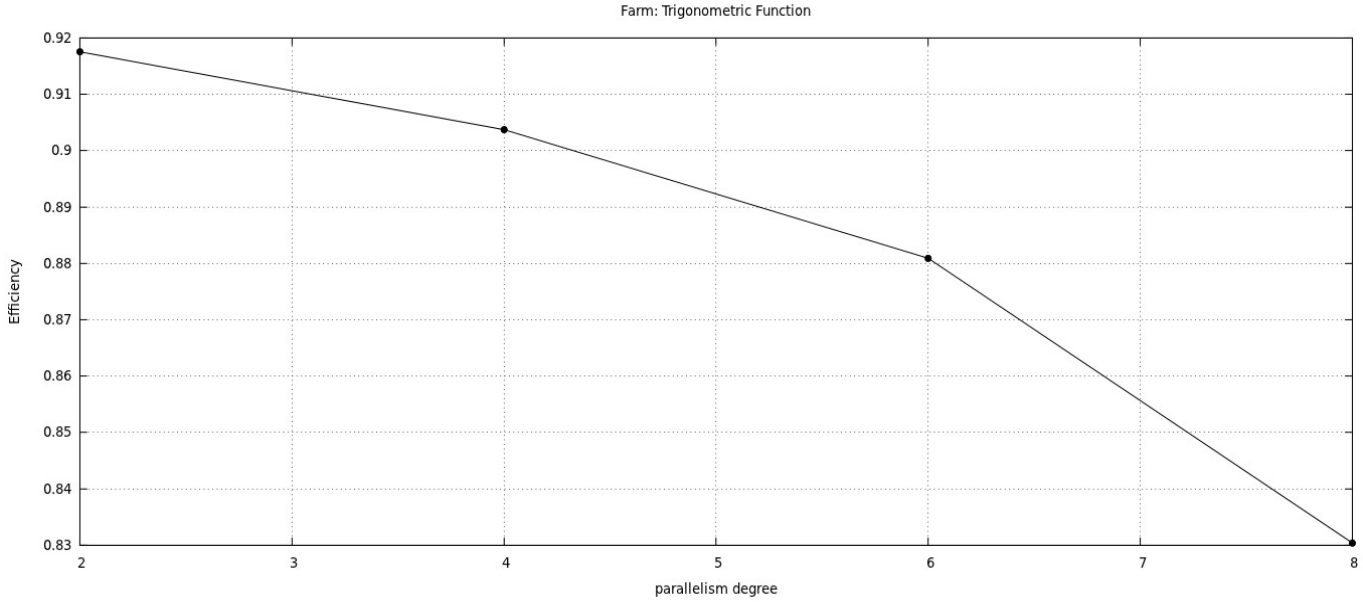

Figure 5.10: Farm Efficiency: Trigonometric Function computation

Pipeline Gaussian Elimination Computation

As we did for the above two application, here we will present the scalability and efficiency of our library by computing a Gaussian elimination application.
This computation is coarse grain and the communication will also be an intense operation. Each node will communicate a big size matrix that represents a linear system equation (refer in the first section of this chapter). In this case we use a 384X384 matrix.
In pipeline this computation is executed by partitioning the operation into a "forward elimination" followed by "back substitution". Those two operations will be computed by different stage of a pipeline.

The following two results show the scalability and efficiency of this computation respectively.

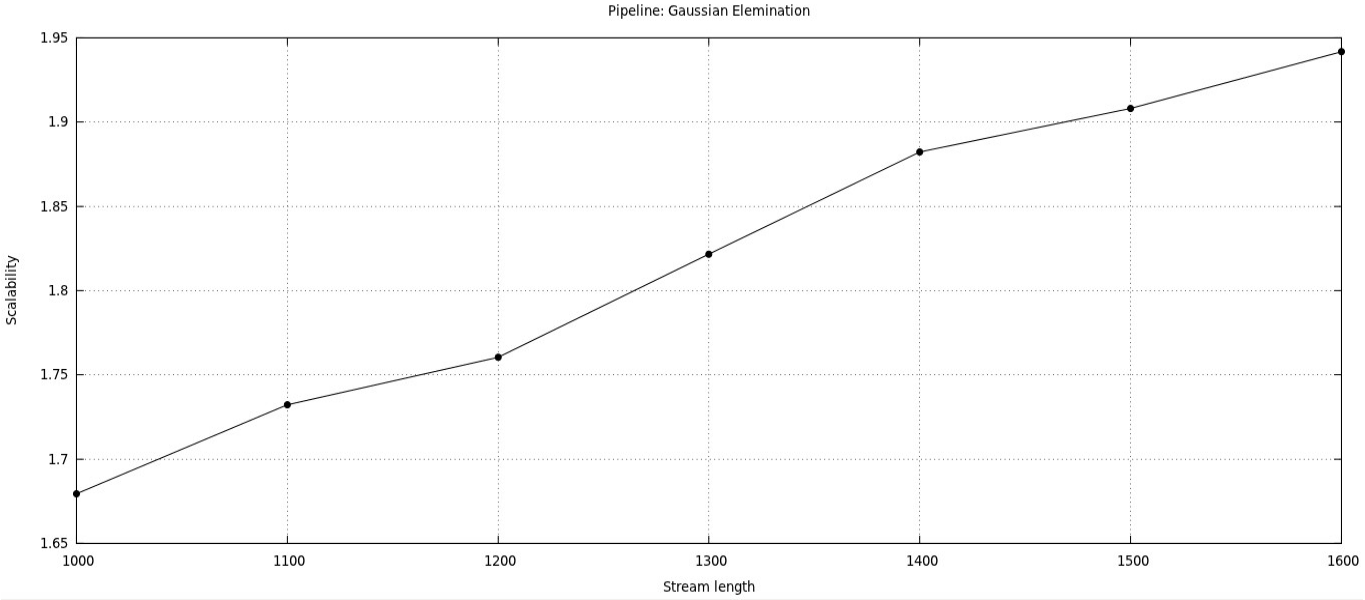

Figure 5.11: Pipeline Scalability: Gaussian Elimination computation

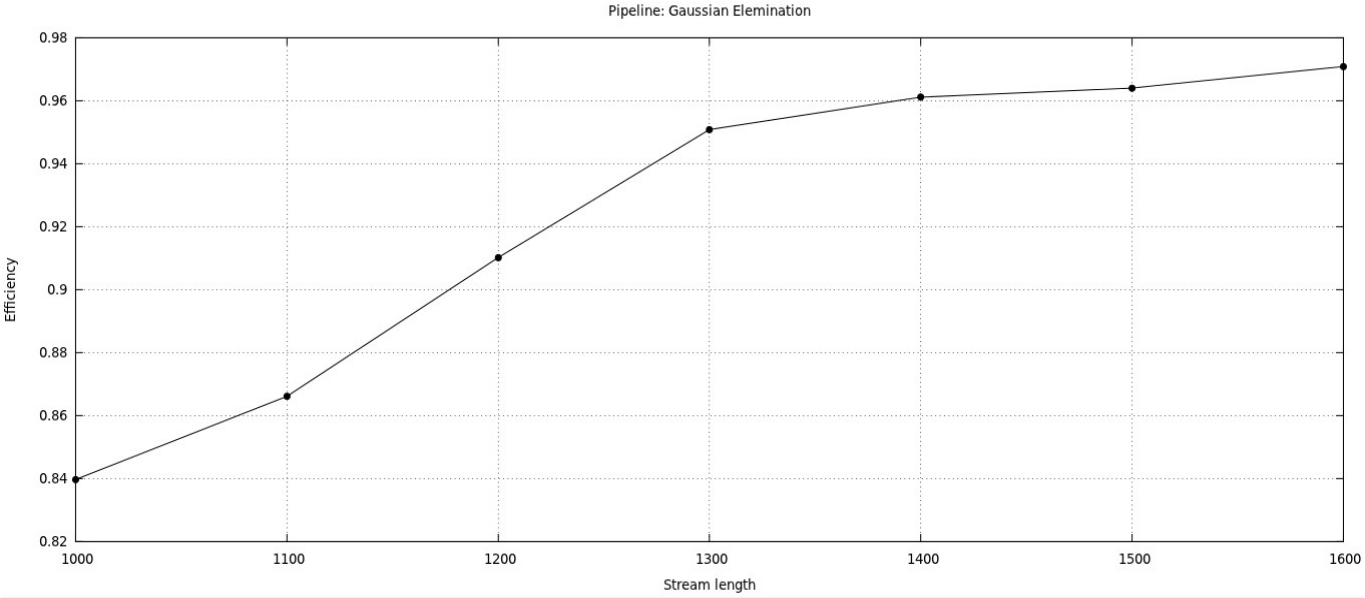

Figure 5.12: Pipeline Efficiency: Gaussian Elimination computation

The results are again consistent with the above results. In this case however since the computation is coarse grain and the communication is also an intensive communication, we can see that both scalability and efficiency rise very well even for small number of stream length. Mspp achieves this goal because the communication and computation overlap of this application is suitable to effectively hide communication latency.

Farm Gaussian Elimination Computation

Similarly we also show the computation of Gaussian Elimination using farm pattern. In this case farm workers will compute the overall operation of Gaussian Elimination. This includes the "forward elimination" and the "back substitution". As a result is ready each worker will communicate the result to farm-Collector. Again the matrix size of 384X384 is used.
The following two results show the scalability and efficiency of this computation respectively.

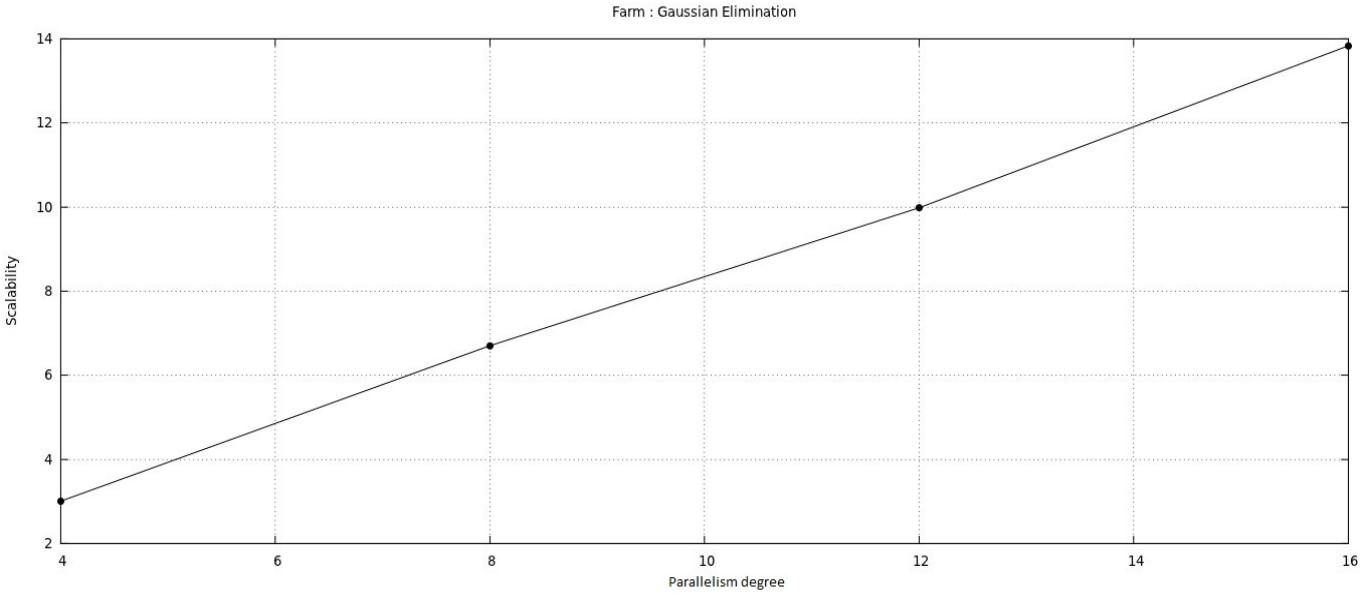

Figure 5.13: Farm Scalability: Gaussian Elimination computation

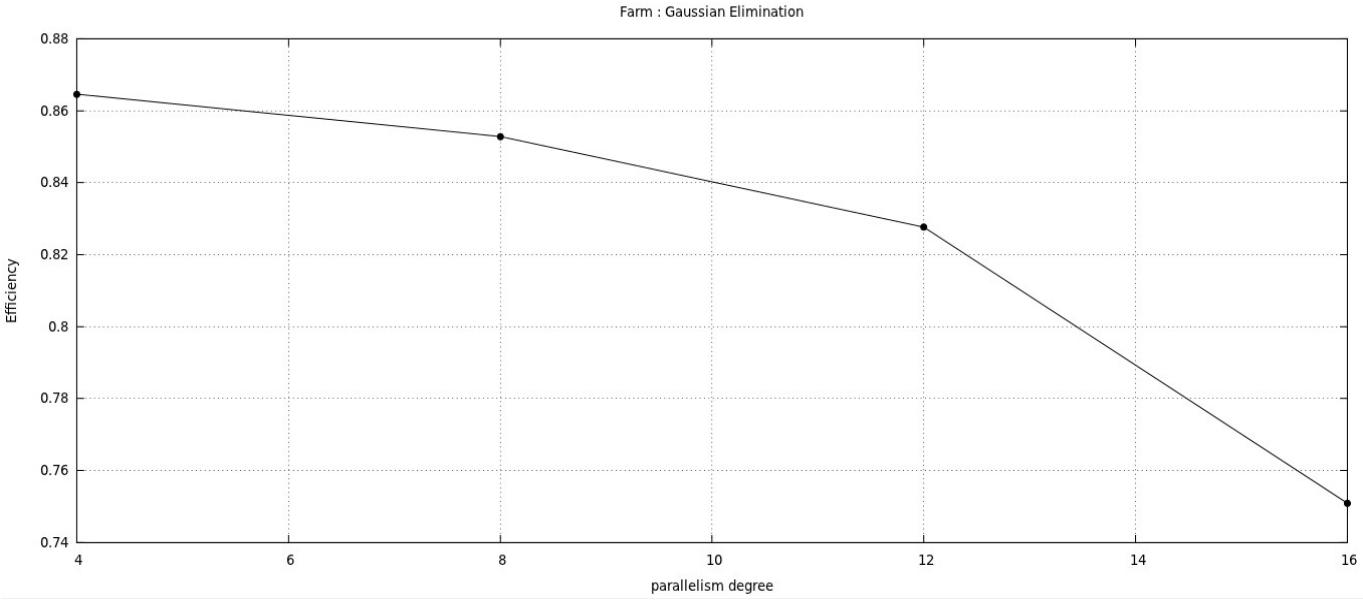

Figure 5.14: Farm Efficiency: Gaussian Elimination computation

The result is consistent with what we achieved previously; except that this application is coarse grain computation and the data to be communicated between farm worker and farm collector is big matrix. This imposes bottleneck effect, as a result we can see that for greater farm-worker is the scalability is not as expected.

We conclude this section by giving an overview of what has been covered in this section.

Conclusion

In this sub section we present scalability and efficiency of mspp. We present 3 different application executed using the farm and pipeline patterns of the mspp library.
For both image transformation and Gaussian elimination operations the library scales very well. This is direct consequence of the fact that those computations are coarse grain computations. For the trigonometric function computation, scalability is achieved after a greater stream length is given as an input. This in contrary is the effect of fine grain computation that this application imposes.
The library also achieves a better efficiency. From the result we can see that as number of farm worker increases efficiency graph declines. This means that for greater number of worker the actual execution did not achieve the expected ideal execution time. The main factor for this is the communication between the skeleton nodes. Specifically the communication from worker to collector is a bottleneck.

5.5.3 Comparison Of mspp and Fastflow.

In this comparison we have used different types application as presented above; Image processing operation, trigonometric function computations, and Gaussian Elimination numeric computation.

Comparison On Pipelined Image Processing Application.

We experiment this application with both libraries. The following figure shows a result of this experiment. In this experiment we use an image of 100X100 pixel size.
From the figure we can see how Fastflow performs much better than mspp for hundreds of images. After around 270 images are given as input Fastflow starts increasing in completion time more rapidly than mspp.
The sequential computation completion time increases exponentially, while Fastflow and mspp increase even less than a linear increase in completion time. One positive effect of mspp is the communication, computation overlap. All computation are done while communication is on progress. The other useful tool is the kernel module (KNEM) used for high-performance message passing between processes.

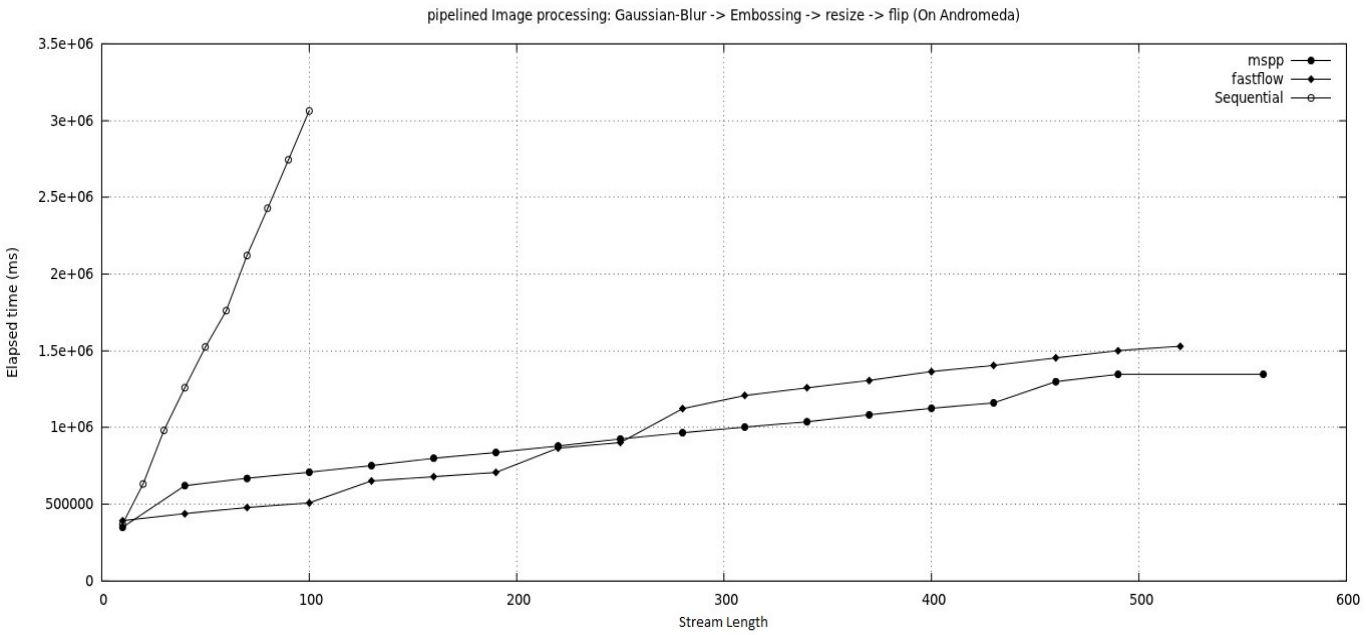

Figure 5.15: Pipelined Image operation using sequential computation, using mspp and Fastflow

Comparison On Image Embossing Using Farm Pattern.

We have experimented this application using different number of farm workers. We fix the image pixel size to 480X360, and the stream length to 1000 images. In addition to this, we

raise the magnitude of embossing factor so that the computation time can be balanced with the communication time.
The following figure shows the experimental result (completion time vs parallelism degree). Result of both Fastflow and mspp is shown.

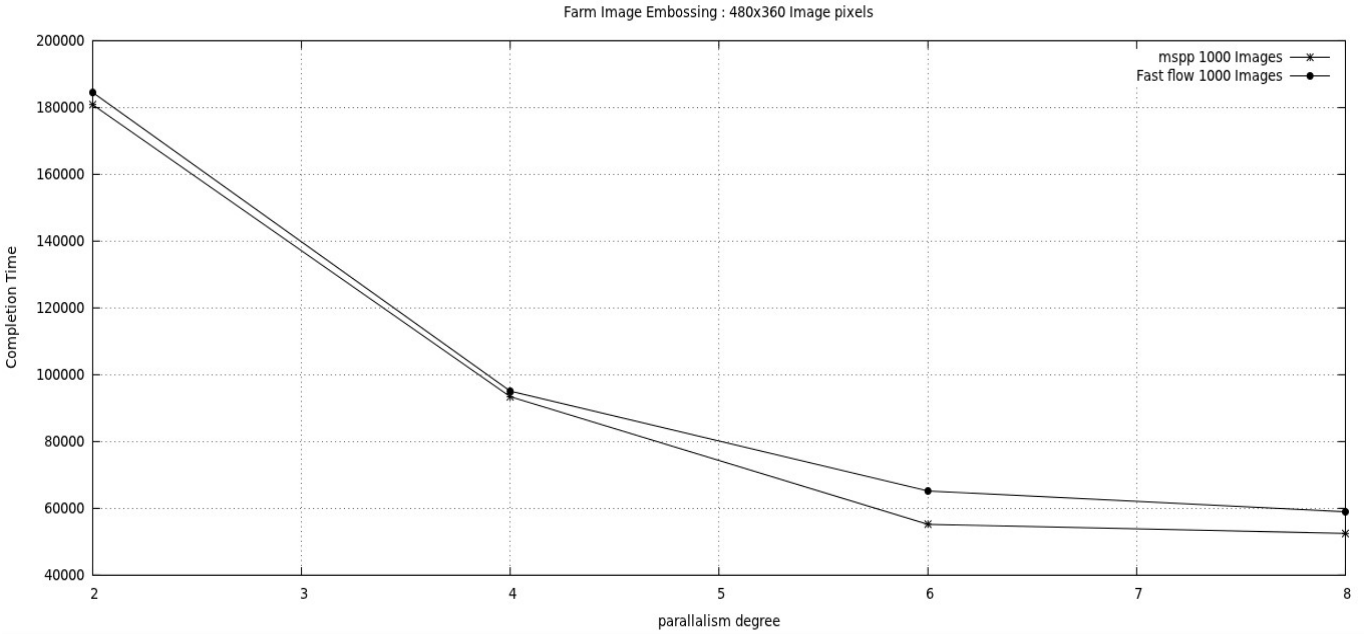

Figure 5.16: Farm Image Embossing operation. (completion time with respect to different number of workers)

The result is clear: in both implementations, as the number of worker increases the completion time decreases.
The other information that this figure provides is how the different parallelism degree react to the grain size. As parallelism degree increases the computation of the application with greater grain size is handled in less time.

The result of the two implementations can also be compared to be concluded as follows. In all cases mspp achieves better completion time, which is a direct impact of the coarse grain computation and communication/computation overlap.
This result is further analyzed in the next section where we experiment on different grain size by imposing small magnitude of image embossing factor.

Comparison On Trigonometric Function Computation

In this experiment we use a simple trigonometric computation describe in section 5.2. Again this application is developed to be executed by farm pattern and by pipeline pattern.

Using pipeline pattern

In this experiment, we have used the trigonometric computation by 4 stages of a pipeline. Each stage compute the same function $\sin^9()$ on an input item and communicate the result to the next stage in pipeline.
In this case a pipeline will have balanced stages, each stage executing the same code. The following figure shows the result of this experiment. Completion time of the pipeline pattern using the mspp framework and the Fastflow framework is shown.

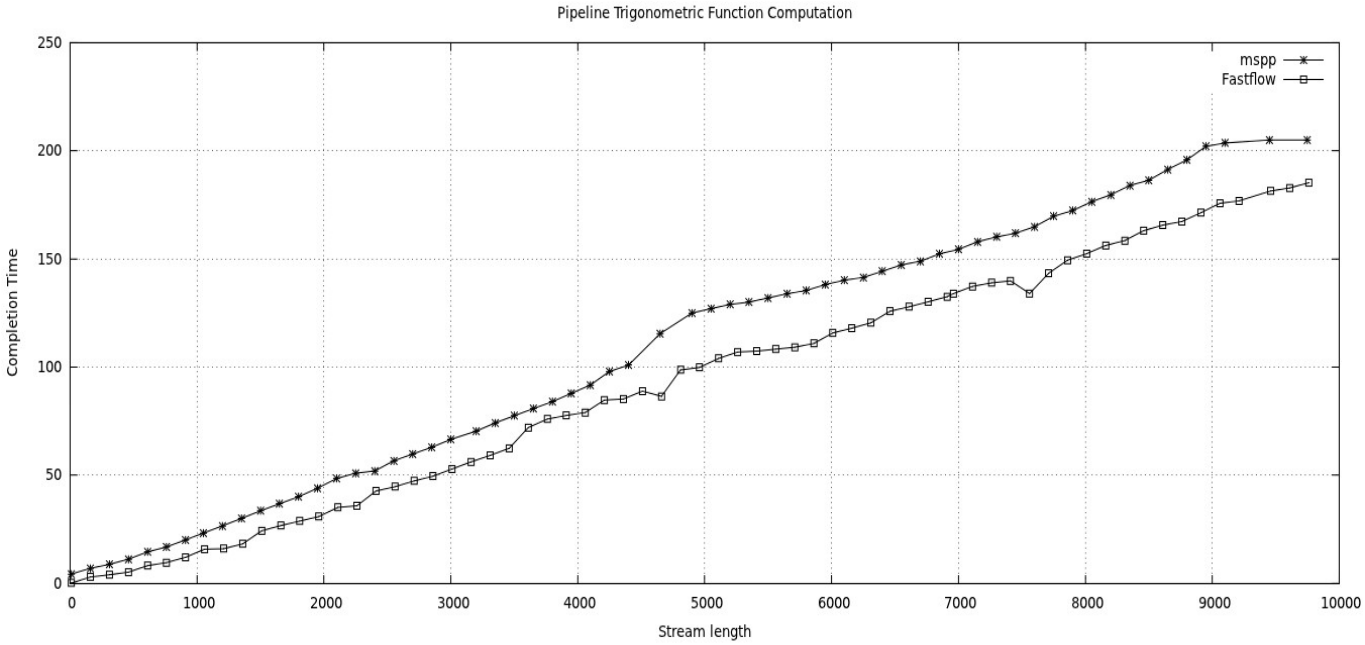

Figure 5.17: Trigonometric Function Computation: Pipeline pattern in mspp vs Fasflow

This computation is a fine grain computation; as a result we see expected behavior of the curve that Fastflow achieves better completion time compared to mspp.

Farm pattern using mspp and Fastflow

The above application is also experimented using farm pattern where each worker compute the nested trigonometric function, $\sin^9()$ iterating several times.
Farm emitter schedules a task to workers, workers will execute the function described above and communicate the result to farm-collector.
The following figure shows completion time of the above application executed using a farm pattern. Different curve is plotted for different farm-workers. The diagram shows both the result of mssp and Fastflow. Here we show the case when 2 worker and 8 workers are used to compute the application. In both cases Fastflow achieves better completion time. Again this is the effect of fine grain computation. As number of farm worker increases mspp become closer to Fastflow, thus trying to overlap more computation. In this case we don't see the bottleneck effect between farm-workers and farm-collector because the data to be communicated is very small to impose this effect.

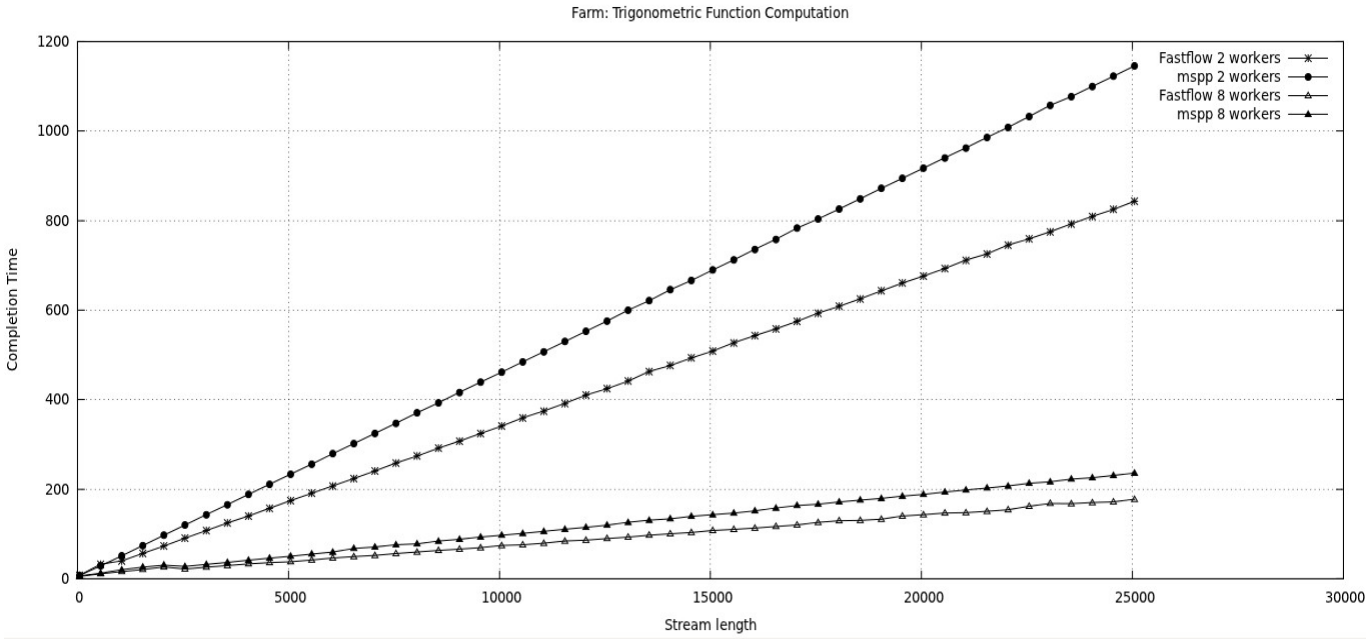

Figure 5.18: Trigonometric Function Computation: Farm pattern mspp vs Fastflow

Comparison On Gaussian Elimination Computation

In this section we present the computation of Gaussian elimination.
This application contains an expensive operation that operates on matrices. For this reason, experimentation of this application is performed on a the machine with 24 cores, *"Titanic"*.

Using Farm Pattern

Farm Emitter will schedule the task (matrix representing a system of linear equation) to workers. Each worker of a farm will compute a gaussian elimination (forward elimination, followed by back substitution) on a give matrix element and deliver the result to the farm collector.

In this case we experiment this application by fixing stream length to 500 matrices and grain size to be 128X182 for the first experiment and 640X640 for the second one.
The following results shows the case when 4, 6, 8, 12, and 16 workers are used to execute the application.

Both the mspp and Fastflow are presented on the same diagram.

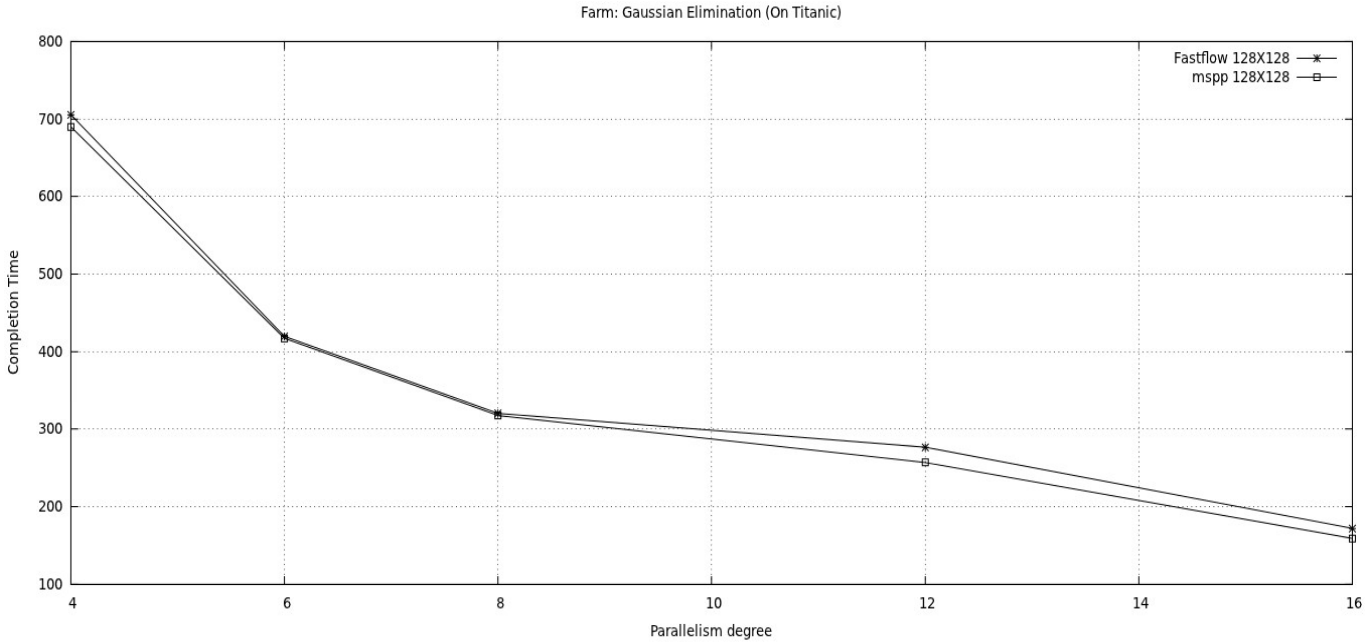

Figure 5.19: Gaussian Elimination: Computing 128X128 matrix

In this result a 128X128 matrix is used. The computation of Gaussian Elimination increases with the matrix size. as a result we can see that mspp achives better completion time benefiting from the communication and computation overlap.
To give us more clear idea of the computation we raise the grain size of the matrix to 640X640. The following figure shows this result. In this case the completion time gap between the two implementations become big, which again is the effect of computation grain.

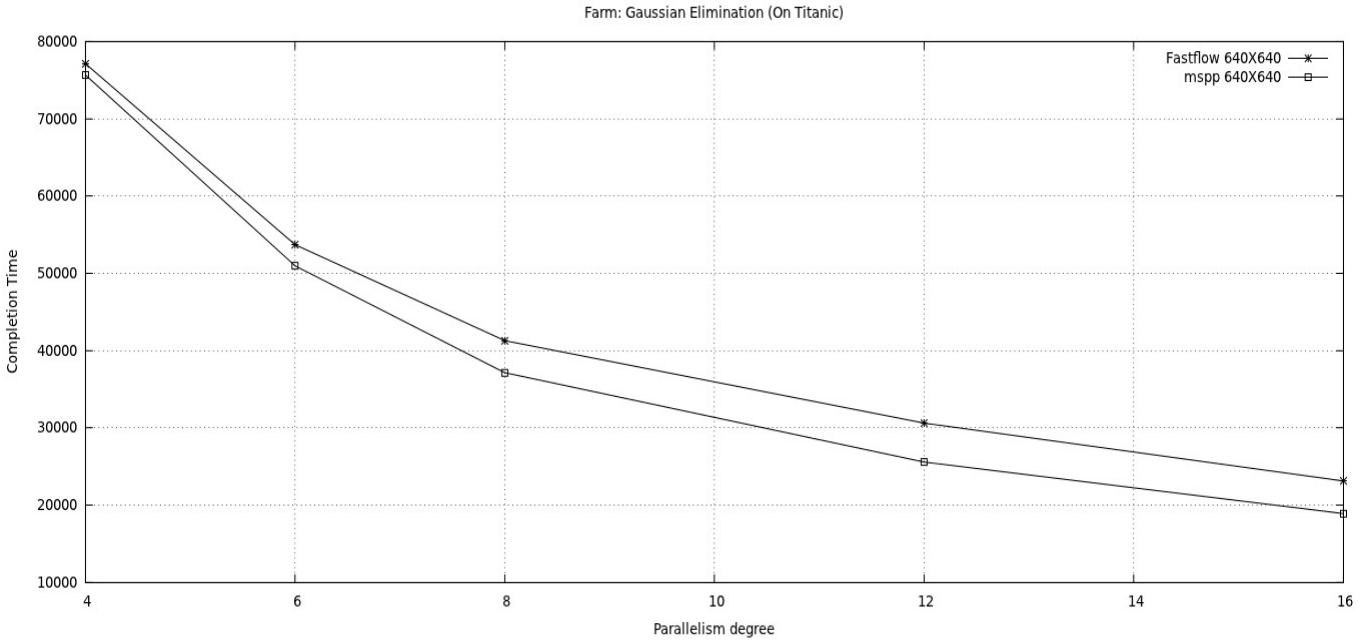

Figure 5.20: Gaussian Elimination: Computing 640X640 matrix

In the following figure we further show the effect of different gain size. we fix the parallelism degree to 8, and we depict different curve for those different grain sizes. We can see that the result of the 128X128 matrix in this result mostly overlap and only small size gap between the two curves, which is what we show separately above.

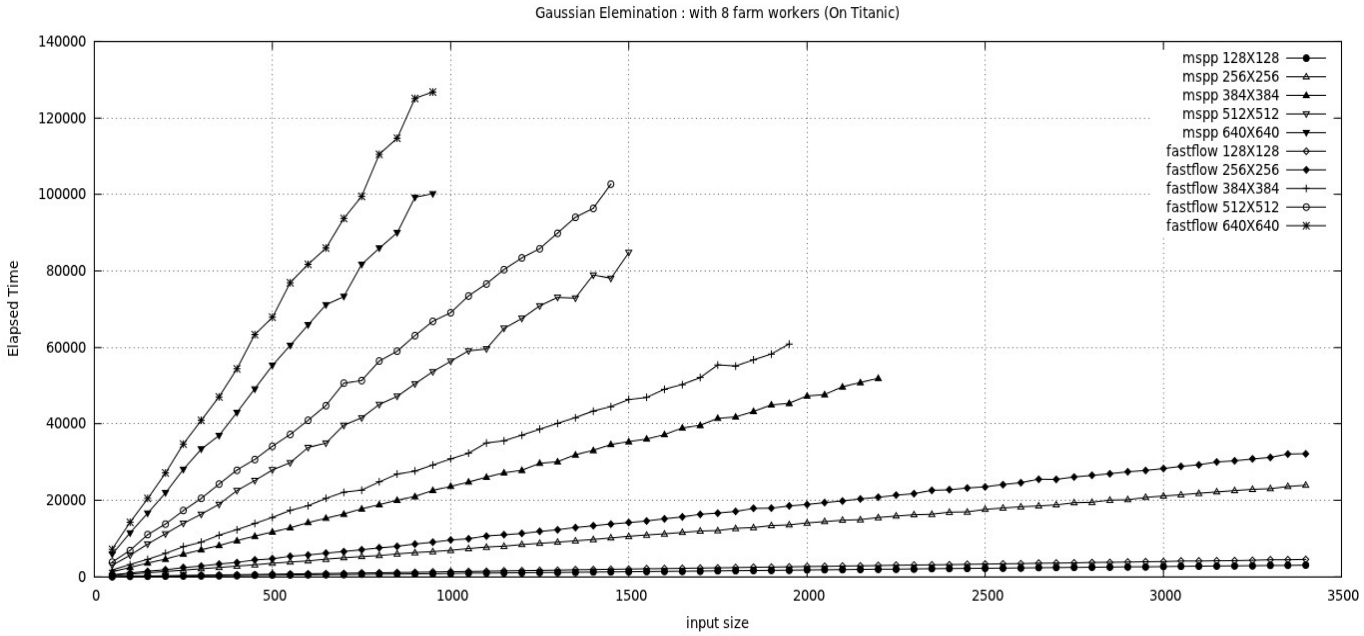

Figure 5.21: Gaussian Elimination: using 8 farm worker mspp vs fasflow

Using Pipeline Pattern

We have also experimented Gaussian elimination application using pipeline pattern. This computation is also suitable to be turned to a pipeline pattern. As presented previously, we have two main steps in solving Gaussian Elimination: The first step "forward elimination", and second step "back substitution" to find the solution of the system.
This two steps can fit into a two stage pipeline where first stage compute the "forward Elimination" and second stage compute "the back substitution". Those operations are a bit expensive to be computed by sequential stage. The "forward Elimination" stage is a bit more expensive operation than the second stage, the "back substitution". For this reason we decide to use a farm skeleton in both stage of a pipeline, where the first stage execute by 4 farm-workers and the second one using 3 farm-workers.
This is an interesting experiment result that gave us a good realization of skeleton pattern composition of the two libraries.
The following diagram shows result of both implementation.

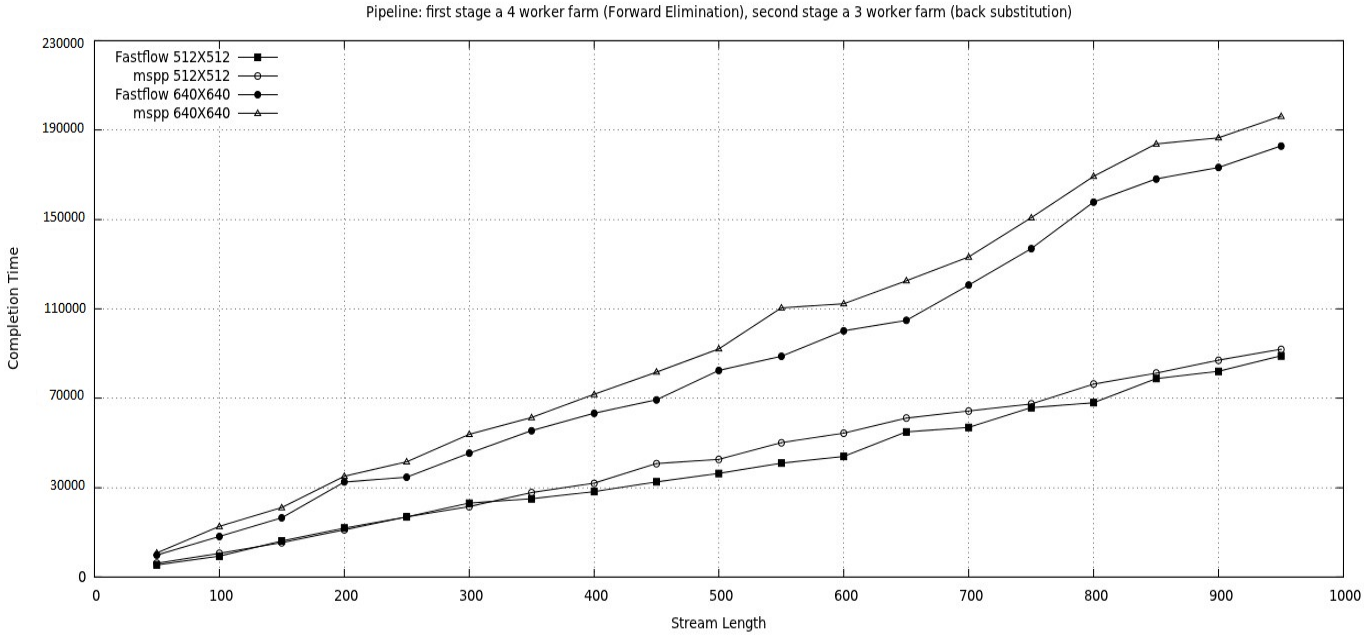

Figure 5.22: Gaussian Elimination: using pipeline pattern mspp vs Fasflow

The above diagram shows a different result than what is achieved so far when comparing the two implementations.
Unlike previous results this time Fastflow achieves better completion time even as the grain size increases. The above result shows when matrix dimension of 512X512 and 640X640 is used for computation.
This comes from the following main reason:
mspp implementation always uses a farm-Emitter and a farm-Collector in a farm skeleton whether it is stand alone or used for composition. So in the above scenario where we have farm in two stages of pipeline, there should be some communication (thus becoming bottle-

neck) between the collector of farm in first stage and emitter of farm in the second stage. This bottleneck effect is more sensible when the grain size increases. This issue is put forward in the future work section of this thesis.
Fastflow on the other hand has a good implementation consideration for this issue, in Fastflow there is a way to not use a collector at a given farm skeleton, instead directly communicate it to the emitter of the second stage.

The following diagram clearly shows the difference

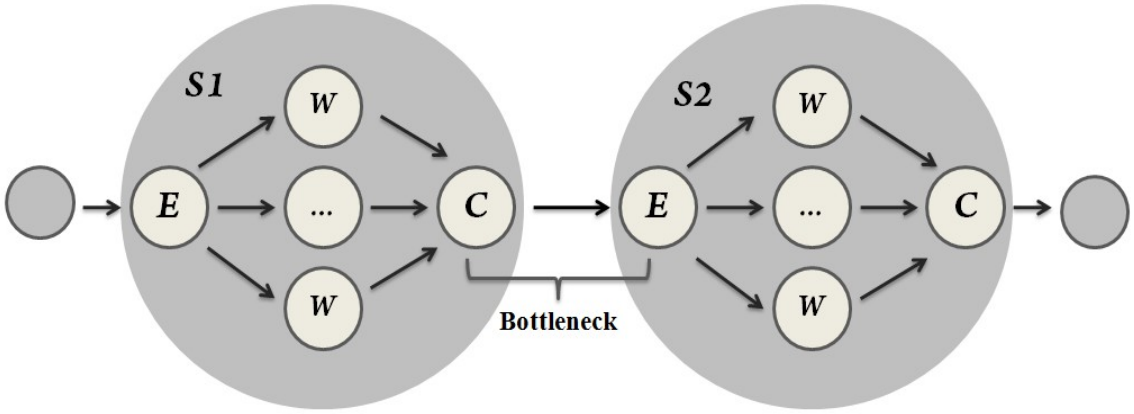

Figure 5.23: mspp implementation of composition of Farm in pipeline stages

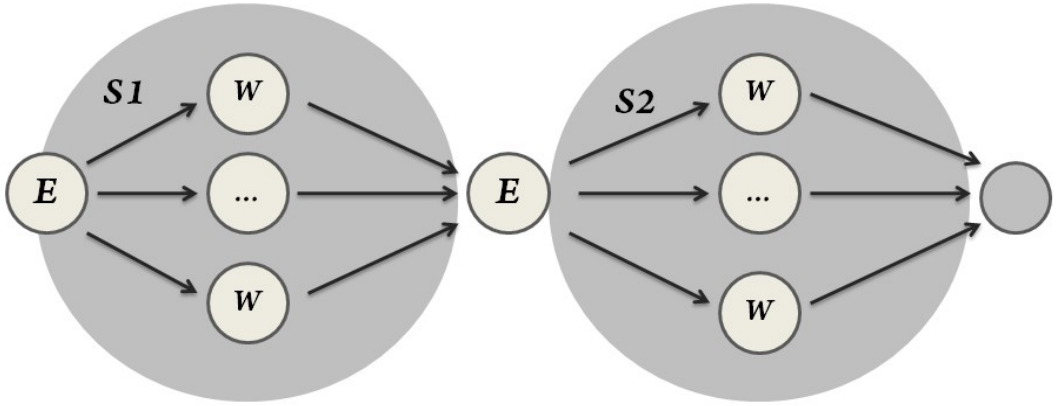

Figure 5.24: Fastflow possible implementation of composition of Farm in pipeline stages

In the following section we will also present the comparison of both implementation on their scalability and efficiency.

Scalability Of Pipeline

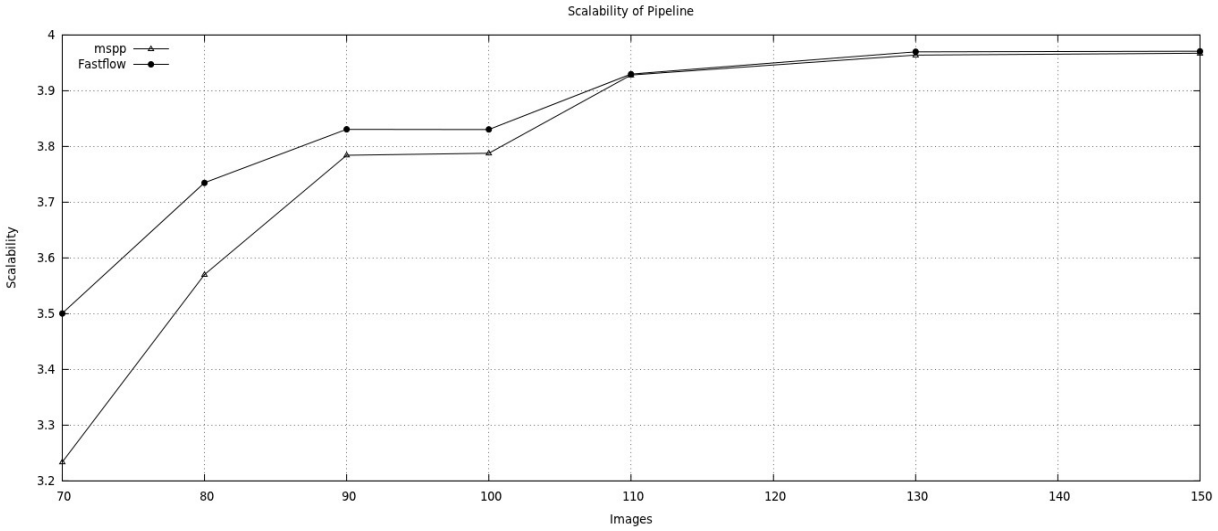

Figure 5.25: Scalability of pipeline pattern. mspp vs Fastflow.

The diagram shows that as number of input size increases pipeline able to achieve better and better scalability. This result shows a pipeline of 4 stages. Thus as expected, scalability up to 4 is almost achieved after few hundreds of inputs. We can see that the effect of fill-in transient and emptying transient phase of a pipeline in the few input size region of the graph; as the input size increases this effect become negligible, thus scaling very well.
The difference of scalability between mspp and Fastflow is consistent with what we present in previous sections. By considering figure 5.25 we can see that Fastflow achieve better completion time for few hundreds of image inputs. Here image grain size of 100X100 is used, as a result Fastflow scalability is better than that of mspp.

Scalability Of Farm

The following figure shows scalability of farm pattern as number of worker increases. In this case we have multiple curve, each belonging to different input item number of a stream(scalability is measured for image embossing computation).

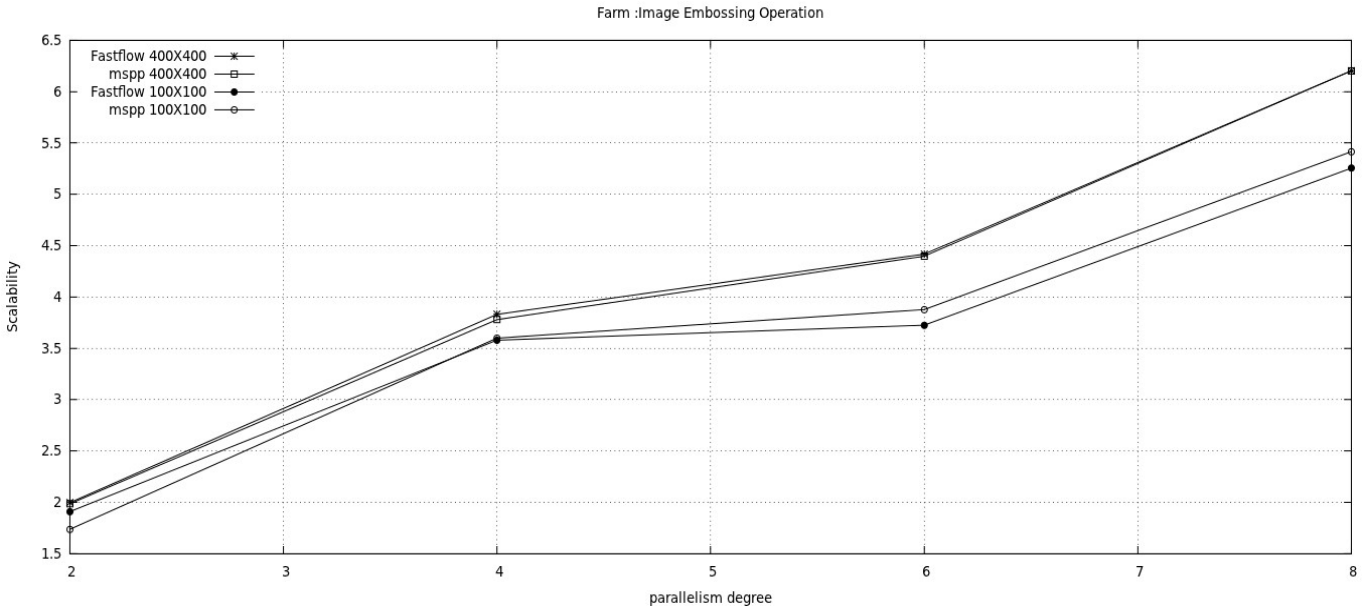

Figure 5.26: Scalability of farm pattern : Image Embossing. mspp vs Fastflow

From the result, we can see that as the number of farm-worker increases, scalability also increases. In fact some communication latency has to be paid for communication from farm-emitter to schedule a task to workers and also to collect results from workers by a farm-collector; this effect hinders the achievement of a the potential possible speed up. Specially, as number of worker increases this effect become more sensible.
The above diagram shows that for farm-workers 2 and 4 we can see that, a scalability of approximately 2 and 4 is achieved receptively. As farm-worker increases (see for when it is 6 and 8 workers) even though it keeps increasing, it did not increase as per the number of nodes used to accelerate the computation.
The other realization here is the grain size of the input items, as grain size increases we always achieve better speed up, because the sequential computation will take greater and greater time as grain size of input items increase.

Comparing the two implementations, Fastflow achieves better scalability for the 100X100 image pixel computation (thus fine grain computation). As image pixel increases, 400X400 which imposes the computation grain to be higher, mspp achieves a slightly better scalability.

Efficiency Of Pipeline

The following figure shows how pipeline achieve efficiency as the stream length increases. The result of both mspp and Fastflow is shown. The difference between the two curves is again consistent with the previous results. Here small input size is considered as a result, Fastflow achieves better efficiency until the input size reaches 140 Images, after which both curves overlap.

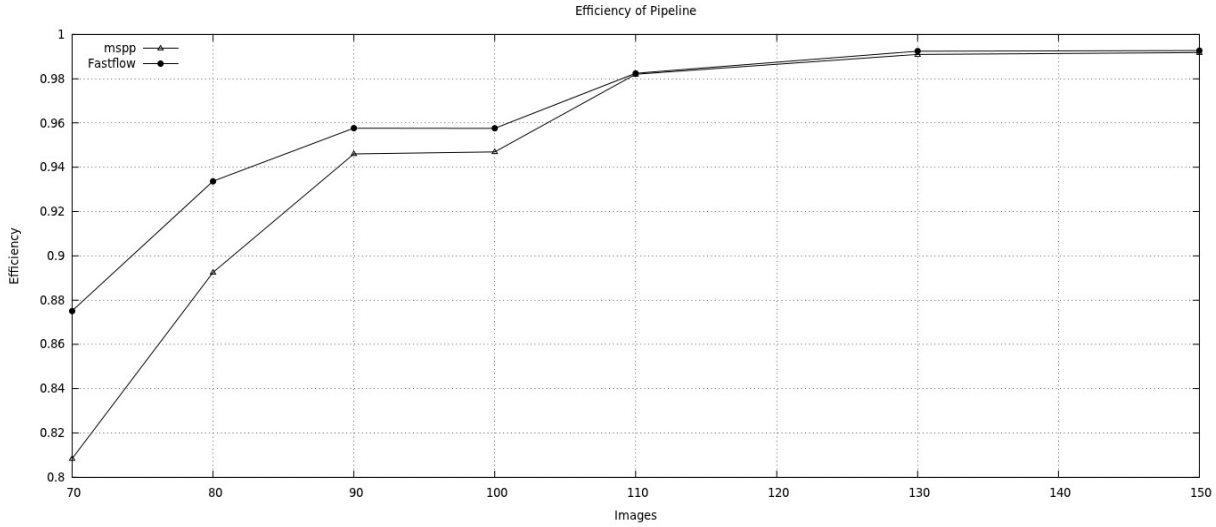

Figure 5.27: Efficiency of pipeline pattern. mspp vs Fastflow

Efficiency Of Farm

In this subsection we also present efficiency of farm skeleton. Again we use the image embossing application to measure efficiency.
The following figure shows Efficiency of farm paradigm as number of worker increases. In this case we have multiple curve, each belonging to different stream length.

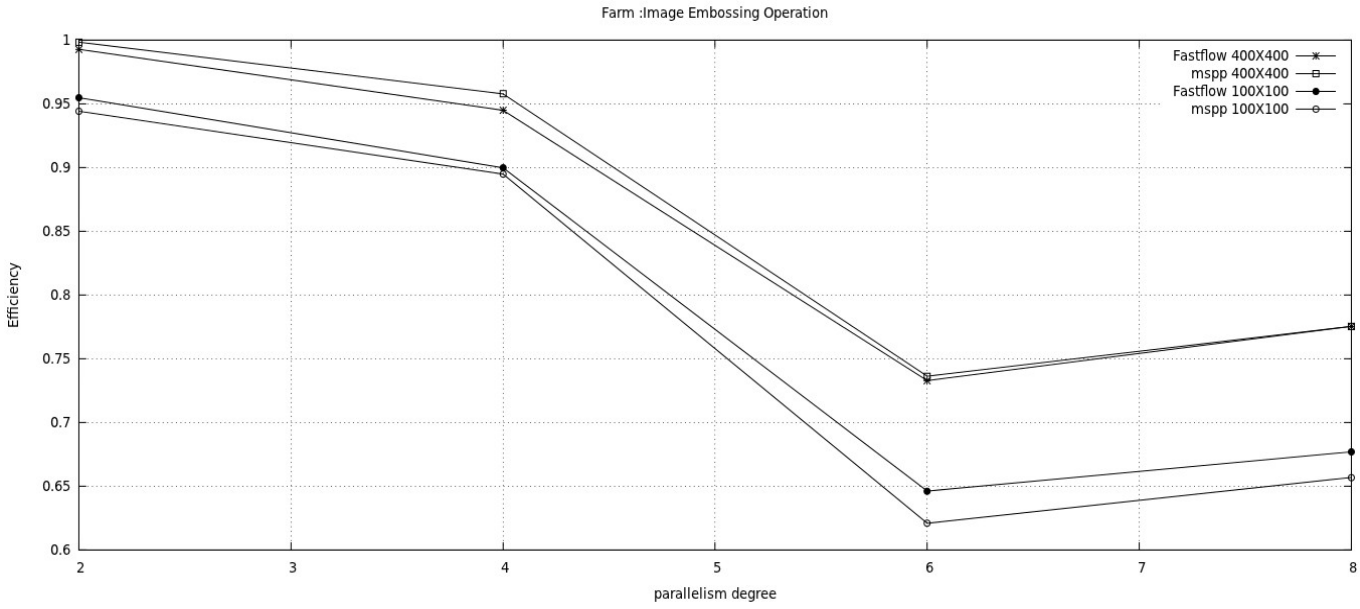

Figure 5.28: Efficiency of farm pattern. mspp vs Fastflow

This diagram shows efficiency of farm for different input grain size.
Just like a good scalability is achieved for farm-worker 2 and 4, a good efficiency revels for

those cases here.
Again with the same reasoning for scalability, as the input grain size increases, a better efficiency is achieved.
Comparing the results of mspp and Fastflow, we can see that the effect of grain size; which has a direct relationship with the computation grain. For Image embossing where greater magnitude of embossing factor is used, the increase in image size will contribute to increase the computation grain. As a result we can see that while Fastflow achieves better efficiency for 100X100 image size, mspp become slightly better than Fastflow for 400X400 image size.

Conclusion

In this section we present the comparison of mspp and Fastflow. We also explicitly deal with comparison on the two implementation based on their scalability and efficiency.
In all experimental cases we got a result where Fastflow achieves better performance whenever the computation is fine grain. mspp on the other hand achieves better performance when the computation is coarse grain. Specially, when both the computation is coarse grain and the communication is intense operation, mspp achives better performance by benefiting from communication and computation overlap. We also present the case when the communication is intense but the computation is fine grain in which case Fastflow achieves a good performance while mspp is affected by the communication bottleneck between farm-worker and farm-collector.
In the next section we will further compare the two implementations focusing on varying the computation grain.

Comparison On Computation grain

From result of previous sections we can see that sometimes Fastflow is better than mspp and the other time mspp is better. This effect is seen on different applications and in fact the effect is the direct consequence of computation grain. Therefore we decide to do more experiment to show how the different implementations handle computation grain.

Here we present the trigonometric function computation. We vary the computation grain of a nested sin() function. The label on graph shows how much time it takes to compute the function using a single input (it is given in microseconds).

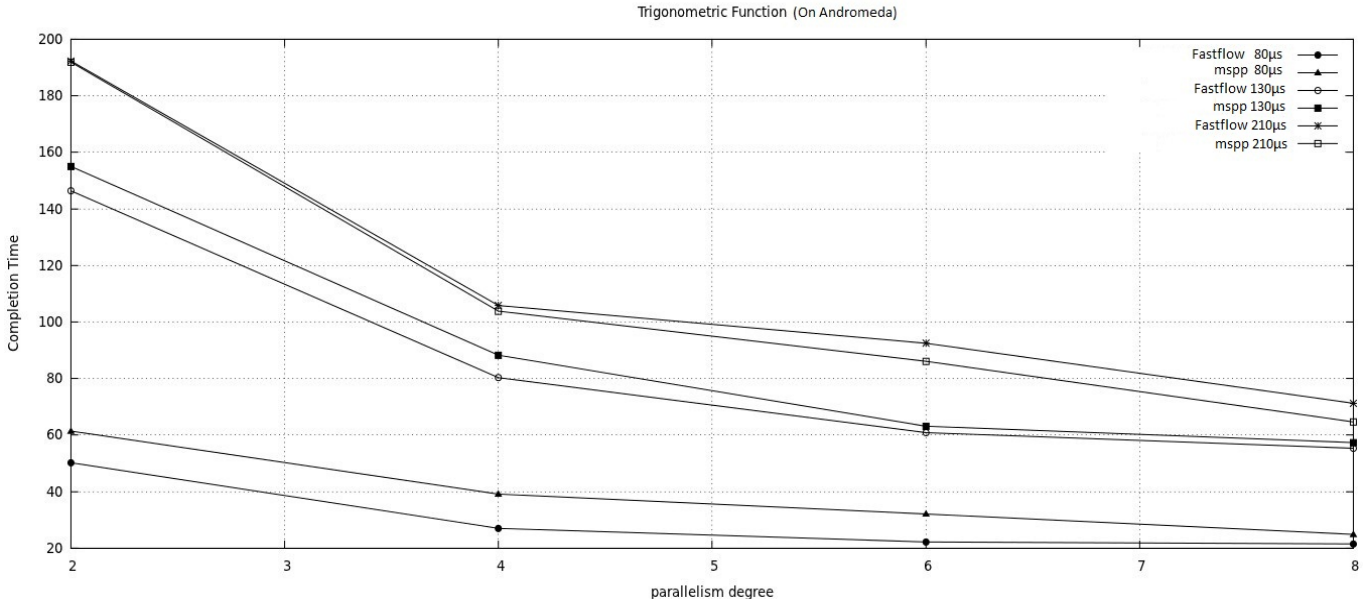

Figure 5.29: Trigonometric function Computation varying computation grain (On Adromeda)

This result shows that for fine grain computation Fastflow achieves better completion time. As grain size increases mspp becomes better in completion time.

The other very important result is shown below. Here we run the same application above on a machine with greater number of core (*Titanic*). As a result we can see that Fastflow become worse only after computasion grain is rise to 610μs. This shows how Fastflow make use of the available resources.

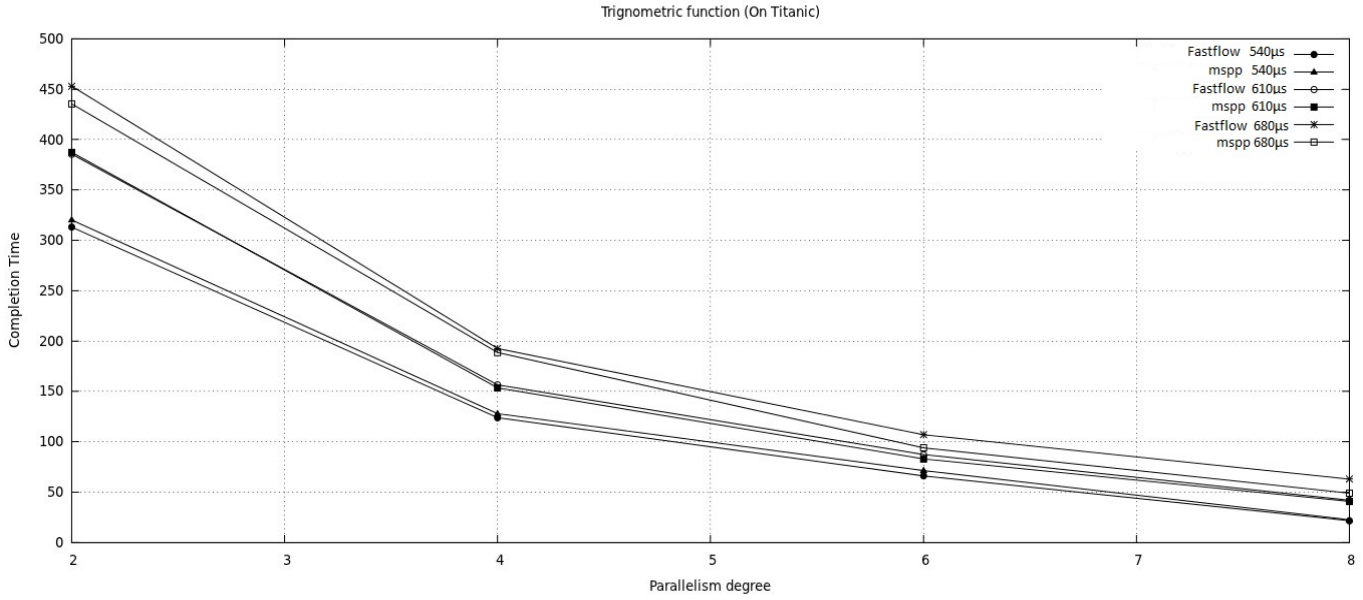

Figure 5.30: Trigonometric function Computation varying computation grain (On Titanic)

Similarly we show the result of Image transformation application. Here we use Image embossing operation. A fine grain computation is used by specifying a small magnitude of embossing factor (a factor of 0.075). As a consequence the computation is not that much intense. Now with this in mind, we can see that as the image pixel size increases the communication of this data will take more time than computing the image.
With Fastflow based on shared memory it able to achieve better completion time. In this case mspp take more time communicating the matrices between nodes. As specified in the previous section the communication between farm worker and the collector is a bottleneck. Though KNEM optimize the communication of data between processes, the size of the image pixel, the very fine grain operation used and the bottleneck between workers and farm-collector add up to impose the effect we see in the following result.

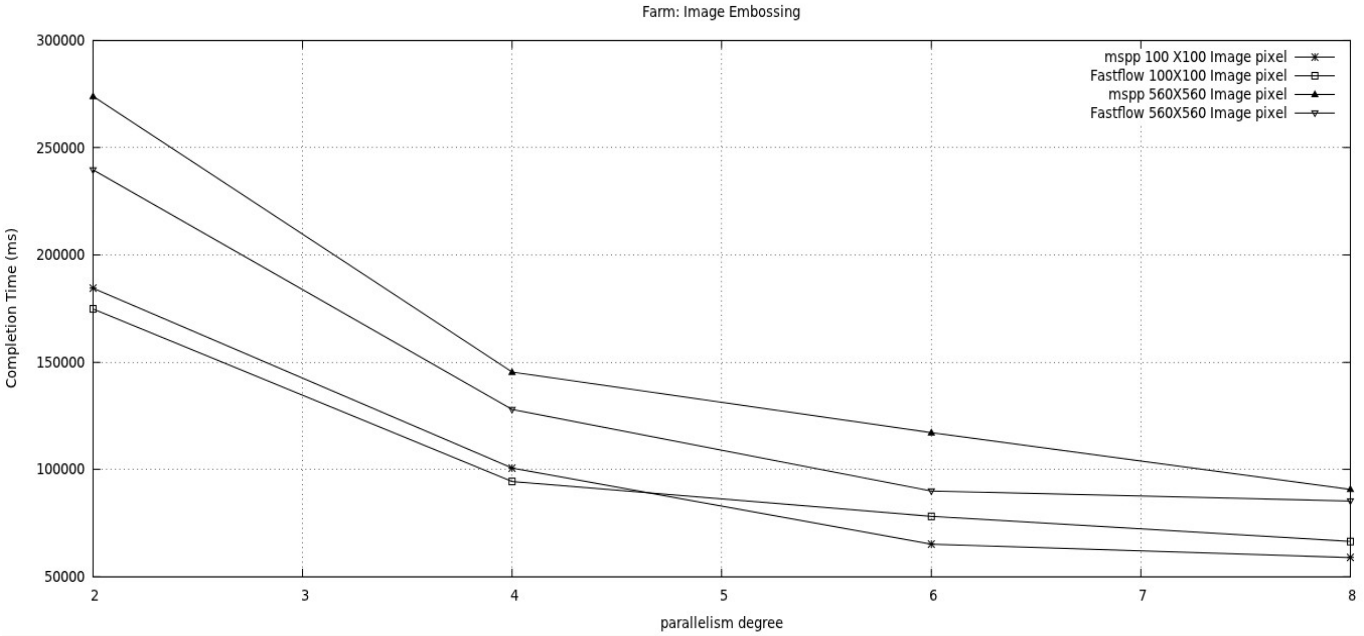

Figure 5.31: Farm Image Embossing operation. Computation grain.

For comparison on Gaussian elimination computation, we already show such comparison based on grain size, see figure 5.19 and 5.20. In that case the computation is coarse grain, and the communication is also intense that can overlap with the computation. For this reason mspp able to achieve better completion time.

This brings us to the conclusion of results achieved in this section. In the following chapter, the overall conclusion of experimental results and future work is presented.

Chapter 6

Conclusion And Future Work

In this chapter we give conclusion of the overall experimental result by pointing to the main objective of the thesis and mentioning the obtained result, and we also present feature work of the thesis.

6.1 Conclusion

One objective of this thesis is to produce a stream parallel skeletons framework that will be consumed by programmer as a library. Besides that, this thesis aims at working on abstraction of the skeleton implementation as much as possible.
As presented in previous chapters, these two goals are achieved in a well structured manner. The other main objective of this work is to experiment the stream parallel skeleton frameworks. In this work we compare this implementation with Fastflow [11] implementation. The implementations are based on message passing programming model and shared memory model respectively.
The overall conclusion is given as follows. For all applications experimented in this work a consistent result is collected.

i) Both implementations able to achieve better scalability and efficiency with respect to the "execution of parallelism degree 1" and "sequential" counterparts respectively.

ii) Fastflow achieves better efficiency and scalability for fine grained computation. As data grain size and computation grain increase, mspp become slightly better than Fastflow.

iii) Tuning shared memory configuration of open-MPI helps to get minimal communication latency (KNEM supports a zero copy communication).

iv) Communication and computation overlap implementation is another very important part that contributes to the better efficiency and scalability of mspp. This is specially sensible when the computation is coarse grain and the communication can be masked by computation.

v) In general, for fine grained computation shared memory based implementation achieves better performance. As the computation grain increases message passing based implementation is an interesting choice.

6.2 Future Work

This work addresses the most part of the implementation of the skeleton framework and presents experimentation in stream parallel patterns. However there are some issues that we want to put forward to expand the work for better and optimized use.

First, some of the interesting futures in Fastflow has to be adopted in this work. This includes developing a custom way for a programmer to define different scheduling mechanism, a way to make the farm-collector an optional node.
In addition to this we also want mention the use of streaming of item from a primitive sources. The Current version only supports stream generation programmatically from inside the framework.
All those future work should still take care of the abstraction of the underlying distribution language (MPI) as much as possible.

Finally, even-though this work scoped in stream parallel skeleton framework, it can easily be expanded to also include data parallel skeleton patterns. Thanks to the power of MPI, this can be achieved by few programming effort. However, we still underline the consideration of abstraction from MPI while developing those new skeleton patterns.

Bibliography

[1] Murray Cole. *Algorithmic Skeletons: structured management of parallel computation*, MIT Press, Cambdridge, MA, USA, 1989

[2] Cole M. *Algorithmic skeletons: A structured approach to the management of parallel computation*. PhD Thesis, University of Edinburgh, Computer Science Dpt, Edinburgh 1988

[3] Cole M. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing, Pitman/MIT Press: London, 1989

[4] Horacio Gonzalez-Velez and Mario Leyton *A survey of algorithmic skeleton frameworks: high-level programming enablers Software* : Practice and Experience Volume 40, Issue 12, pages 1135-1160, November/December 2010.

[5] eSkel : The Endumburg Skeleton Library. http://homepages.inf.ed.ac.uk/mic/eSkel/ (Last accessed: 20 January 2013)

[6] T. A. Ngo and L. Snyder, ON the Influence of Programming Models on Shared Memory Computer Performance, Scalable High Performance Computing Conf. (April 1992).

[7] T. LeBlanc and E. Markatos, Shared Memory vs. Message Passing in Shared Memory Multiprocessor, Fourth SPDP (1992)

[8] MPI-Forum. The official message passing interface standard. web site 2007. http://www.mpi-forum.org (Last accessed: 20 January 2013)

[9] Shared memory configuration for open-MPI http://www.openMPI.org/faq/?category=sm#poor-sm-btl-performance (Last accessed: 20 January 2013)

[10] M. Danelutto *Distributed systems : paradigm and models*. March 2, 2012

[11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *FastFlow: high-level and efficient streaming on multi-core, in: Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing*, chapter 13. Wiley, 201

[12] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: *A Structured High level programming language and its structured support*. Concurrency Practice and Experience, 7(3):225255, May 1995

[13] A. Benoit, M. Cole, S. Gilmore, J. Hillston *Using eSkel to implement the Multiple Baseline Stereo Application.* published in Parallel Computing: *Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005.* John von Neumann Institute for Computing

[14] ImageMagick library: http://www.imagemagick.org/script/index.php (Last accessed: 20 January 2013)

[15] Marco Vanneschi. Courses note of *High Performance Computing Systems and Enabling Platforms* Department of Computer Science, University of Pisa, 2011

[16] Steven Cameron Woo, Jaswinder Pal Singh and John L. Hennessy. *The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors.* Computer Systems Laboratory Stanford University Stanford, CA 94305

[17] 2012/10/08 - Release 1.0.0 of KNEM, the high-performance intra-node MPI

[18] KNEM, High-Performance Intra-Node MPI Communication. http://runtime.bordeaux.inria.fr/knem/ (Last accessed: 20 January 2013)

[19] ASSIST as a Research Framework for High-performance Grid Programming Environments (2004)

[20] Marco Aldinucci , Marco Danelutto , Marco Vanneschi , Corrado Zoccolo *Dynamic shared data in structured parallel programming frameworks*, M. Aldinucci. PhD. thesis, Computer Science Department, University of Pisa, Dec. 2003

[21] Yong Luo Scientific Computing Group CIC-19 Los Alamos National Laboratory *Shared Memory vs. Message Passing: the COMOPS Benchmark Experiment*

[22] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui. *Shared Memory vs Message Passing.* December 2, 2003

[23] http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about (Last accessed: 20 January 2013).

[24] Thomas J. leBlanc, Evangelos P. Markatos *Shared Memory vs message Passing in Shared-Memory Multiprocessors.* Computer Science Department, University of Rochester.

[25] Marco Aldinucci*, Massimo Torquati, Massimiliano Meneghin. *FastFlow: Effcient Parallel Streaming Applications on Multi-core.* September 2, 2009.

[26] *The Pisa Parallel Programming Language* http://www.di.unipi.it/ susanna/p3lintro.html (Last accessed 20 January 2013)

[27] Mario Leyton and Jose M. Piquer. *Skandium: Multi-core Programming with Algorithmic Skeletons Universidad de Chile, Santiago, Chile.*

Appendix A

Source Code

Listing 6.1: Pipeline Skeleton Interface

```
1  #include <sys/time.h>
2
3  int MPI_Parallel_pipe(p_stage *stages, int ns, int task_item_num, void*
       input_buff,
4       void* output_buff, int in_item_size, int out_item_size, Pipe_type
           pty)
5  {
6    pipe_t pipet;
7    pipe_t *pt = &pipet;
8    pt->isOdering = pty;
9    if (myWRank() == 0)
10     {
11       pt->input_buff = input_buff;
12     }
13   if (myWRank() == getWSize() - 1)
14     {
15       pt->output_buff = output_buff;
16     }
17
18   // First Inistantiate the skeleton.
19   MPI_Pipeline_init(stages, ns, in_item_size, out_item_size, pty, pt);
20
21   // First start the Stream Consumer.
22   if (myWRank() == getWSize() - 1)
23     {
24       MPI_Status status;
25       if (isPrint)
26         printf("External Stream Collector .... Starts \n");
27       MPI_Recv(pt->output_buff, pt->out_byte, MPI_BYTE, pt->final_stId,
28           MPI_ANY_TAG, pt->parentComm, &status);
29       int tag = status.MPI_TAG;
30       while (tag != SHUTDOWN_TAG)
31         {
32           pt->output_buff = pt->output_buff + pt->in_byte;
33           MPI_Recv(pt->output_buff, pt->out_byte, MPI_BYTE, pt->
```

```
            final_stId,
34              MPI_ANY_TAG, pt->parentComm, &status);
35          tag = status.MPI_TAG;
36        }
37
38      // get the starting time stamp and calculate the completion time.
39      double start;
40      static struct timeval tv_end =
41        { 0, 0 };
42      gettimeofday(&tv_end, NULL);
43      MPI_Recv(&start, sizeof(double), MPI_BYTE, 0, TIME_STAMP_INFO,
44          MPI_COMM_WORLD, &status);
45
46      double end = tv_end.tv_sec * 1000000 + tv_end.tv_usec;
47      double alltimeSpent = (end - start) / 1000;
48      printf("Elapsed time = %.2f millisecond \n\n", alltimeSpent);
49      if (isPrint)
50        printf("External Stream Collector .. ends\n");
51      printf("The Skeleton shutdown successfully! \n");
52    } //End of stream consumer.
53
54  // Start Pipeline (and my be its composed skeletons)
55  MPI_Pipe_start(pt);
56
57  // At Last Start the task Streamer
58  if (myWRank() == 0)
59    {
60      int flag, tag_max_size;
61      void* tagmax;
62      MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &tagmax, &flag);
63
64      //if can not read flag, use the default one specified in MPI
            specification:  32767
65      if (!flag)
66        {
67          tag_max_size = 32767;
68        }
69      else
70        {
71          tag_max_size = *(int *) tagmax;
72        }
73      int tag = TAG_START_NUM;
74      int j;
```

```
75        if (isPrint)
76          printf("External Streamer .. Starts %d \n", pt->ini_stId);
77
78        // Read time stamp before start generating stream
79        static struct timeval tv_start =
80          { 0, 0 };
81        gettimeofday(&tv_start, NULL);
82        double sec = tv_start.tv_sec * 1000000 + tv_start.tv_usec;
83
84        for (j = 0; j < task_item_num; j++)
85          {
86            // Send to the initial stage, it can be a simple pipe-stage or
                  Emitter of a FARM.
87            MPI_Send(pt->input_buff, pt->in_byte, MPI_BYTE, pt->ini_stId,
                 tag,
88                  pt->parentComm);
89
90            // To advance the pointer, increase by size of item.
91            pt->input_buff = pt->input_buff + pt->in_byte;
92            ++tag;
93            if (tag >= tag_max_size)
94              {
95                tag = TAG_START_NUM;
96              }
97          }
98
99        // Send Shutdown Tag.
100       MPI_Send(pt->input_buff, pt->in_byte, MPI_BYTE, pt->ini_stId,
101           SHUTDOWN_TAG, pt->parentComm);
102
103       //Send time stamp used to measure the over all complition time
104       MPI_Send(&sec, sizeof(double), MPI_BYTE, getWSize() - 1,
              TIME_STAMP_INFO,
105           MPI_COMM_WORLD);
106
107       if (isPrint)
108         printf("Streamer .. Finishes\n");
109     } //End of streamer.
110 }
```

Listing 6.2: Pipeline Skeleton Main Manager

```
1
2  int MPI_Pipeline_init(p_stage *stages, int ns, int in_item_size, int
       out_item_size,
3       Pipe_type pty, pipe_t *pt)
4  {
5
6    // Fill the pipeline strature.
7    pt->parentComm = MPI_COMM_WORLD;
8    pt->ns = ns;
9
10   //get the pipeine stages
11   pt->stage = stages;
12   pt->parentComm_sz = getWSize();
13
14   // Two processes are already taken for external streamer and External
         streamer collector.
15   int expected_procs = 2;
16   int j;
17   int check_fp = 0;
18
19   // Calculate required number of processes.
20   for (j = 0; j < pt->ns; j++)
21     {
22       if (pt->stage[j].stg_type == SEQ_STAGE)
23         {
24           expected_procs += pt->stage[j].stg.pipe.parl_degree;
25         }
26
27       else if (pt->stage[j].stg_type == FRM_STAGE)
28         {
29           //if Farm consider also Emitter and Collector.
30           expected_procs += pt->stage[j].stg.pipe.parl_degree + 2;
31         }
32       else if (pt->stage[j].stg_type == FP_STAGE)
33         {
34           check_fp = 1;
35           expected_procs += (pt->stage[j].stg.farm.parl_degree
36               * pt->stage[j].stg.farm.worker.p_worker.ns) + 2;
37         }
38       else if (pt->stage[j].stg.pipe.parl_degree < 1)
39         {
40           printf("ERROR: Parallelism degree at stage %d can not be less
                 than 1",
```

```
41              (j + 1));
42          MPI_Finalize();
43          exit(-128);
44        }
45    }
46
47  // Process which have rank 1 at World level is always responsible to
        take input stream.
48  pt->ini_stId = 1;
49  if (pt->stage[ns - 1].stg_type == FRM_STAGE)
50    {
51      pt->final_stId = expected_procs - 2
52          - pt->stage[pt->ns - 1].stg.pipe.parl_degree;
53    }
54  else if (pt->stage[ns - 1].stg_type == FP_STAGE)
55    {
56      pt->final_stId = expected_procs - 2
57          - (pt->stage[pt->ns - 1].stg.pipe.parl_degree
58              * pt->stage[j].stg.farm.worker.p_worker.ns);
59    }
60  else
61    {
62      pt->final_stId = pt->parentComm_sz - 2;
63    }
64
65  pt->isOdering = pty;
66  pt->in_byte = in_item_size;
67  pt->out_byte = out_item_size;
68
69  //check if number of Pipeline stage is greater than two
70  if (pt->ns < 2)
71    {
72      if (myWRank() == 0)
73        printf(
74            "ERROR: too few stage. There should be at least two pipeline
                stages \n");
75      MPI_Finalize();
76      exit(-128);
77    }
78
79  if (pt->parentComm_sz < expected_procs)
80    {
81      if (myWRank() == 0)
```

```
82          printf(
83              "ERROR: too few process. Please run this program with %d
                    processes\n",
84              expected_procs);
85        MPI_Finalize();
86        exit(-128);
87      }
88    else if (pt->parentComm_sz > expected_procs)
89      {
90        if (myWRank() == 0)
91          printf(
92              "ERROR: too many process. Please run this program with %d
                    processes only \n",
93              expected_procs);
94        MPI_Finalize();
95        exit(-128);
96      }
97
98    // From 'workers' create nw*ns stages duplicating the single stage
        instance passed in workers.
99    pstage_t *stgs;
100   if (check_fp)
101     {
102       int j;
103       for (j = 0; j < pt->ns; j++)
104         {
105           if (pt->stage[j].stg_type == FP_STAGE)
106             {
107               stgs = malloc(
108                   sizeof(pstage_t) * pt->stage[j].stg.farm.parl_degree
109                       * pt->stage[j].stg.farm.worker.p_worker.ns);
110               int i;
111               int w_index = 0;
112               int inbyte, outbyte;
113               void*
114               (*func)(void*);
115               for (i = 0;
116                   i
117                       < (pt->stage[j].stg.farm.parl_degree
118                           * pt->stage[j].stg.farm.worker.p_worker.ns); i
                                ++)
119                 {
120                   stgs[i].parl_degree = 1;
```

```
121                     func =
122                         pt->stage[j].stg.farm.worker.p_worker.stage[
                               w_index].f_ptr;
123                     stgs[i].f_ptr = func;
124                     inbyte =
125                         pt->stage[j].stg.farm.worker.p_worker.stage[
                               w_index].in_count;
126                     stgs[i].in_count = inbyte;
127                     outbyte =
128                         pt->stage[j].stg.farm.worker.p_worker.stage[
                               w_index].out_count;
129                     stgs[i].out_count = outbyte;
130                     w_index++;
131                     if (w_index >= pt->stage[j].stg.farm.worker.p_worker.
                           ns)
132                       {
133                         w_index = 0;
134                       }
135                   }
136               }
137           }
138       }
139
140   // Fill more information for the pipe stages.
141   processStage(pt, stgs);
142
143   // Now Create the ns Communiators
144   MPI_Comm_split(pt->parentComm, pt->color, pt->key, &pt->st_comm);
145   // start the Pipe in Farm
146   if (check_fp)
147     {
148       for (j = 0; j < pt->ns; j++)
149         {
150           if (pt->stage[j].stg_type == FP_STAGE)
151             {
152               //Build farm_fp_t.
153               farm_fp_t frmt;
154               fillFarmStruct(&frmt, pt, j);
155               MPI_Farm_start_fp(&frmt, stgs);
156             }
157         }
158     }
159
```

```
160   // Successful completion
161   return 0;
162 }
```

Listing 6.3: Pipeline Helper Class

```
1
2  //iterate over all stage and fill more info
3  int processStage(pipe_t *pt, pstage_t *stages)
4  {
5
6    if (myWRank() == 0 || myWRank() == getWSize() - 1)
7      {
8        pt->color = pt->ns; // we can't have more than ns stages
9        pt->key = 0;
10       pt->used_processes = getWSize(); //it doesn't matter
11     }
12
13   //all the following should exclude external processes (i.e 0 & size-1)
14   else
15     {
16       pt->used_processes = 1; //keeps track of used processes. NOTE: one
             processe are already taken for Streamer
17       int i;
18       int isBeforeFarm = 0;
19       for (i = 0; i < pt->ns; i++)
20         {
21           //please Don't confuse color of a stage with rank or process
22           //in_rank and out_rank are MPI process ranks to communicate to
                 neighhboring stages,
23           //while stage[i] is used to fill information for a stage with
                color i.
24
25           if (i > 0 && pt->stage[i - 1].stg_type == FRM_STAGE && myWRank
                () != 0
26               && myWRank() != getWSize() - 1)
27             {
28               pt->stage[i].stg.pipe.in_rank = pt->used_processes + 1;
29               pt->used_processes = pt->used_processes
30                   + pt->stage[i - 1].stg.pipe.parl_degree + 2;
31               isBeforeFarm = 1;
32               pt->stage[i].stg.pipe.out_rank = pt->used_processes + 1;
```

```
33              }
34            else if (i > 0 && pt->stage[i - 1].stg_type == FP_STAGE
35                && myWRank() > pt->used_processes && myWRank() != 0
36                && myWRank() != getWSize() - 1)
37              {
38                pt->stage[i].stg.pipe.in_rank = pt->used_processes + 1;
39                pt->used_processes = pt->used_processes
40                    + (pt->stage[i - 1].stg.farm.parl_degree
41                        * pt->stage[i - 1].stg.farm.worker.p_worker.ns) +
                          2;
42                isBeforeFarm = 1;
43                pt->stage[i].stg.pipe.out_rank = pt->used_processes + 1;
44              }
45
46            if (pt->stage[i].stg_type == SEQ_STAGE)
47              { //it is then a sequential pipeline
48                if (myWRank() == pt->used_processes)
49                  {
50                    if (i == pt->ns - 1 && !isBeforeFarm)
51                      { //last Stage
52                        pt->stage[i].stg.pipe.in_rank = pt->used_processes
                              - 1;
53                        pt->stage[i].stg.pipe.out_rank = getWSize() - 1;
54                      }
55                    else if (i >= 0 && i < pt->ns - 1 && !isBeforeFarm)
56                      { //First or intermediate stages
57                        pt->stage[i].stg.pipe.in_rank = pt->used_processes
                              - 1;
58                        pt->stage[i].stg.pipe.out_rank = pt->
                              used_processes + 1;
59                      }
60
61                    //set color for each process
62                    pt->color = i;
63
64                    //for sequential stage we don't have to worry about
                          process rank in new communicator
65                    pt->key = 0;
66                  }
67                pt->used_processes++; // for sequential stage only one
                      process is used
68                isBeforeFarm = 0;
69              }
```

```
70
71              //Nested Farm in a pipeline-stage
72              else if (pt->stage[i].stg_type == FRM_STAGE)
73                {
74                  // give range of processes: [used_processes -
                        used_processes+ns+2] to a Farm stage
75                  if (myWRank() >= pt->used_processes
76                      && myWRank()
77                          < pt->used_processes + pt->stage[i].stg.pipe.
                              parl_degree
78                           + 2)
79                    {
80                      //fill farm structure patially before calling MPI_Farm
81                      farm_t frm;
82                      frm.used_processes = pt->used_processes;
83                      frm.isLib = true; //set this when you call farm from a
                            library
84                      frm.color = i;
85                      frm.isPreceedingFarm = isBeforeFarm;
86                      generic_fp_worker wrkr;
87                      wrkr.f_worker.task = pt->stage[i].stg.pipe.f_ptr;
88                      if (isBeforeFarm)
89                        frm.preceedingParDeg =
90                            pt->stage[i - 1].stg.pipe.parl_degree;
91                      Farm_type ftyp = FARM_No_Ord;
92
93                      //call FarmSkeleton
94                      MPI_Farm_init(pt->stage[i].stg.pipe.parl_degree, wrkr,
95                          pt->stage[i].stg.pipe.in_count,
96                          pt->stage[i].stg.pipe.out_count, ftyp, &frm);
97
98                      //get update from number of used processes, key and
                            value, returned from MPI_Farm
99                      pt->used_processes = frm.used_processes;
100                     pt->color = frm.color;
101                     pt->key = frm.key;
102
103                     //And update the whole farm struct. till now we work
                            as ordinary stage,
104                     //Now since we know that it is a farm, we replace the
                            stage with color i to be a farm.
105                     pt->stage[i].stg.farm = frm;
106                     isBeforeFarm = 0;
```

```
107                 }
108               }
109             else if (pt->stage[i].stg_type == FP_STAGE)
110               {
111                 if (myWRank() >= pt->used_processes
112                     && myWRank()
113                         < pt->used_processes
114                             + (pt->stage[i].stg.farm.parl_degree
115                                 * pt->stage[i].stg.farm.worker.p_worker.ns
                                        ) + 2)
116                   {
117                     //fill farm structure patially before calling MPI_Farm
118                     farm_fp_t frmt;
119                     frmt.used_processes = pt->used_processes;
120                     frmt.isLib = true; //set this when you call farm from
                            a library
121                     frmt.isPreceedingFarm = isBeforeFarm;
122                     if (isBeforeFarm && pt->stage[i - 1].stg_type ==
                            FP_STAGE)
123                       {
124                         frmt.preceedingParDeg =
125                             (pt->stage[i - 1].stg.farm.parl_degree
126                                 * pt->stage[i - 1].stg.farm.worker.
                                        p_worker.ns);
127                       }
128                     else if (isBeforeFarm)
129                       {
130                         frmt.preceedingParDeg =
131                             pt->stage[i - 1].stg.pipe.parl_degree;
132                       }
133                     Farm_type ftyp = FARM_No_Ord;
134                     frmt.frm_type = ftyp;
135                     frmt.next_worker = pt->stage[i].stg.farm.worker.
                            p_worker.ns;
136                     frmt.parl_degree = pt->stage[i].stg.farm.parl_degree;
137                     mpiFarmHelper_fp(&frmt, stages, i);
138
139                     //get update from number of used processes, key and
                            value, returned from MPI_Farm
140                     pt->used_processes = frmt.used_processes;
141                     pt->color = frmt.color;
142                     pt->key = frmt.key;
143
```

```
144                     //Now build the farm struct to send back
145                     farm_t frm;
146                     frm.used_processes = frmt.used_processes;
147                     frm.isLib = true; //set this when you call farm from a
                            library
148                     frm.color = frmt.color;
149                     frm.frm_type = frmt.frm_type;
150                     frm.emt_rank = frmt.emt_rank;
151                     frm.coll_rank = frmt.coll_rank;
152                     frm.key = frmt.key;
153                     frm.parl_degree = frmt.parl_degree;
154                     frm.identifier_Flag = frmt.identifier_Flag;
155                     frm.start_rank = frmt.start_rank;
156                     frm.end_rank = frmt.end_rank;
157                     frmt.in_byte = pt->stage[i].stg.farm.in_byte;
158                     frmt.out_byte = pt->stage[i].stg.farm.out_byte;
159                     frm.next_worker = pt->stage[i].stg.farm.worker.
                           p_worker.ns;
160                     pt->stage[i].stg.farm = frm;
161                     isBeforeFarm = 0;
162                 }
163             }
164         } //end of the outer for-loop
165     }
166 }
167
168 // Fill farm struct driving from pipe struct.
169 int fillFarmStruct(farm_fp_t *frmt, pipe_t *pt, int j)
170 {
171   frmt->used_processes = pt->stage[j].stg.farm.used_processes;
172   frmt->isLib = true;
173   frmt->color = pt->stage[j].stg.farm.color;
174   frmt->frm_type = pt->stage[j].stg.farm.frm_type;
175   frmt->key = pt->stage[j].stg.farm.key;
176   frmt->fcomm = pt->parentComm;
177   frmt->parentComm = pt->parentComm;
178   frmt->parl_degree = pt->stage[j].stg.farm.parl_degree;
179   frmt->identifier_Flag = pt->stage[j].stg.farm.identifier_Flag;
180   frmt->emt_rank = pt->stage[j].stg.farm.emt_rank;
181   frmt->coll_rank = pt->stage[j].stg.farm.coll_rank;
182   frmt->start_rank = pt->stage[j].stg.farm.start_rank;
183   frmt->end_rank = pt->stage[j].stg.farm.end_rank;
184   frmt->in_byte = pt->stage[j].stg.farm.in_byte;
```

```
185   frmt->out_byte = pt->stage[j].stg.farm.out_byte;
186   frmt->worker.p_worker.ns = pt->stage[j].stg.farm.next_worker;
187   return 0;
188 }
```

Listing 6.4: Pipeline Low Level communication and computation

```
1
2  //Low level computation of a sequential stage and communication with
       neighboring stage
3
4  int MPI_Pipe_seq(pipe_t *pt, int color)
5  {
6
7    // Check if the process belong to this color/stage.
8    if (pt->color == color)
9      {
10       //If the pipeline is ordering and if this stage is last stage,
             call item ordering stage
11       if (myRank(pt->parentComm) == pt->final_stId && pt->isOdering ==
             PIPE_Ord)
12         {
13           item_Ordering_Stage(pt->stage[pt->color].stg.pipe.in_rank,
14               pt->stage[pt->color].stg.pipe.out_rank, pt->in_byte, pt->
                    out_byte,
15               pt->stage[pt->color].stg.pipe.f_ptr, pt->parentComm);
16         }
17       else
18         {
19           MPI_Request request;
20           MPI_Status status, snd_status;
21           void* rcv;
22           void* snd;
23           rcv = malloc(pt->in_byte);
24           snd = malloc(pt->out_byte);
25
26           if (isPrint)
27             printf("Stage with color %d ...  starts  \n", color);
28
29           // Receive once to enable Non-blocking Send.
30           MPI_Recv(rcv, pt->in_byte, MPI_BYTE,
31               pt->stage[pt->color].stg.pipe.in_rank, MPI_ANY_TAG,
```

```
32                  pt->parentComm, &status);
33              snd = pt->stage[pt->color].stg.pipe.f_ptr(rcv);
34              int tag = status.MPI_TAG;
35              while (tag != SHUTDOWN_TAG)
36                {
37                  MPI_Isend(snd, pt->out_byte, MPI_BYTE,
38                      pt->stage[pt->color].stg.pipe.out_rank, tag, pt->
                            parentComm,
39                      &request);
40
41                  // Do some usefull work, to overlap it with communication.
42                  MPI_Recv(rcv, pt->in_byte, MPI_BYTE,
43                      pt->stage[pt->color].stg.pipe.in_rank, MPI_ANY_TAG,
44                      pt->parentComm, &status);
45                  snd = pt->stage[pt->color].stg.pipe.f_ptr(rcv);
46                  tag = status.MPI_TAG;
47
48                  // At this point check if the send completes.
49                  MPI_Wait(&request, &snd_status);
50                }
51
52              // Now Send Shutdown Tag
53              MPI_Send(snd, pt->out_byte, MPI_BYTE,
54                  pt->stage[pt->color].stg.pipe.out_rank, SHUTDOWN_TAG,
55                  pt->parentComm);
56          }
57        if (isPrint)
58          printf("Stage with color %d ends \n", color);
59      } // End of check if a process belong to this color.
60  }
```

Listing 6.5: Pipeline skeleton starter

```
1  // MPI pipeline skeleton Start.
2
3  int MPI_Pipe_start(pipe_t *pt)
4  {
5
6    // Composition is achived by calling the skeletons from left to right.
7    // First the stream comsumer, then the last stage, the second last, ..
         etc.
8
```

```
9     // Last stage, the second last, ... etc.
10    int i;
11    for (i = pt->ns - 1; i >= 0; i--)
12      {
13        if (pt->stage[i].stg_type == SEQ_STAGE)
14          {
15            // Call Sequential stage.
16            MPI_Pipe_seq(pt, i);
17          }
18        else if (pt->stage[i].stg_type == FRM_STAGE)
19          {
20            pt->stage[i].stg.farm.fcomm = pt->st_comm;
21            pt->stage[i].stg.farm.isLib = true;
22
23            // Call MPI Farm Start.
24            MPI_Farm_start(&pt->stage[i].stg.farm);
25          }
26      }
27
28  }
```

Listing 6.6: Farm skeleton Interface

```
1  #include <sys/time.h>
2  int MPI_Farm(int nw, generic_fp_worker workers, Worker_type w_type, int
       in_byte,
3       int out_byte, int item_num, void* input_buff, void* output_buff,
4       Farm_type fty)
5  {
6    farm_t frmt;
7    farm_t *frm = &frmt;
8
9    // Input and output Buffers are filled only by the streamer and stream
         consumer respectively.
10   if (myWRank() == 0)
11     {
12       frm->input_buff = input_buff;
13     }
14   if (myWRank() == getWSize() - 1)
15     {
16       frm->output_buff = output_buff;
17     }
```

```
18
19    // Not library call.
20    frm->isLib = false;
21
22    if (w_type == SEQ_WORKER)
23      {
24        MPI_Farm_init(nw, workers, in_byte, out_byte, fty, frm);
25      }
26    else
27      {
28        MPI_Farm_fp(nw, workers, w_type, in_byte, out_byte, item_num,
              input_buff,
29              output_buff, fty);
30        return 0;
31      }
32
33    //Create Communicator for Farm
34    MPI_Comm_split(frm->parentComm, frm->color, frm->key, &frm->fcomm);
35
36    MPI_Status status;
37
38    // First start the Stream Consumer.
39    if (myWRank() == getWSize() - 1)
40      {
41        MPI_Status status;
42        if (isPrint)
43          printf("External Stream Collector .... Starts \n");
44        MPI_Recv(frm->output_buff, frm->out_byte, MPI_BYTE, frm->coll_rank
              ,
45              MPI_ANY_TAG, frm->parentComm, &status);
46        int tag = status.MPI_TAG;
47        while (tag != SHUTDOWN_TAG)
48          {
49            // Advance the pointer, increase by size of item received.
50            frm->output_buff = frm->output_buff + frm->out_byte;
51            MPI_Recv(frm->output_buff, frm->out_byte, MPI_BYTE, frm->
                  coll_rank,
52                  MPI_ANY_TAG, frm->parentComm, &status);
53            tag = status.MPI_TAG;
54          }
55
56        long double start;
57        static struct timeval tv_end =
```

```
58          { 0, 0 };
59        gettimeofday(&tv_end, NULL);
60        //clock_t *strt = malloc(sizeof(clock_t));
61        MPI_Recv(&start, sizeof(long double), MPI_BYTE, 0, TIME_STAMP_INFO
              ,
62            MPI_COMM_WORLD, &status);
63
64        long double end = tv_end.tv_sec * 1000000 + tv_end.tv_usec;
65        double alltimeSpent = (end - start) / 1000;
66        printf("Elapsed time = %.2f millisecond \n\n", alltimeSpent);
67        if (isPrint)
68          printf("External Stream Collector .. ends\n");
69        printf("The Skeleton shutdown successfully! \n");
70      }
71
72    // Strart the Farm skeleton and its compositions.
73    MPI_Farm_start(frm);
74
75    //Now start the streamer.
76    // At last the task Streamer.
77    if (myWRank() == 0)
78      {
79        int flag, tag_max_size;
80        void* tagmax;
81        MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &tagmax, &flag);
82
83        //if can not read flag, use the default one specified in MPI
              specification:  32767
84        if (!flag)
85          {
86            tag_max_size = 32767;
87          }
88        else
89          {
90            tag_max_size = *(int *) tagmax;
91          }
92        int tag = TAG_START_NUM;
93        int j;
94        if (isPrint)
95          printf("External Streamer ......AT FARM.... Starts \n");
96
97        //Register initial Time stamp.
98        static struct timeval tv_start =
```

```
99          { 0, 0 };
100       gettimeofday(&tv_start, NULL);
101       long double sec = tv_start.tv_sec * 1000000 + tv_start.tv_usec;
102       for (j = 0; j < item_num; j++)
103         {
104           MPI_Send(frm->input_buff, frm->in_byte, MPI_BYTE, frm->
                  emt_rank, tag,
105                    frm->parentComm);
106
107           // To advance the pointer, increase by size of item.
108           frm->input_buff = frm->input_buff + frm->in_byte;
109           ++tag;
110           if (tag >= tag_max_size)
111             {
112               tag = TAG_START_NUM;
113             }
114         }
115       if (isPrint)
116         printf("Streamer .. Finishes\n");
117
118       //Then send a shutdown Tag.
119       MPI_Send(frm->input_buff, frm->in_byte, MPI_BYTE, frm->emt_rank,
120           SHUTDOWN_TAG, frm->parentComm);
121       //Send time stamp used to measure the over all complition time
122       MPI_Send(&sec, sizeof(long double), MPI_BYTE, getWSize() - 1,
123           TIME_STAMP_INFO, frm->parentComm);
124     }
125 }
```

Listing 6.7: Farm Skeleton Main Manager

```
1
2  // Farm Skeleton.
3  int MPI_Farm_init(int nw, generic_fp_worker workers, int in_byte, int
       out_byte,
4      Farm_type fty, farm_t *fptr)
5  {
6
7    // Fill Farm Struct.
8    fptr->parentComm = MPI_COMM_WORLD;
9    fptr->parl_degree = nw;
10   fptr->worker = workers;
```

```
11    fptr->in_byte = in_byte;
12    fptr->out_byte = out_byte;
13    fptr->frm_type = fty;
14
15    // Check if Farm is called from a library for neasting or directly by
          a user.
16    if (!fptr->isLib)
17      {
18        // One process is already used for Streamer.
19        fptr->used_processes = 1;
20        if (myWRank() == 0 || myWRank() == getWSize() - 1)
21          {
22            fptr->color = getWSize() + 1;
23            fptr->key = 0;
24          }
25        else
26          {
27            fptr->color = 0;
28          }
29
30        if (fptr->parl_degree < 2)
31          {
32            if (myWRank() == fptr->used_processes)
33              printf("ERROR: at least two Workers are needed for FARM \n")
                    ;
34            MPI_Finalize();
35            exit(-128);
36          }
37
38        // Two processes are already taken for external streamering &
              collecting.
39        int expected_procs = 2;
40        expected_procs += nw + 2;
41        if (getWSize() < expected_procs)
42          {
43            if (myWRank() == fptr->used_processes)
44              printf(
45                  "ERROR: No enough process to start FARM with %d Workers.
                        %d processes should be started \n",
46                  nw, expected_procs);
47            MPI_Finalize();
48            exit(-128);
49          }
```

```
50        else if (getWSize() > expected_procs)
51          {
52            if (myWRank() == fptr->used_processes)
53              printf(
54                  "ERROR: too many process. Please run this program with %
                       d processes only \n",
55                  expected_procs);
56            MPI_Finalize();
57            exit(-128);
58          }
59      }
60
61    // Fill more information about Farm skeleton.
62    processFarm(fptr, fptr->color);
63
64    //to indicate successfull completion.
65    return 0;
66  }
```

Listing 6.8: Farm Helper Class

```
1   // MPI_Farm Helper.
2   processFarm(farm_t *frm, int color)
3   {
4     if (frm->parl_degree > 1)
5       {
6
7         // Choose only ns + 2 processes for a Farm stage.
8         if ((myWRank() >= frm->used_processes
9             && myWRank() < frm->used_processes + frm->parl_degree + 2)
10            || myWRank() == 0 || myWRank() == getWSize() - 1)
11          {
12            int k, emtr, coll;
13
14            // ADD number_of_Workers + Emitter + Collector process to same
                  color.
15            for (k = 0; k < frm->parl_degree + 2; k++)
16              {
17
18                // Choose the first process for Emitter task.
19                if (k == 0 && myWRank() == frm->used_processes)
20                  {
```

```
21                  if (frm->isLib && frm->isPreceedingFarm)
22                    {
23                      frm->in_rank = frm->used_processes - 1
24                          - frm->preceedingParDeg;
25                    }
26                  else
27                    {
28                      frm->in_rank = frm->used_processes - 1;
29                    }
30
31                  // flag this process for Emitter task.
32                  frm->identifier_Flag = 1;
33                  frm->key = k;
34                }
35
36              // If it is the last stage, the collector has to direct
                    the output item to a channel.
37              else if (k == 1 && myWRank() == frm->used_processes + 1)
38                {
39                  // Worker Flag.
40                  frm->identifier_Flag = 2;
41                  frm->out_rank = frm->used_processes + frm->parl_degree
                        + 2;
42                  frm->key = k;
43                }
44
45              if (k > 1 && myWRank() == frm->used_processes + k)
46                { //the rest are Workers
47                  frm->identifier_Flag = 3;
48                  frm->key = k;
49                }
50              if (k == 0)
51                {
52                  // This is used to tell rank of Emitter to a Worker
                        processes.
53                  emtr = frm->used_processes;
54                  frm->emt_rank = emtr;
55                }
56              if (k == 1)
57                {
58                  //this is used to tell rank of Collector to a Worker
                        processes
59                  coll = frm->used_processes + 1;
```

```
60                    frm->coll_rank = coll;
61                  }
62                // specify color for each farm processes
63                frm->color = color;
64              }
65
66            // Start range processes used for this Farm.
67            frm->start_rank = frm->used_processes;
68
69            // Increased used process by number of Workers+2.
70            frm->used_processes += frm->parl_degree + 2;
71
72            //start range processes used for this Farm
73            frm->end_rank = frm->used_processes;
74          }
75      }
76  }
```

Listing 6.9: Farm Low Level communication and computation

```
1   // Farm Compute.
2   int MPI_Farm_compute(farm_t *frm)
3   {
4
5     int myFRank = myRank(frm->fcomm);
6
7     // Check if this process belong to this Farm.
8     if (myWRank() >= frm->start_rank && myWRank() < frm->end_rank)
9       {
10        MPI_Request request;
11        MPI_Status status, snd_status;
12
13        // EMITTER.
14        if (myFRank == 0 && frm->identifier_Flag == 1)
15          {
16
17            void* rcv;
18            rcv = malloc(frm->in_byte);
19            int next_worker = 2;
20
21            // Receive once to enable Non-bloking operation.
22            MPI_Recv(rcv, frm->in_byte, MPI_BYTE, frm->in_rank,
```

```
                MPI_ANY_TAG,
23                frm->parentComm, &status);
24            int tag = status.MPI_TAG;
25
26            if (isPrint)
27              printf("Farm Emitter ... starts \n");
28            while (tag != SHUTDOWN_TAG)
29              {
30                MPI_Isend(rcv, frm->out_byte, MPI_BYTE, next_worker, tag,
31                    frm->fcomm, &request);
32
33                // Do some usefull work, to overlap it with communication.
34                MPI_Recv(rcv, frm->in_byte, MPI_BYTE, frm->in_rank,
                      MPI_ANY_TAG,
35                    frm->parentComm, &status);
36                next_worker = getNextworker(next_worker, frm->parl_degree
                      + 1);
37                tag = status.MPI_TAG;
38
39                // At this point check if the send completes.
40                MPI_Wait(&request, &snd_status);
41              }
42
43            // TODO change the following to a broad cast operation.
44            int i = 0;
45            next_worker = 2;
46            for (i = 0; i < frm->parl_degree; i++)
47              {
48                MPI_Send(rcv, frm->out_byte, MPI_BYTE, next_worker,
                      SHUTDOWN_TAG,
49                    frm->fcomm);
50                next_worker = getNextworker(next_worker, frm->parl_degree
                      + 1);
51              }
52            if (isPrint)
53              printf("Farm Emitter ... ends\n");
54          }
55
56        // WORKERS.
57        else if (myFRank >= 2 && myFRank < (frm->parl_degree + 2)
58            && frm->identifier_Flag == 3)
59          {
60
```

```
61            void* rcv;
62            void* snd;
63            rcv = malloc(frm->in_byte);
64            snd = malloc(frm->out_byte);
65            int tag;
66            int isItemReceived = 0; // used to implement Non-bloacking
                 operation in Workers
67            int isSendPending = 0;
68            int s;
69
70            while (true)
71              {
72                if (isItemReceived && tag != SHUTDOWN_TAG)
73                  {
74                    // Workers always communicate to process 1 ->
                         Collector
75                    MPI_Isend(rcv, frm->out_byte, MPI_BYTE, 1, tag, frm->
                         fcomm,
76                         &request);
77                    isSendPending = 1;
78                  }
79
80                //Do some usefull work.
81                // Workers always Receive from process   0 -> Emitter.
82                MPI_Recv(rcv, frm->in_byte, MPI_BYTE, 0, MPI_ANY_TAG, frm
                     ->fcomm,
83                     &status);
84                tag = status.MPI_TAG;
85
86                // if a Shutdown tag is received, send shutdown tag and
                     die.
87                if (tag == SHUTDOWN_TAG)
88                  {
89                    if (isSendPending)
90                      {
91                        MPI_Wait(&request, &snd_status);
92                        isSendPending = 0;
93                      }
94                    break;
95                  }
96                else
97                  {
98                    isItemReceived = 1;
```

```
99                }
100
101               snd = frm->worker.f_worker.task(rcv);
102
103               // Workers always communicate to process 1 -> Collector
104               if (isSendPending)
105                 {
106                   // At this point check if the send completes.
107                   MPI_Wait(&request, &snd_status);
108                   isSendPending = 0;
109                 }
110             }
111
112           // Now send shutdown TAG.
113           MPI_Send(snd, frm->out_byte, MPI_BYTE, 1, SHUTDOWN_TAG, frm->
                  fcomm);
114         }
115
116       // COLLECTOR.
117       else if (myFRank == 1 && frm->identifier_Flag == 2)
118         {
119           // If the Farm is ordering call item ordering Collector.
120           if (frm->frm_type == FARM_Ord)
121             {
122               ordering_Collector(frm->out_rank, frm->in_byte, frm->
                      out_byte,
123                   frm->parl_degree, frm->fcomm, frm->parentComm);
124             }
125           else
126             {
127               void* rcv;
128               rcv = malloc(frm->in_byte);
129               int shutDownCount = 0;
130               if (isPrint)
131                 printf("Farm Collector ... starts  \n");
132
133               // Receive once to enable Non-blocking Send.
134               MPI_Recv(rcv, frm->in_byte, MPI_BYTE, MPI_ANY_SOURCE,
                      MPI_ANY_TAG,
135                   frm->fcomm, &status);
136               int tag = status.MPI_TAG;
137
138               int isSendPending = 0;
```

```
139                 while (shutDownCount < frm->parl_degree)
140                   {
141
142                     if (tag != SHUTDOWN_TAG)
143                       {
144                         MPI_Isend(rcv, frm->out_byte, MPI_BYTE, frm->
                                out_rank,
145                                  tag, frm->parentComm, &request);
146                         isSendPending = 1;
147                       }
148                     else
149                       {
150                         shutDownCount++; //listen to number-of-workers
                                times shutdown tag.
151                       }
152
153                     // Do some usefull work, to overlap it with
                            communication.
154                     if (shutDownCount < frm->parl_degree)
155                       {
156                         MPI_Recv(rcv, frm->in_byte, MPI_BYTE,
                                MPI_ANY_SOURCE,
157                                  MPI_ANY_TAG, frm->fcomm, &status);
158                         tag = status.MPI_TAG;
159                       }
160
161                     // At this point check if the send completes.
162                     if (isSendPending)
163                       {
164                         MPI_Wait(&request, &snd_status);
165                         isSendPending = 0;
166                       }
167                   }
168
169                 // Now send shutdown TAG.
170                 MPI_Send(rcv, frm->out_byte, MPI_BYTE, frm->out_rank,
171                     SHUTDOWN_TAG, frm->parentComm);
172               }
173             if (isPrint)
174               printf("Farm Collector  ... ends\n");
175           }
176       }
177 }
```

Listing 6.10: Farm skeleton starter

```
1
2  // Farm Skeleton Start.
3
4  int MPI_Farm_start(farm_t *frm)
5  {
6    // The the Farm compute skeleton
7    if (myWRank() >= frm->start_rank && myWRank() < frm->end_rank
8        && myWRank() != 0 && myWRank() != getWSize() - 1)
9      {
10       MPI_Farm_compute(frm);
11     }
12 }
```

Listing 6.11: Farm skeleton starter for pipeline workers

```
1
2  // Farm Skeleton Start.
3
4  int MPI_Farm_start_fp(farm_fp_t *frm, pstage_t *stages)
5  {
6
7    if (myWRank() > frm->start_rank + 1 && myWRank() < frm->end_rank)
8      {
9
10       //call the Pipe stage for all stages
11       int wrks_stage_info = 1;
12       int wrkr_info = 1;
13       int i;
14       for (i = 0; i < frm->parl_degree * frm->worker.p_worker.ns; i++)
15         {
16           if (isPrint && (myWRank() == frm->start_rank + 2 + i))
17             printf("Stage %d in Worker %d Farm workers ... starts \n",
18                 wrks_stage_info, wrkr_info);
19           if (myWRank() == frm->start_rank + 2 + i)
20             {
21               MPI_Pipe_seq_fp(stages[i].in_rank, stages[i].out_rank,
22                   stages[i].in_count, stages[i].out_count, stages[i].
                        f_ptr,
23                   frm->parentComm);
```

```
24              }
25            wrks_stage_info++;
26            if (wrks_stage_info > frm->worker.p_worker.ns)
27              {
28                wrks_stage_info = 1;
29                wrkr_info++;
30              }
31            if (isPrint && (myWRank() == frm->start_rank + 2 + i))
32              printf("Stage %d in Worker %d Farm workers ... finishes \n",
33                  wrks_stage_info, wrkr_info);
34          }
35      }
36
37    //followed by call to Farm Emitter and collector
38    if (myWRank() == frm->emt_rank || myWRank() == frm->coll_rank)
39      {
40        MPI_Farm_compute_fp(frm);
41      }
42  }
```

Listing 6.12: Constants header file

```
1
2   // Header file containing constant values.
3   //-----------------------------------------------------------------
4   // A value used to mean the skeleton need to shutdown.
5   #define          SHUTDOWN_TAG             0
6   #define          TAG_START_NUM            2
7   #define          TIME_STAMP_INFO          1
8
9   // Boolean Values
10  #define          true                     1
11  #define          false                    0
12
13  // Flag to print more info,
14  // set it to 1 if you want to see detail information
15  // when the library is running
16  #define          isPrint                  0
17  #define          ordering_info            0
18
19  //Number of items that a library can order, Chage it to order more or
        less items.
```

```
20 #define          MAX_OUT_OF_ORDER         10000
21
22 //-----------------------------------------------------------------
```

Listing 6.13: pipeline header

```
1
2  typedef enum
3  {
4    SEQ_STAGE, FRM_STAGE
5  } stage_type;
6
7  //A Stage can be simple pipe or a Farm
8  struct farm_t;
9  typedef union
10 {
11   pstage_t pipe;
12   farm_t* farm;
13 } stage;
14
15 typedef struct
16 {
17   stage stg;
18   stage_type stg_type;
19 } p_stage;
20
21 //**STRUCTURE OF THE OVERALL PIPELINE
22 typedef struct
23 {
24 //number of pipeline stages
25   int ns;
26
27 //array of pipeline stages
28   p_stage *stage;
29
30 //total number of stream items is known to a streamer.
31 //int items_num;
32
33 //pointer to external input buffer
34   void* input_buff;
35
36 //size of input item in byte
```

```
37    int in_byte;
38
39  //size of output item in byte
40    int out_byte;
41
42  //pointer to the final output buffer
43    void* output_buff;
44
45  //the Parent Comm, it can be MPI_COMM_WORLD
46    MPI_Comm parentComm;
47
48    int parentComm_sz;
49
50  // rank of the first and the last pipeline stages
51    int ini_stId, final_stId;
52
53  //a Pipeline can be ordering or non-odering
54    Pipe_type isOdering;
55
56  //a color that has to specified by each process,
57  //it will be used to create communicator with the right number of
        processes.
58    int color;
59
60  // a key is used to order process rank in new communicator,
61  // specially very help full for skeletons such as Farm, in nesting
62    int key;
63
64  // If we use MPI_Comm_split, all stages will have the same Comm handler,
65  // infact they are different handles.,
66  // process can identify where it belongs using color
67    MPI_Comm st_comm;
68
69  //Number of processes used so far, helps to group process to stages
70    int used_processes;
71  } pipe_t;
72
73  //
        ------------------------------------------------------------------------
```

Listing 6.14: Farm header

```
1
2  //                        FARM structure                                  //
3
4  //FARM structure
5  typedef struct
6  {
7  // Number of worker.
8    int parl_degree;
9
10 // A function pointer that a worker need to execute
11   generic_fp_worker worker;
12
13 // Number of input and output item
14   int in_byte, out_byte;
15
16 // A farm can be ordering or non-ordering
17   Farm_type frm_type;
18
19 //rank of a worker that has to assinged a work, in a round robin way
20   int next_worker;
21
22 //key and color are used to create communicator
23   int key;
24   int color;
25
26   int identifier_Flag;
27
28 //Number of processes used so far for the overl all skeleton.
29   int used_processes;
30
31 //ranks that a Farm skeleton communcate with (i.e external to Farm)
32   int in_rank, out_rank;
33
34 //Parent Communicator
35   MPI_Comm parentComm;
36
37 //communcator used for Farm
38   MPI_Comm fcomm;
39
40   MPI_Comm inner_comm;
41
42 //to check if Farm is called from lib or from user directly
43   int isLib;
```

```
44
45 //range of rank of processes used for Farm
46   int start_rank, end_rank;
47 //input streams number
48   int item_num;
49
50 //input and output buffer
51   void* input_buff;
52   void* output_buff;
53
54 //Emitter and collector rank at world level
55   int emt_rank;
56   int coll_rank;
57
58 //to check if the stage before this farm is also a farm
59   int isPreceedingFarm;
60
61   int preceedingParDeg;
62
63 } farm_t;
64
65 //-------------------------------------------------------------------
```

Listing 6.15: Item Ordering Collector

```
1
2 // Item Ordering stage
3 int ordering_Collector(int destination, int inbyteSize, int outbyteSize,
      int nw,
4     MPI_Comm comm, MPI_Comm parentComm)
5 {
6
7   if (isPrint)
8     printf("Item ordering at Farm-Collector is activated ..... \n\n");
9
10  MPI_Status stat, snd_status;
11  MPI_Request request;
12
13  MPI_Request *requestBuff = (MPI_Request *) malloc(sizeof(MPI_Request))
      ;
14  MPI_Status *snd_statusBuff = (MPI_Status *) malloc(sizeof(MPI_Status))
      ;
```

```
15
16  // prepare the correct MPI tag threashold.
17  int tag_max_size, flag;
18  int countRequest = 0;
19  void* tagmax;
20  MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &tagmax, &flag);
21
22  // If can not read flag, use the default one specified in MPI
        specification:  32767
23  if (!flag)
24    {
25      tag_max_size = 32767;
26    }
27  else
28    {
29      tag_max_size = *(int *) tagmax;
30    }
31
32  // Prepare ordering Buffer.
33  void* orderBuffer = malloc(outbyteSize * MAX_OUT_OF_ORDER);
34
35  // Prepare a space where to receive data, which will or will not be
        buffered.
36  // this needs to circulate around once it is incremeted
        MAX_OUT_OF_ORDER times. d = d - (sizeof(int)*MAX_OUT_OF_ORDER)
37  void* d = malloc(inbyteSize * MAX_OUT_OF_ORDER);
38  // Pointer to iterate over the buffer.
39  void* data[MAX_OUT_OF_ORDER];
40
41  // Item tag buffer.
42  int Tag_Buffer[MAX_OUT_OF_ORDER] =
43    { 0 };
44  int tag, i, j;
45
46  // to keep track of how may times d is incremeted, please also see
        comment on void* d above.
47  j = 0;
48
49  // keep track of the expected tag.
50  int local_check = TAG_START_NUM;
51
52  //AT least receive once to check the tag
53  MPI_Recv(d, inbyteSize, MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &
```

```
    stat);
54  tag = stat.MPI_TAG;
55
56  // Call the function, and store result in orderBuffer.
57  orderBuffer = d;
58  int isOderingActive = 1;
59  int isSendPending = 0;
60
61  while (tag != SHUTDOWN_TAG)
62    {
63      // If MPI_TAG_UB is reached return back to TAG_START_NUM
64      if (local_check >= tag_max_size)
65        {
66          local_check = TAG_START_NUM;
67        }
68      if (isOderingActive)
69        {
70          if (tag == local_check)
71            {
72              if (ordering_info)
73                printf("IN ORDER: with tag %d   \n\n", tag);
74              MPI_Isend(orderBuffer, outbyteSize, MPI_BYTE, destination,
                    tag,
75                    parentComm, &request);
76              isSendPending = 1;
77              local_check++;
78            }
79          else
80            {
81              // Check if there are items in in buffer that can fill the
                    missing item gap.
82              for (i = 0; i < MAX_OUT_OF_ORDER; i++)
83                {
84                  if (Tag_Buffer[i] != 0 && Tag_Buffer[i] == local_check
                        )
85                    {
86                      MPI_Isend(data[i], outbyteSize, MPI_BYTE,
                            destination,
87                          local_check, parentComm, &requestBuff[0]);
88                      //stream out the item in the buffer
89                      if (ordering_info)
90                        printf(" \n FROM Buffer.....  with TAG =  %d  \
                            n\n",
```

```
91                              local_check);
92                      local_check++;
93
94                      // If MPI_TAG_UB is reached return back to
                           TAG_START_NUM
95                      if (local_check >= tag_max_size)
96                        {
97                          local_check = TAG_START_NUM;
98                        }
99                      Tag_Buffer[i] = 0;
100                     i = 0;
101                     MPI_Wait(&requestBuff[0], &snd_statusBuff[0]);
102                   }
103               }
104
105           //The above checking is done prior to this to help free
                some buffer slot (in case!)
106           // If a matching item tag did not arrive,  we have to
                buffer it.
107           int checkSlote = 0;
108           for (i = 0; i < MAX_OUT_OF_ORDER; i++)
109             {
110               if (Tag_Buffer[i] == 0)
111                 {
112                   //buffer it
113                   data[i] = orderBuffer;
114                   Tag_Buffer[i] = tag;
115                   checkSlote = 1;
116                   break;
117                 }
118             }
119
120           if (!checkSlote)
121             {
122               //Print an  info to a user that more items are getting
                      out of order and the library is not
123               printf(
124                   "\n\nWARNNING:More items are getting out of order,
                        the library isn't ordering them any more\n");
125               printf(
126                   "        Item ordering is dectivated , items will
                       be streamed as they arrive ...\n\n");
127               isOderingActive = 0;
```

```
128                 }
129               }
130         } // Odering block end
131
132       // if ordering is deactivated becuase of many items that are
              getting out of order.
133       // send what is received directly and also don't forget to flush
              what is in the buffer.
134       if (!isOderingActive)
135         {
136           MPI_Send(orderBuffer, outbyteSize, MPI_BYTE, destination, tag,
137               parentComm);
138         }
139
140       d = d + inbyteSize;
141
142       //check if d is reached its maximum address range and turn it to
              initial adress space
143       ++j;
144       if (j >= MAX_OUT_OF_ORDER)
145         {
146           d = d - (inbyteSize * MAX_OUT_OF_ORDER);
147           orderBuffer = orderBuffer - (outbyteSize * MAX_OUT_OF_ORDER);
148           j = 0;
149         }
150
151       MPI_Recv(d, inbyteSize, MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG,
              comm,
152           &stat);
153       tag = stat.MPI_TAG;
154       orderBuffer = orderBuffer + outbyteSize;
155       // Call the function, and store result in orderBuffer.
156       orderBuffer = d;
157
158       //check if there is pending Send, if so complete it.
159       if (isSendPending)
160         {
161           MPI_Wait(&request, &snd_status);
162           isSendPending = 0;
163         }
164     }
165
166   // Flush out if anything is in Buffer. if ordering process is still
```

```
     active take care of the ordering
167  // otherwise just flush what you have in buffer.]
168
169  // if active.
170  if (isOderingActive)
171    {
172      for (i = 0; i < MAX_OUT_OF_ORDER; i++)
173        {
174          if (Tag_Buffer[i] != 0 && Tag_Buffer[i] == local_check)
175            {
176              MPI_Isend(data[i], outbyteSize, MPI_BYTE, destination,
177                  local_check, parentComm, &requestBuff[0]);
178              if (ordering_info)
179                printf("Streamed in order from Buffer .... with TAG = %d
                       \n\n",
180                    local_check);
181              local_check++;
182
183              // If MPI_TAG_UB is reached return back to TAG_START_NUM
184              if (local_check >= tag_max_size)
185                {
186                  local_check = TAG_START_NUM;
187                }
188              Tag_Buffer[i] = 0;
189              i = 0;
190              MPI_Wait(&requestBuff[0], &snd_statusBuff[0]);
191            }
192        }
193    }
194
195  // clean the buffer any way.
196  for (i = 0; i < MAX_OUT_OF_ORDER; i++)
197    {
198      if (Tag_Buffer[i] != 0)
199        {
200          if (ordering_info)
201            printf("Streamed out of order from Buffer .... with TAG = %d
                     \n\n",
202                 Tag_Buffer[i]);
203          MPI_Send(data[i], outbyteSize, MPI_BYTE, destination,
                 Tag_Buffer[i],
204                 parentComm);
205          Tag_Buffer[i] = 0;
```

```
206             i = 0;
207           }
208       }
209
210     // clarify if there are pending send operations
211     MPI_Waitall(1, requestBuff, MPI_STATUSES_IGNORE);    //&index,
          snd_statusBuff);
212
213     // A shutdown Tag.
214     MPI_Send(orderBuffer, outbyteSize, MPI_BYTE, destination, SHUTDOWN_TAG
          ,
215          parentComm);
216
217     return 0;
218 }
```

Listing 6.16: Item Ordering Stage

```
1
2  // Item Ordering stage
3  int item_Ordering_Stage(int source, int destination, int inbyteSize,
4      int outbyteSize, void*
5      (*f_ptr)(void*), MPI_Comm comm)
6  {
7
8    if (isPrint)
9      printf("Item ordering at pipeline stage is activated ..... \n\n");
10
11   MPI_Status stat, snd_status;
12   MPI_Request request, requestBuff;
13
14   // prepare the correct MPI tag threashold.
15   int tag_max_size, flag;
16   void* tagmax;
17   MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_TAG_UB, &tagmax, &flag);
18
19   // If can not read flag, use the default one specified in MPI
          specification:  32767
20   if (!flag)
21     {
22       tag_max_size = 32767;
23     }
```

```
24    else
25      {
26        tag_max_size = *(int *) tagmax;
27      }
28
29    // Prepare ordering Buffer.
30    void* orderBuffer = malloc(outbyteSize * MAX_OUT_OF_ORDER);
31
32    // Prepare a space where to receive data, which will or will not be
          buffered.
33    // this needs to circulate around once it is incremeted
          MAX_OUT_OF_ORDER times. d = d - (sizeof(int)*MAX_OUT_OF_ORDER)
34    void* d = malloc(inbyteSize * MAX_OUT_OF_ORDER);
35    // Pointer to iterate over the buffer.
36    void* data[MAX_OUT_OF_ORDER];
37
38    // Item tag buffer.
39    int Tag_Buffer[MAX_OUT_OF_ORDER] =
40      { 0 };
41    int tag, i, j;
42
43    // to keep track of how may times d is incremeted, please also see
          comment on void* d above.
44    j = 0;
45
46    // keep track of the expected tag.
47    int local_check = TAG_START_NUM;
48
49    //AT least receive once to check the tag
50    MPI_Recv(d, inbyteSize, MPI_BYTE, source, MPI_ANY_TAG, comm, &stat);
51    tag = stat.MPI_TAG;
52
53    // Call the function, and store result in orderBuffer.
54    orderBuffer = f_ptr(d);
55    int isOderingActive = 1;
56    int isSendPending = 0;
57
58    while (tag != SHUTDOWN_TAG)
59      {
60        // If MPI_TAG_UB is reached return back to TAG_START_NUM
61        if (local_check >= tag_max_size)
62          {
63            local_check = TAG_START_NUM;
```

```
64          }
65        if (isOderingActive)
66          {
67            if (tag == local_check)
68              {
69                if (ordering_info)
70                  printf("IN ORDER: with tag %d   \n\n", tag);
71                MPI_Isend(orderBuffer, outbyteSize, MPI_BYTE, destination,
                      tag,
72                      comm, &request);
73                isSendPending = 1;
74                local_check++;
75              }
76            else
77              {
78                // Check if there are items in in buffer that can fill the
                      missing item gap.
79                for (i = 0; i < MAX_OUT_OF_ORDER; i++)
80                  {
81                    if (Tag_Buffer[i] != 0 && Tag_Buffer[i] == local_check
                        )
82                      {
83                        MPI_Isend(data[i], outbyteSize, MPI_BYTE,
                            destination,
84                            local_check, comm, &request);
85                        //stream out the item in the buffer
86                        if (ordering_info)
87                          printf(" \n FROM Buffer......  with TAG =  %d  \
                              n\n",
88                              local_check);
89                        local_check++;
90
91                        // If MPI_TAG_UB is reached return back to
                            TAG_START_NUM
92                        if (local_check >= tag_max_size)
93                          {
94                            local_check = TAG_START_NUM;
95                          }
96                        Tag_Buffer[i] = 0;
97                        i = 0;
98                        MPI_Wait(&request, &snd_status);
99                      }
100                 }
```

```
101
102             //The above checking is done prior to this to help free
                    some buffer slot (in case!)
103             // If a matching item tag did not arrive,  we have to
                    buffer it.
104             int checkSlote = 0;
105             for (i = 0; i < MAX_OUT_OF_ORDER; i++)
106               {
107                 if (Tag_Buffer[i] == 0)
108                   {
109                     //buffer it
110                     data[i] = orderBuffer;
111                     Tag_Buffer[i] = tag;
112                     checkSlote = 1;
113                     break;
114                   }
115               }
116
117             if (!checkSlote)
118               {
119                 //Print an  info to a user that more items are getting
                        out of order and the library is not
120                 printf(
121                     "\n\nWARNNING:More items are getting out of order,
                            the library isn't ordering them any more\n");
122                 printf(
123                     "        Item ordering is dectivated , items will
                           be streamed as they arrive ...\n\n");
124                 isOderingActive = 0;
125               }
126           }
127       } // Odering block end
128
129     // if ordering is deactivated becuase of many items that are
            getting out of order.
130     // send what is received directly and also don't forget to flush
            what is in the buffer.
131     if (!isOderingActive)
132       {
133         MPI_Send(orderBuffer, outbyteSize, MPI_BYTE, destination, tag,
                comm);
134       }
135
```

```
136         d = d + inbyteSize;
137
138         //check if d is reached its maximum address range and turn it to
                initial adress space
139         ++j;
140         if (j >= MAX_OUT_OF_ORDER)
141           {
142             d = d - (inbyteSize * MAX_OUT_OF_ORDER);
143             orderBuffer = orderBuffer - (outbyteSize * MAX_OUT_OF_ORDER);
144             j = 0;
145           }
146
147         MPI_Recv(d, inbyteSize, MPI_BYTE, source, MPI_ANY_TAG, comm, &stat
                );
148         tag = stat.MPI_TAG;
149         orderBuffer = orderBuffer + outbyteSize;
150         // Call the function, and store result in orderBuffer.
151         orderBuffer = f_ptr(d);
152
153         //check if there is pending Send, if so complete it.
154         if (isSendPending)
155           {
156             MPI_Wait(&request, &snd_status);
157             isSendPending = 0;
158           }
159       }
160
161     // Flush out if anything is in Buffer. if ordering process is still
            active take care of the ordering
162     // otherwise just flush what you have in buffer.]
163
164     // if active.
165     if (isOderingActive)
166       {
167         for (i = 0; i < MAX_OUT_OF_ORDER; i++)
168           {
169             if (Tag_Buffer[i] != 0 && Tag_Buffer[i] == local_check)
170               {
171                 MPI_Isend(data[i], outbyteSize, MPI_BYTE, destination,
172                     local_check, comm, &requestBuff);
173                 if (ordering_info)
174                   printf("Streamed in order from Buffer .... with TAG = %d
                        \n\n",
```

```
175                     local_check);
176               local_check++;
177
178               // If MPI_TAG_UB is reached return back to TAG_START_NUM
179               if (local_check >= tag_max_size)
180                 {
181                   local_check = TAG_START_NUM;
182                 }
183               Tag_Buffer[i] = 0;
184               i = 0;
185               MPI_Wait(&requestBuff, &snd_status);
186             }
187         }
188     }
189
190   // clean the buffer any way.
191   for (i = 0; i < MAX_OUT_OF_ORDER; i++)
192     {
193       if (Tag_Buffer[i] != 0)
194         {
195           if (ordering_info)
196             printf("Streamed out of order from Buffer .... with TAG = %d
                    \n\n",
197                 Tag_Buffer[i]);
198           MPI_Send(data[i], outbyteSize, MPI_BYTE, destination,
                  Tag_Buffer[i],
199                  comm);
200           Tag_Buffer[i] = 0;
201           i = 0;
202         }
203     }
204
205   // A shutdown Tag.
206   MPI_Send(orderBuffer, outbyteSize, MPI_BYTE, destination, SHUTDOWN_TAG
       , comm);
207
208   return 0;
209 }
```

Listing 6.17: other utility functions

```
1
```

```
2  // Returns rank at WORLD COMM level.
3  int myWRank()
4  {
5    int rank;
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    return rank;
8  }
9
10 // Returns size at WORLD_Comm.
11 int getWSize()
12 {
13   int size;
14   MPI_Comm_size(MPI_COMM_WORLD, &size);
15   return size;
16 }
17
18 // Returns rank for requested communicator.
19 int myRank(MPI_Comm comm)
20 {
21   int rank;
22   MPI_Comm_rank(comm, &rank);
23   return rank;
24 }
25
26 int getSize(MPI_Comm comm)
27 {
28   int size;
29   MPI_Comm_size(comm, &size);
30   return size;
31 }
```