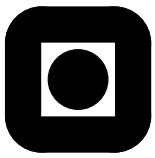
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK



MASTEROPPGAVE

Kandidatens navn: Einar Magnus Råberg Rosenvinge

Fag: Datateknikk

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk): Online Task Scheduling on Heterogeneous Clusters: An Experimental Study

Oppgavens tekst:

In today's international HPC arena, there is a clear trend from traditional supercomputers towards cluster computing solutions. This gives rise to new challenges, represented by slow communication between nodes (often provided by Ethernet or the faster Myrinet), heterogeneity (a cluster's nodes might have different performance characteristics), and dynamism (a cluster's nodes might have widely varying background processing loads).

The focus of this Master's thesis is to investigate the suitability of a special class of applications for clusters. Such applications are composed of many independent work units which can be computed independently of each other and in any order. Example applications include large data set processing, signal and image processing, scientific and numerical computing, query processing in database systems and Monte Carlo simulations.

In order to run these applications efficiently on a cluster, an adequate scheduling strategy is needed. Many strategies are suggested in the literature, and the student must identify these and implement them with a test case application. The implementation should include appropriate techniques to overcome the challenges a cluster poses.

Also, the student should aim to create a scheduling strategy of his own, and compare this experimentally to the previous scheduling strategies found in the literature.

The implementation created in this work must run on IDI's cluster, ClustIS, and should be written in C, using the MPI library. It must be portable, so that it can run on other clusters at a later date.

Oppgaven gitt:	20. januar 2004
Besvarelsen leveres innen:	15. juni 2004
Besvarelsen levert:	15. juni 2004
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veiledere:	Anne Cathrine Elster og Cyril Banino

Trondheim, 15. juni 2004

Anne Cathrine Elster
Faglærer

Master's Thesis: Online Task Scheduling on Heterogeneous Clusters: An Experimental Study

Einar Magnus Råberg Rosenvinge
einarmr@tihlde.org

Norwegian University of Science and Technology (NTNU)
Faculty of Information Technology,
Mathematics and Electrical Engineering (IME)
Department of Computer and Information Science (IDI)

Supervisors:
Anne Cathrine Elster, PhD
Cyril Banino

15th June 2004

Abstract

We study the problem of scheduling applications composed of a large number of tasks on heterogeneous clusters. Tasks are identical, independent from each other, and can hence be computed in any order. The goal is to execute all the tasks as quickly as possible. We use the Master-Worker paradigm, where tasks are maintained by the master which will hand out batches of a variable amount of tasks to requesting workers. We introduce a new scheduling strategy, the Monitor strategy, and compare it to other strategies suggested in the literature. An image filtering application, known as matched filtering, has been used to compare the different strategies. Our implementation involves data-staging techniques in order to circumvent the possible bottleneck incurred by the master, and multi-threading to prevent possible processor idleness.

Preface

This thesis documents the work done on my Master of Computer Science degree from the Norwegian University of Science and Technology (NTNU), Faculty of Information Technology, Mathematics and Electrical Engineering (IME), Department of Computer and Information Science (IDI).

A paper derived from this thesis has been accepted for the PARA '04 conference, [1], and will be presented in Copenhagen, Denmark on June 23, 2004.

I would like to thank my supervisors, Anne Cathrine Elster and Cyril Banino, who have made many valuable contributions and suggestions along the way.

I would also like to thank Ole Christian Eidheim for his Matlab implementation of the matched filtering algorithm, Robin Holtet for some master/worker code which has been used as a starting point, Cyril Banino for some of the text about ClustIS in Section 2.1, and Nils O. Selåsdal for the implementation of threadqueue.c.

Einar Magnus Råberg Rosenvinge

Contents

List of Figures

Chapter 1

Introduction

In today's international HPC arena, new applications are emerging that demand more and more computing power. This computing power is provided by new computing platforms, at a decreasing cost. There is a clear trend from traditional supercomputers towards cluster computing solutions, which can be a large number of single-processor computers, but also a set of smaller multiprocessor shared-memory computers. This trend is mainly motivated by the fact that a cluster typically can be constructed at a cost that is modest compared to the cost for a traditional supercomputer that has equivalent computing power characteristics. The operation, use and performance characteristics of such cluster configurations are however significantly different from those of supercomputers. One prominent example is the slow communication between nodes, often provided by a high-latency, low-bandwidth interface such as Ethernet or the faster Myrinet. Another drawback of clusters is the significant human effort needed to maintain them. If each node has a mean time between failure (MTBF) of x seconds; with p of nodes, this means that a node will fail every x/p seconds. This means that long-term applications must be able to handle failures on some of the nodes.

A cluster gives rise to some challenges not found on traditional supercomputers. A cluster is a *heterogeneous* environment, meaning that its nodes may have different performance characteristics. Also, a cluster's nodes might not be dedicated, meaning that they may have a non-negligible background processing load. This is called a *dynamic* environment. These challenges imply that one needs an adequate scheduling strategy to get good performance on a cluster.

Our goal is to schedule a certain class of applications in the most effective way on a cluster.

Our work is relevant for many institutions; in Norway most notably for The Norwegian High Performance Computing consortium, NOTUR, [22]. NOTUR is planning to add a large-scale production cluster to the Norwegian HPC infrastructure in 2005, and is currently investigating what the consequences will be for applications and users, [27].

1.1 Bag-of-Tasks Applications

The applications we are studying are often called *bag-of-tasks* applications (BoT), and are divisible into a large number of *work units*, or *tasks*, down to a certain granularity. There is no inter-task communication, and tasks can be computed in any order. Finally, tasks are atomic, i.e. their computation cannot be preempted.

Many applications can be parallelized in such a way. Examples of such applications include matrix multiplication, Gaussian elimination, image processing applications such as ray-tracing, [17], and Monte Carlo simulations, [3].

1.2 Scheduling

In the field of parallel scheduling, it is a well known fact that all processors must finish their computations simultaneously for the total execution time to be minimized. This calls for an adequate scheduling strategy, which gives each node a proper amount of work. The scheduling strategy must be able to handle processor heterogeneity, dynamic environments, and processor failures.

As an example, the simplest scheduling strategy, called Static Chunking (SC), consists of dividing the problem domain into p batches of tasks when we have p processors. In a homogeneous environment, where each processor has the same computing speed, the batches have the same size. Similarly, in a heterogeneous environment, the batch sizes are proportional to the processor computing speeds. However, Hagerup, [14], argued that task/batch execution times vary for two main reasons:

- Algorithmic variance: The algorithm itself runs faster on one data set than on another. This is typical for many algorithms, e.g. image processing—very few spend the same time independently of input data.
- System-induced variance: Local cache misses, varying memory latencies, clock interrupts, other processes in the system, and operating-system interference (dynamism).

The SC strategy will hence lead to some nodes being idle at the end of the computation, waiting for the other nodes to finish.

Therefore, more fine-grained scheduling strategies appeared in the literature, which distribute the problem in batches of decreasing sizes in several rounds. The aim is to keep some work for processors that will finish first.

Another important issue is fault-tolerance. As earlier mentioned, a cluster has a relatively high failure rate compared to a traditional supercomputer. Without centralized control, if one node fails during the computation, there is no way to recover.

The need for adequate scheduling and fault-tolerance is the main motivation behind the master/worker paradigm, described in the following section.

1.2.1 The Master/Worker Paradigm

In the master/worker paradigm, a master process allocates the tasks for computation to worker processes. The workers compute the given tasks and return the results of each task back to the master, after which the master reassembles

all results. The computation is finished when all tasks have been processed. See figure 1.1 for an illustration of the classic master/worker paradigm.

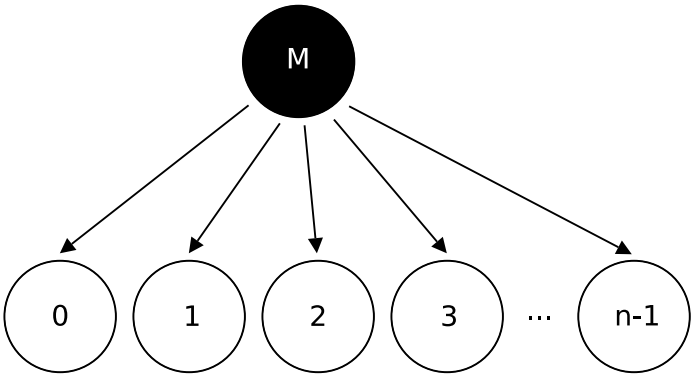


Figure 1.1: The master/worker paradigm

The master's primary goal is to employ various scheduling algorithms to ensure that workers get just enough tasks, and consequently, finish computing at the same time.

Fault-tolerance is also achieved through the master/worker paradigm. If one worker fails, the master will not receive a result from this worker, and can reassign the associated task to another worker after a user-specified timeout. Should the master fail, one can implement a collective protocol that makes one of the workers become master.

However, the master/worker approach does have its drawbacks. If the computation to communication ratio is low, the master node will have problems feeding workers with new tasks without the workers having to wait. Congestion at the master node is a well-known problem of the master/worker paradigm. One solution to this problem is the use of data staging, [10], see Section 1.4.3.

1.3 Notation

Throughout this thesis, the following notation is used.

N denotes the total number of tasks

p denotes the number of processors

R_t denotes the number of remaining unassigned tasks held by the master at time step t

W is the set of all processors, and a processor is referred to as $w_i, i \in [0, p\rangle$

1.4 Related Work

1.4.1 Task Scheduling

Here, we will briefly present relevant scheduling strategies suggested in the literature, [14].

The *Static Chunking (SC)* strategy, [17], assigns one batch of N/p tasks to each worker. This means that in a heterogeneous and/or dynamic environment, worker nodes will not finish their batches at the same time, and processing power will be "lost" due to some nodes being idle at the end of the computation. In a homogeneous and static environment, however, this strategy performs quite well, and is extremely easy to implement.

At the other end of the spectrum is the *Self Scheduling (SS)* strategy, [17], where tasks are handed out one by one. This means that worker nodes will spend a lot of time communicating with the master node, and the master might become overloaded. With non-negligible communication latencies and/or an overloaded master, this strategy will perform very badly.

The *Fixed-Size Chunking (FSC)* strategy uses batches of tasks of one fixed size; greater than 1 but less than N/p. It is possible to approximate the optimal batch size, [20].

The *Guided Self Scheduling (GSS)* strategy, [24], gives each worker batches of size R/p. GSS thus uses exponentially decreasing batch sizes. Polychronopoulos and Kuck view this strategy as a trade-off between giving out too many tasks in one batch (more than R/p) and too few tasks in one batch (too large communication overhead).

The *Trapezoid Self-Scheduling (TSS)* strategy, [30], also uses decreasing batch sizes, but the batch size decreases linearly from a first size f to a last size l. At first, TSS computes $Q = [2N/(f+l)]$, $\delta = (f-l)/(Q-1)$ and $k = f$. Each request returns a batch of size k and reduces k by δ. Tzen and Ni advocate the use of $f = N/(2p)$ and $l = 1$.

The *Factoring*, [17], and *Weighted Factoring*, [18], strategies also use decreasing batch sizes. A slightly simplified variant, which does not rely upon the mean and standard deviation of task execution times, gives out half of the remaining tasks R in each round. Factoring divides these tasks evenly among workers; the batch size is then $(R/2)/p$. Weighted Factoring relies upon a precomputed weight ω for each worker, and divides the tasks according to the workers' weights; batch size $(R/2) \times \omega$.

1.4.2 Divisible Load Theory

The *divisible load* model depicts applications that are composed of a large number of homogeneous low-granularity computations called *tasks*. There are no communication dependencies and tasks can therefore be independently processed in parallel. The total application's workload can hence be split into *batches* of arbitrary size (each batch comporting a number of tasks), and this in a linear fashion, i.e. the computation and communication time components of a batch are proportional to its size. The divisible load model has been widely studied since its introduction in 1988, and *divisible load theory* (DLT) has emerged as a new paradigm for scheduling divisible loads on distributed computing platforms [4]. In addition, DLT literature is vast, and recent surveys have been published [5, 26].

DLT provides a practical framework for processing independent tasks onto (possibly heterogeneous) distributed computing resources, and has therefore been utilized for a wide range of applications including image processing (e.g. edge detection [32]), processing of massive experimental data set, signal processing applications [5]. In [9] several other applications have been implemented:

pattern searching, file compression, joining operation in relational databases operations, graph coloring and genetic search.

The basic assumptions of divisible load theory (DLT) are the following. A system composed of p processors $P_0, P_1, \ldots, P_{p-1}$ is considered. The processor P_0 called *originator* or *master* plays a particular role that we will precise hereafter. At the beginning of the computation, the whole workload is stored in the memory of the originator processor P_0. The originator then scatters the workload over the network to the p remote processors. Each processor will then process its part of the workload in parallel. It is widely accepted in the divisible load theory that the return of the results to the master can be neglected. This assumption is made for the sake of simplicity, and may not be realistic for some applications. However, the gathering of the results to the originator can be incorporated in the model for special cases as shown in [6].

1.4.3 Data Staging

Without data staging, the master sends all necessary input data to a worker node when that node requests a task. If one task requires a substantial amount of input data, this means that the master can only serve a very limited number of workers during a period of time.

With data staging, the input data is prepositioned on a storage server, perhaps near the worker nodes, and a pointer to the data is sent to the workers when they request tasks. The workers retrieve the data from the storage when they start working on their tasks.

Elwasif et al., [10], show using both experiments, simulations and theoretical analysis that it is possible to improve task throughput, due to increased master utilization, through the use of data staging.

Chapter 2

Operating Environment

2.1 ClustIS

ClustIS is the computational cluster of the Division for Intelligent Systems (DIS) at the Department of Computer and Information Science (IDI). ClustIS also integrates some nodes from the Division of Complex Computing Systems (KDS).

ClustIS is running Source Mage GNU/Linux, [25], a source-based Linux distribution. The queueing system is OpenPBS, [2], and the MPI implementation is MPICH 1.2.5.2, [21]. As of this writing, the compilers available are the GNU Compiler Collection (gcc) version 3.3.1, [29], and the Intel C Compiler version 8.0, [19]. The C library is glibc 2.2.5, [28].

2.1.1 ClustIS' Profile

ClustIS consists of 1 master node called ClustIS, 1 storage node called Story and 38 computational nodes called node01, node02,..., node38. These nodes are of different types:

- **Master node:** AMD Dual Athlon MP 2100+ (1.66 GHz), 2 GB RAM, 80 + 120 GB IDE HD, 1×Gigabit Ethernet, 1×100Mbit Ethernet
- **Node type 1:** (16 nodes: 1 → 16) AMD Athlon XP 1700+ (1.46 GHz), 2 GB RAM, 1×40 GB IDE HD, 1×100Mbit Ethernet
- **Node type 2:** (12 nodes: node17 → node28) AMD Athlon XP 1700+ (1.46 GHz), 1 GB RAM, 1×40 GB IDE HD, 1×100Mbit Ethernet
- **Node type 3:** (8 nodes: node29 → node36) AMD Athlon MP 1600+ (1.4 GHz), 1 GB RAM, 1×18 GB SCSI HD, 2×100Mbit Ethernet
- **Node type 4:** (2 nodes: node37, node38) AMD Dual Athlon MP 1600+ (1.4 GHz), 1 GB RAM, 3×18 GB SCSI HD, 2×100Mbit Ethernet
- **Storage node:** AMD Athlon XP 1700+ (1.46 GHz), 0.5 GB RAM, 8 + 2×80 GB IDE HD, 1×Gigabit Ethernet, 1×100Mbit Ethernet

The nodes are on a switched private network with 100Mbits/sec between the nodes and the switch and a 1Gbit/sec link between the master node and the switch as depicted in Figure 2.1.

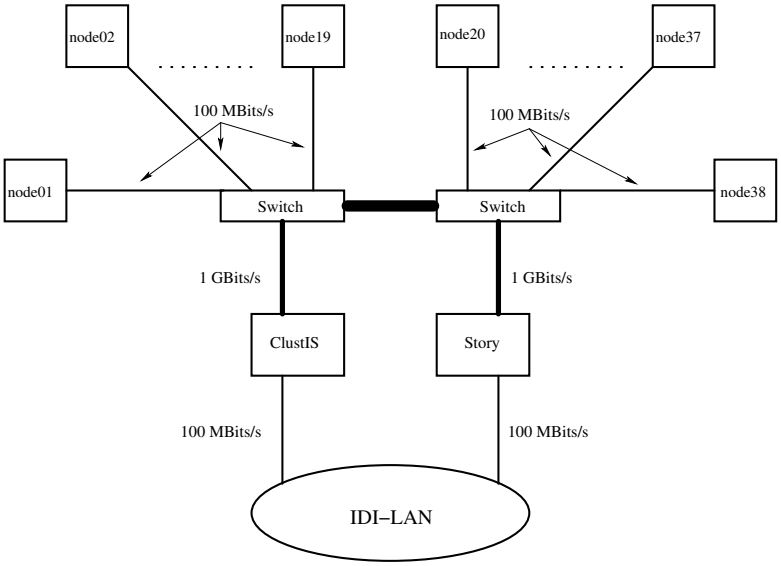


Figure 2.1: Overview of ClustIS' architecture

2.1.2 I/O

On ClustIS, data storage is provided by one node, *Story*. This means that users' home directories are mounted through NFS from Story, and all disk I/O from user applications on processing nodes will go through the slow Ethernet interface.

One solution to avoid this potential bottleneck would consist of scattering input data onto all nodes' local disks before computation starts, and have all nodes write their parts of the total output to their local disks. Still, input data is initially located on Story, and one would have to gather output data to Story. Consequently, the scattering of input data and gathering of output data would add to the total application execution time. In any case, input and output data have to travel through the slow Ethernet interface.

However, we were able to demonstrate I/O parallelism to a certain degree on this cluster. Fig. 2.2 and 2.3 shows concurrent read and write operations on Story. We can see that with up to 8 processes, we have a more or less linear speedup when reading data from the same file. This indicates that having the worker nodes read their part of the data themselves will be faster than having one master scatter and gather data to/from workers.

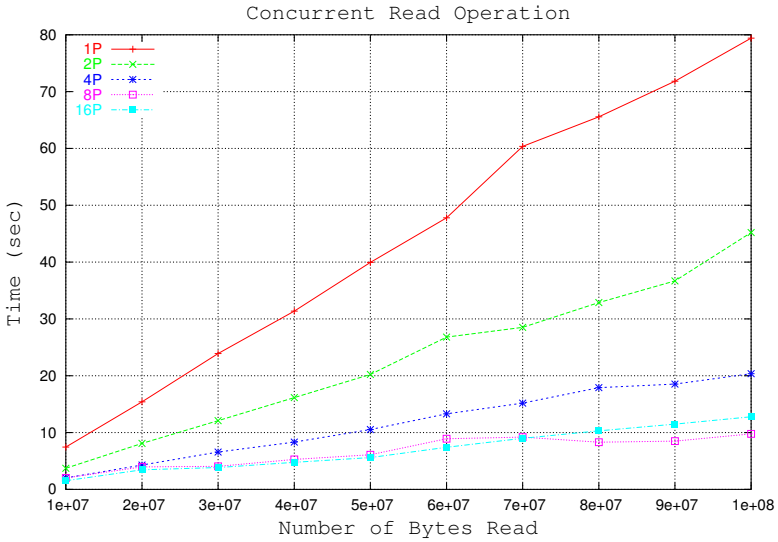


Figure 2.2: 1, 2, 4, 8 and 16 processes reading data concurrently through NFS.

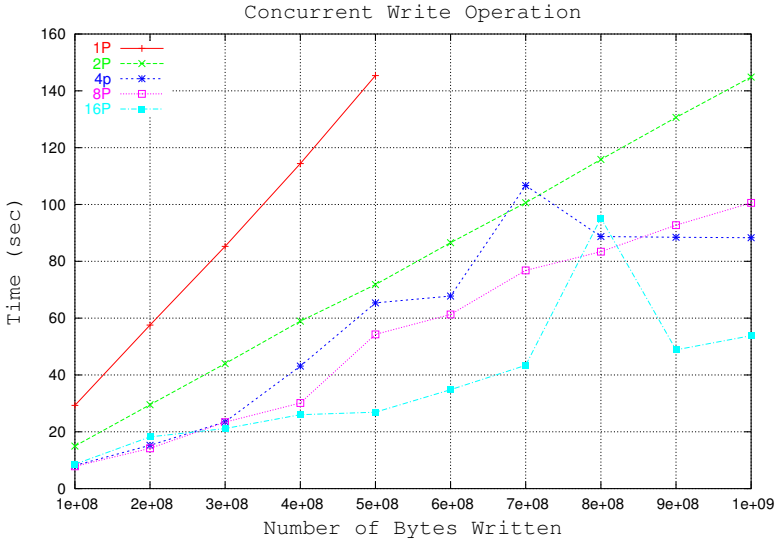


Figure 2.3: 1, 2, 4, 8 and 16 processes writing data concurrently through NFS.

Chapter 3

The Monitor Strategy

3.1 Motivation

On a cluster, each processor might have very different performance characteristics (heterogeneity), as well as a background processing load which may vary a lot (dynamism). To a certain degree, heterogeneity can be dealt with through the job scheduler, by requesting processors with a certain CPU frequency etc. Such functionality, however, is not implemented in many job scheduling systems, and is non-existent in OpenPBS, which is used on our cluster (see Section 2.1). When submitting a job to the job scheduler, one does not know which processors one will get.

Dynamism, however, cannot be dealt with through a job scheduler. The processors' background processing load is unknown before the computation starts, and must be handled by the scheduling strategy in use.

The Monitor strategy is an attempt to create a scheduling strategy which performs well both in a heterogeneous and dynamic environment.

3.2 The Monitor Strategy

The Monitor strategy uses one initial benchmarking phase and several batch computation phases. The notation used in this section is described in Section 1.3.

3.2.1 Benchmarking Phase

During the benchmarking phase, workers request tasks from the master, and the master hands out tasks one by one. Workers measure the time it takes to compute one task, and report these timings to the master when they request another task. When all workers have reported their task computation times, the initialization phase is done.

In most implementations, this means that the benchmarking phase is done when the master has allocated $2p$ tasks. If the workers are implemented with several threads and a task queue (such as described in Section 4.3), the master will allocate more than $2p$ tasks in the benchmarking phase.

Note that workers also report the number of tasks they have been allocated but have not yet computed (tasks waiting in their local task queues) when they request a task.

3.2.2 Batch Computation Phases

In a batch computation phase, the master initially solves the following system of equations

$$\begin{cases} \forall i \in [0, p\rangle, & T_i = (y_i + x_i) \times t_i \\ \forall i \in [1, p\rangle, & T_i = T_{i-1} \\ \forall i \in [0, p\rangle, & R/2 = \sum_i x_i \\ i \in \mathbf{R} \end{cases}$$

Worker w_i has y_i uncomputed tasks in its local task queue, and will be given x_i tasks in this phase. The computation of one task takes t_i time, and it will thus finish its execution of this phase at time $T_i = (y_i + x_i) \times t_i$ (equation 1). For the total execution time to be minimized, all workers must finish their computations simultaneously, hence $T_i = T_{i-1}$ (equation 2). The third equation expresses that we will allocate $R/2$ tasks in this phase.

During a batch computation phase, workers report their task computation times and number of uncomputed tasks every r tasks, so the master is continuously monitoring the workers' task computation times. For the second and following batch computation phases, there is no need for a benchmarking phase, since the master has up-to-date knowledge of the workers' task computation times.

Note that for the following phases, y_i has a lot more significance than it has after the benchmarking phase. As an example, assume that we have two workers, A and B, and assume that worker A slows down to a crawl, while worker B maintains a good computing speed. When worker B is done with a phase, worker A still has lot of a uncomputed tasks. For the following phase, the master solves the linear system again, and the solution yields a negative value for x_A—the master has to subtract tasks from worker A for all the workers to finish at the same time. Subtracting tasks from workers is not part of the strategy, and consequently we must settle for a non-optimal solution. Worker A is given zero tasks, and the tasks that are to be allocated in this phase are distributed among worker B and the other workers.

The task computation time t_i reported by worker w_i will typically be the mean value of its η last computation times. Having $\eta = 1$ might give a non-optimal allocation, since the timing can vary a lot in a dynamic environment. At the other end of the spectrum, a too high value for η conceals changes in processing speeds, which is also non-optimal. The parameter η has to be adjusted for the individual application and/or environment.

3.3 Relation to Other Scheduling Strategies

While the Monitor strategy is quite similar to Weighted Factoring, there is one main difference. The Weighted Factoring strategy assigns tasks to workers in a weighted fashion in each round, where each worker has a predefined weight. This weight has to be computed in advance, before the actual computations

start, which is a disadvantage in a dynamic environment. The Monitor strategy, however, performs such benchmarking *online* throughout the computation, and allows for good performance in a truly dynamic environment.

Previous experiments have shown that the allocation strategy used in Factoring and Weighted Factoring is quite good—in each round, R/p tasks are allocated. We have therefore chosen this approach for the Monitor strategy as well.

Figure 3.2 shows the allocated batch sizes for the scheduling strategies described in Section 1.4.1 as well as the Monitor strategy, when the processors report the task computation times shown in Figure 3.1.

Processor	**Task computation times at time steps**								
	1	2	3	4	5	6	7	8	9
1	0.10	0.15	1.01	0.90	0.28	0.29	0.99	0.90	0.89
2	0.56	0.40	0.50	0.48	0.52	0.53	0.47	0.49	0.50
3	0.89	0.90	0.89	0.24	0.67	0.88	0.60	0.66	0.63
4	0.75	0.76	0.74	0.50	0.45	0.70	0.69	0.63	0.62

Figure 3.1: Examples of task computation times for 4 processors at 9 time steps throughout a computation. Note that for Weighted Factoring, the times at step 1 are used as weights.

Proc.	**Given batches**				
	SC	SS	GSS	TSS	Factoring
1	128	1 1 1 ...	128 40 13 4 1 1	64 48 32 8	64 32 16 8 4 2 1 1
2	128	1 1 1 ...	96 30 9 3 1 1	60 44 28	64 32 16 8 4 2 1 1
3	128	1 1 1 ...	72 23 7 2 1 1	56 40 24	64 32 16 8 4 2 1 1
4	128	1 1 1 ...	54 17 5 2 1	52 36 20	64 32 16 8 4 2 1 1

Proc.	**Given batches**	
	Weighted Factoring	Monitor
1	180 90 45 22 11 6 3 1 1 1	1 1 177 73 11 4 6 4 0 1
2	32 16 8 4 2 1 1 1	1 1 32 27 23 7 3 1 2 2
3	20 10 5 3 1	1 1 20 12 13 14 3 1 1 1
4	24 12 6 3 2 1	1 1 23 14 16 6 4 1 1 1

Figure 3.2: Batch sizes for various scheduling strategies with $N = 512$ tasks and $p = 4$ workers. Note that for the Monitor strategy, we assume $y_i = 0$ at the beginning of every round, meaning that all the processors have computed all their assigned tasks from the previous round.

Chapter 4

Implementation

4.1 Test Case Application

An image filtering application, known as *matched filtering*, [7], is used to compare various scheduling strategies. This application is helpful within medical imaging, in order to detect blood vessels in Computer Tomography (CT) images. Its input is a grayscale image, which is filtered through an image correlation step. The correlation kernel used is a Gaussian hill, which is rotated in all directions and scaled to several sizes. The filter has size $u \times u$ pixels, and consequently, the computation of one pixel in the output image is dependant on the $u \times u$ pixels around it in the input image, see Figure 4.1.

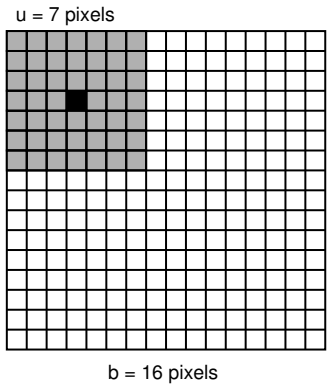


Figure 4.1: One pixel in the output image is dependant on the pixels around it in the input image. In this example, the filter size is 7×7 pixels.

A CT image is a cross-section of a human body, and by detecting blood vessels in multiple CT images, one is able to construct a 3D representation of these blood vessels. Ole Christian Eidheim, who presented us with the matched filtering algorithm, has created a serial application that does this. However, the application is slow, and a parallel version is needed to filter a set of images fast. With more CPU power, one is able to filter higher-resolution CT images, filter more pictures, and thus obtain a higher-quality 3D representation. For more

detailed information about the filtering technique, see [7]. Examples of input and output images are depicted in Fig. 4.3.

An application such as the matched filtering algorithm is, in principle, quite trivial to parallelize. The input image can be divided into tasks corresponding to different parts of the image (lines, columns, blocks), each node can process one or more tasks and hence produce the corresponding parts of the output image.

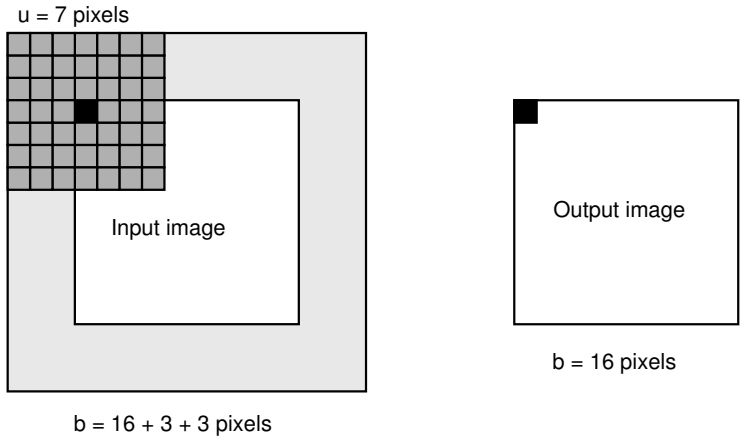
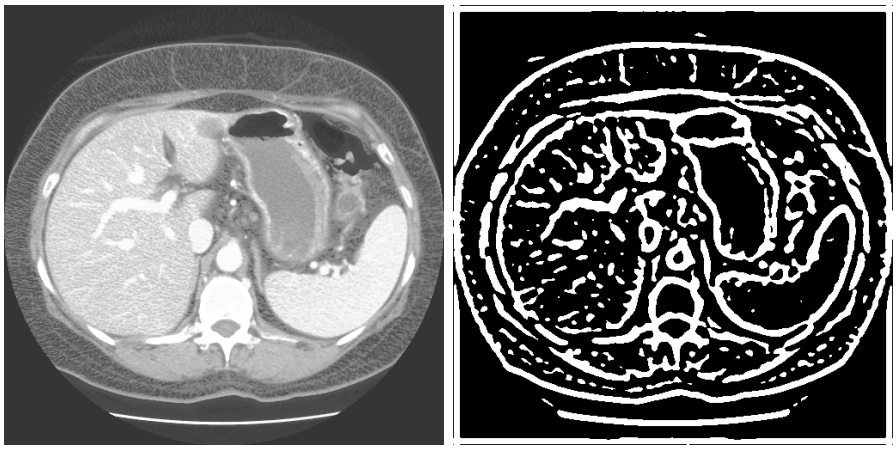


Figure 4.2: To compute an output image block of $b \times b$ pixels, one needs an extra frame of $u/2$ pixels around the corresponding input block.

Since each pixel in the output image is dependant on a $u \times u$ frame around it, this implies that one needs to read a border of $u/2$ pixels around the input image block, see Figure 4.2. Obviously, if the extra data needed is small compared to the block's size, then this will not affect performance much. However, if the amount of data in the border around the input blocks is relatively large compared to the size of the blocks themselves, a significant amount of the input image is actually read multiple times. See Section 6.1.2 for a closer examination of this issue.

To keep our work comparable to other works in the literature of parallel scheduling, we chose to make blocks of a fixed size, including the extra border needed, correspond to tasks in our implementation. Our goal was primarily to design a scheduling strategy for dynamic and heterogeneous environments, and compare it to previous scheduling strategies found in the literature. The matched filtering algorithm was thus only an example application, which has served its purpose well.

When all parts of the image have been filtered, the application needs the global pixel minimum and maximum, in order to normalize the pixel values within the region 0–255. The processes thus have to perform two *MPI_Allreduce()* operations after all their tasks have been processed, and before they start to normalize their data and write it to disk. Therefore, it is very important that all processors finish processing at the same time, before the collective *MPI_Allreduce()* operation.



(a) Input image, a CT scan.

(b) Output image w/50% threshold.

Figure 4.3: The noise in the input image makes the blood vessel identification quite challenging. Fig. (b) shows the output image after filtering, and with a 50% threshold. This image has been obtained with our parallel version of the algorithm. The output image shows that noise has been removed and blood vessels are now identifiable. CT image courtesy of Interventional Centre, Rikshospitalet, Oslo, Norway.

4.2 Master Implementation

The master process is implemented with three main modules, and thread-safe FIFO queues between them, see Figure 4.4. Due to our use of data staging, see Section 1.4.3, we found that the master was idle for extended periods of time. Because of this, we decided to let the master perform computations as well, in a separate worker thread.

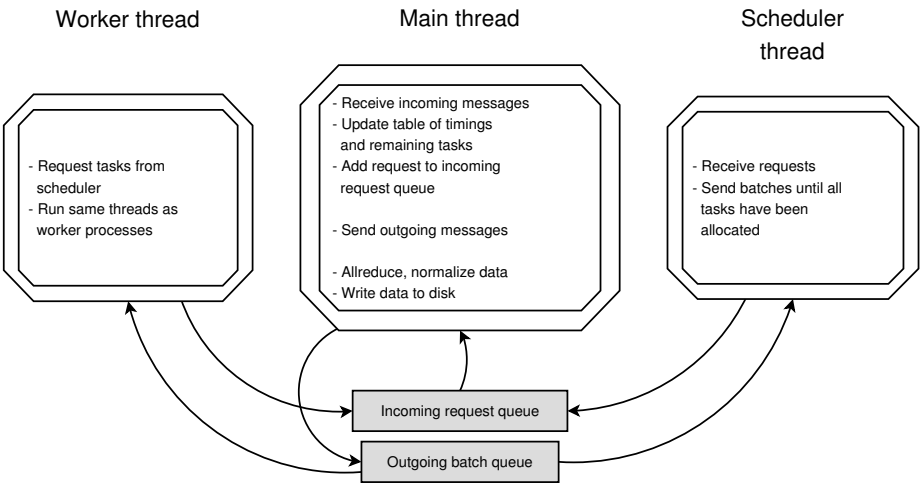


Figure 4.4: Threads in master process.

4.2.1 Main Thread

The main thread does some initializations, and opens the input file to read image metadata (height, width etc.), which it broadcasts to all nodes. Subsequently, the main thread starts the scheduler thread and the worker thread, and goes into a loop. In this loop, it probes for incoming messages from workers. If an incoming message is a request for work, the thread adds it to the incoming request queue, and if it is an update of the task computation time and number of uncomputed tasks, it updates the necessary tables. It also checks the outgoing message queue for messages, and sends them to workers.

When the scheduler thread signals that all tasks have been allocated, the main thread sends shutdown messages to workers, and breaks its loop. It then joins the scheduler thread and the worker thread.

The main thread then performs two *MPI_Allreduce()* operations, and normalizes the output data computed by the worker thread (a necessity of the image processing application, see Section 4.1). The output data from the worker thread is then written to disk, after which the application exits.

An ideal implementation of the master would have one thread for sending messages to workers, and one thread for receiving requests. These could use blocking I/O, and thus consume no CPU time when there are no messages to send or receive. However, MPICH [21], the MPI implementation we used, does

not allow for using MPI calls in two threads simultaneously. Thus, the main thread has to perform both sending and receiving.

This has a few important consequences. When the main thread has no messages to send and none to receive for extended periods of time, its loop calls *MPI_IProbe()* to check for incoming messages and checks the number of messages in the outgoing message queue restlessly. This consumes CPU time on the master, and makes the master's worker thread compute tasks slower. Still, with the current version of MPICH, this is our only option to compute tasks on the master at all, and the scheduling strategy in use must thus treat the master's worker thread as a slow worker.

We could of course have merged the main thread and scheduler thread (seen in Figure 4.4) into a single thread, but this would have had serious consequences for the modular architecture of our master implementation. By keeping the two threads as separate modules in the source code, we were able to implement the various scheduling strategies with great ease.

4.2.2 Scheduler Thread

We have produced implementations of the scheduler thread for the Static Chunking, Self-Scheduling, Guided Self-Scheduling, Trapezoid Self-Scheduling, Factoring, Weighted Factoring and Monitor strategies.

The scheduler thread typically enters a loop, receives requests in the incoming request queue, allocates tasks, and injects batches into the outgoing message queue.

4.2.3 Worker Thread

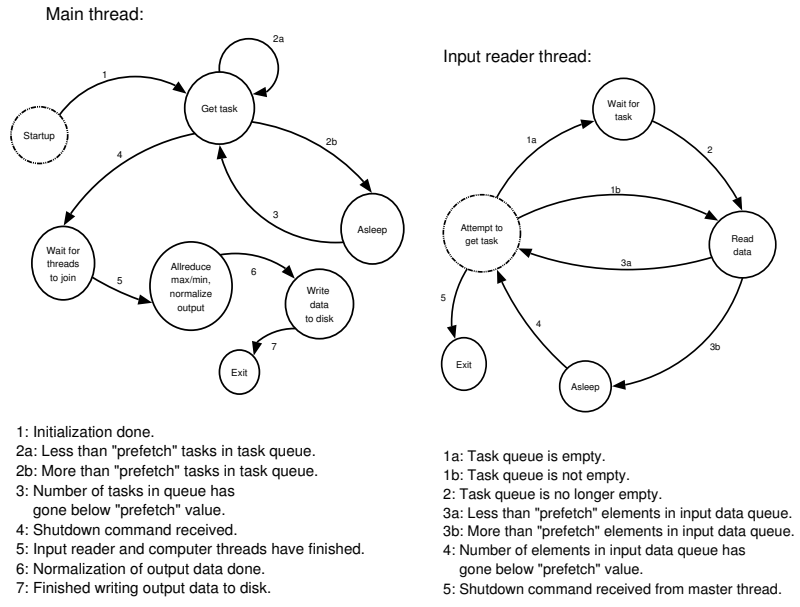
The master's worker thread works exactly as the worker processes described in Section 4.3, only that it communicates with the master by injecting requests direct into the master's incoming request queue. When the master allocates a task to its worker thread, it does not send the task through MPI, but injects it direct into the worker thread's task queue.

4.3 Worker Implementation

In order to avoid process idleness, we decided to implement a multi-threaded approach for the workers. A user defined value, ϕ, decides the size of the task queue and the input data queue. A worker will then request tasks from the master whenever the number of tasks in its task queue drops below ϕ. The goal is to keep the workers' computation threads busy at all times.

4.3.1 Main Thread

The main thread, see Figure 4.5a, starts two threads—one for reading input data from disk, and one for computing output data. The main thread keeps requesting tasks from the master, until the task queue is "full" (it has length ϕ). It then goes to sleep, and does not wake up again until the task queue's length is less than ϕ. When a special shutdown task is received from the master, it starts waiting for the input reader thread and the computer thread to finish,



(a) Worker process: Main thread.

(b) Worker process: Input reader thread.

Figure 4.5: Main thread and input reader thread in the worker processes.

performs two *MPI_Allreduce()* operations, normalizes the output data from the computer thread (see 4.1), writes the data to disk, and exits.

4.3.2 Input Reader Thread

The input reader thread, see Figure 4.5b, will fetch tasks from the task queue, and, if the queue is empty, sleep while waiting for a task. Once it has gotten a task, it starts reading input data into a buffer. A pointer to this buffer is added to the input data queue. This procedure is repeated until the input data queue has length ϕ. It then goes to sleep, and does not wake up again until the input data queue's length is less than ϕ, after which it takes the same actions again. When a special shutdown command is received in the task queue, the input reader thread exits.

4.3.3 Task Computer Thread

The computer thread, see Figure 4.6, will fetch input data from the input data queue, and, if the queue is empty, sleep while waiting for data. Once it has gotten input data, the computer thread computes tasks and adds its results to a result queue. When a special shutdown command is received in the input data queue, the computer thread exits.

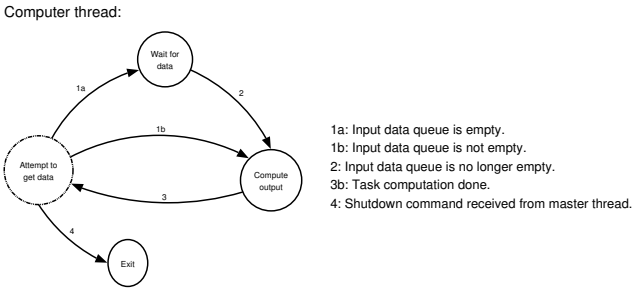


Figure 4.6: Worker process: Computer thread.

Chapter 5

Experiments

All experiments in this and the following chapters have been done multiple times on ClustIS, and all timing values are sums or mean values of the independent test runs.

5.1 Experimental Setup

ClustIS, described in Section 2.1, is a heterogeneous and dedicated cluster. This means that in order to test the various scheduling strategies in a dynamic environment, we have to simulate a varying background load on the nodes in use.

5.1.1 Static Environment

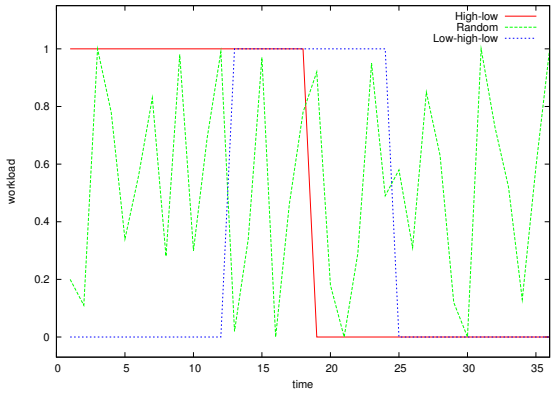
Testing in a static environment was very straight-forward—when submitting a job to the OpenPBS job scheduler, the necessary nodes are reserved, and they have a negligible background load unless a user specifically logs on to these nodes and runs serial applications. Our experience is that this more or less never happens on this cluster. However, data storage is provided by only one node (see Section 2.1.2), and all user processes on the cluster will be using this storage node simultaneously. There is not much we can do to prevent this—reserving the whole cluster is unfortunately not possible.

5.1.2 Dynamic Environment

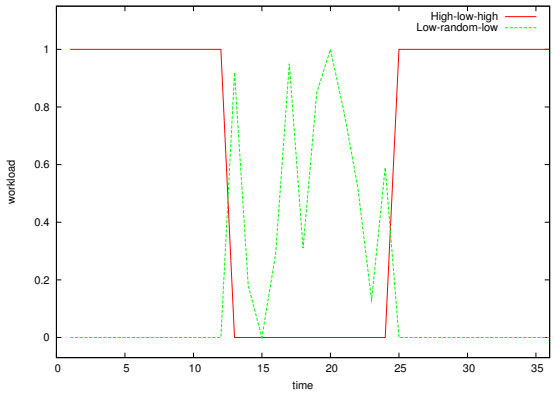
To simulate a dynamic environment, we created a workload simulator, a simple application that generates a workload pattern for a specified running time. A user-supplied parameter switches between a total of seven workload patterns, derived from three basic workload types. These three types are **high** (the application is running some trigonometric functions to ensure a high workload), **low** (the application is sleeping, using no CPU cycles), and **random** (a random number decides whether the application should sleep for one second or work for one second).

Examples of the seven workload patterns are depicted in Figure 5.1.

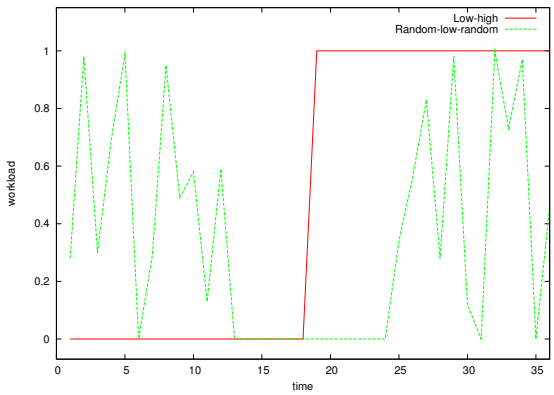
When running the experiments in our simulated dynamic environment, the seven workload patterns are run on nodes in a cyclic fashion—the first and the



(a) High-low, random and low-high-low.



(b) High-low-high and low-random-low.



(c) Low-high and random-low-random.

Figure 5.1: Workload patterns from the workload simulator.

eighth nodes get the first workload pattern, the second and the nineth nodes get the second workload pattern, etc.

5.2 Empirical Results

For the following experiments, we use 8 nodes and an image of 2048×2048 pixels. Unless otherwise noted, the experiments use a value of $\phi = 2$ tasks.

5.2.1 Optimal Task Size

We conducted a simple experiment to find the optimal task size. The application was run several times, with different task sizes—from 8×8 to 256×256 pixels, and the application's total running time using the Self-Scheduling strategy was measured. Figure 5.2 shows the results of the experiment. We can see that the application's total running time is minimized with a task size of 4096 pixels (a 64×64 block).

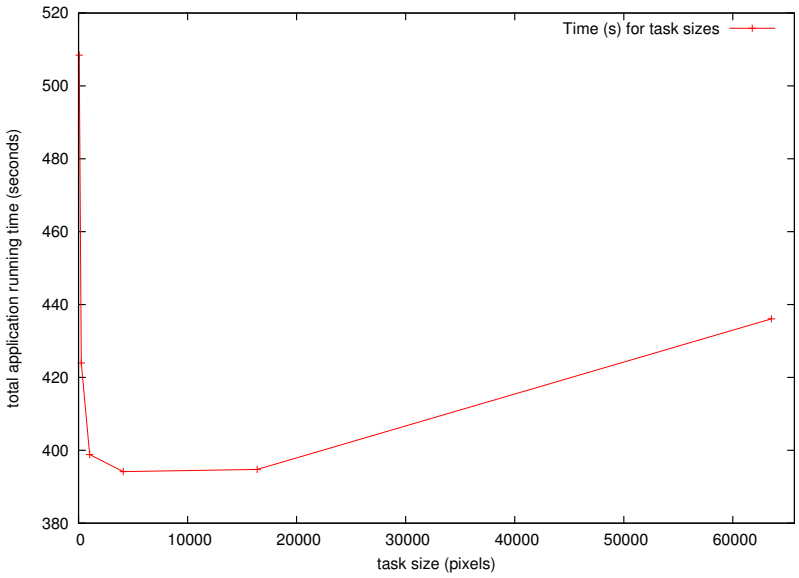


Figure 5.2: The application's total running time with varying task sizes.

In our following experiments, we will use a task size of 64×64 pixels. On many clusters, however, it might give better results if the task size can be adapted to the individual processors, since they might have different CPU cache sizes etc. This has not been implemented, but an approach is discussed in Section 7.3.

5.2.2 Parameters for the Monitor Strategy

The Monitor strategy has two tunable parameters, η and r. A worker's task computation times are reported on every request to the master, as well as after

every r computed task, and the reported values are mean values of the last η computation times. These two parameters have to be adjusted for the actual environment and application.

To find optimal values for these parameters in our setup, we conducted two experiments, one in a static environment, and the other in our simulated dynamic environment. The results can be seen in Figures 5.3 and 5.4. A general trend in both cases is that reporting timings often is better than reporting them seldom, which means that we should have a low r. The value for η, however, does not have a great impact as long as the timings are reported often enough.

We can see in both experiments that reporting timings every $r = 4$ tasks is ideal, and using $\eta = 20$ values gives good results in both cases. These are the values that we will use in the following experiments

5.2.3 Comparison of Scheduling Strategies

As noted in Section 4.3, our implementation uses a parameter ϕ which decides the size of the workers' task queues. This means that a worker will request tasks from the master as soon as the number of tasks in its queue drops below the value of ϕ.

Intuitively, a value of $\phi = 1$ means that a worker might become idle while waiting for the master to honor its request for tasks. At the other end of the spectrum, a too high value of ϕ means that a worker will request more tasks a long time before it is out of work. This will be non-optimal if the workers' speeds are very different, and one might get a situation where the faster workers will be waiting for the slower workers to finish.

The implemented scheduling strategies have been compared with increasing values of ϕ, both in a static and a dynamic environment. The results are shown in Figures 5.5, 5.6, 5.7 and 5.8.

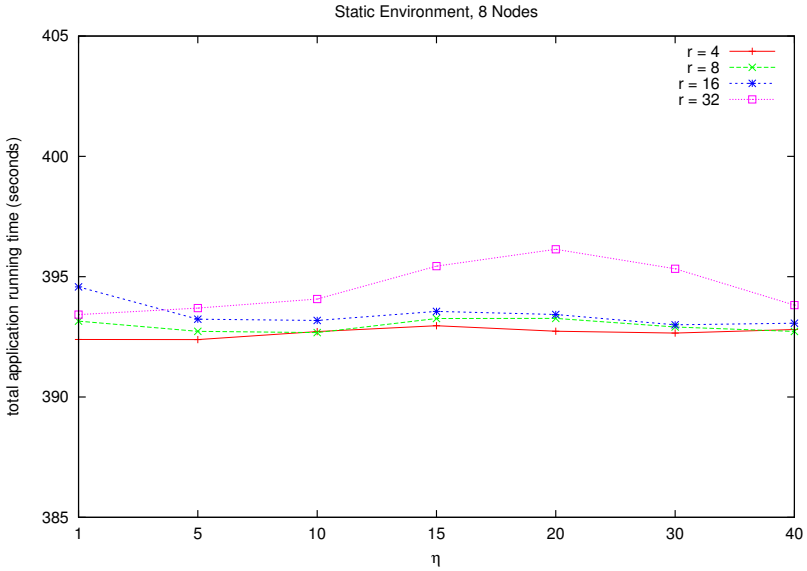


Figure 5.3: The graph shows the impact of varying r and η values on the application's total running time in a static environment. Image size 2048×2048 pixels, task size 64×64 pixels, 8 nodes. The results are as expected in a static environment—the difference between the fastest and the slowest run is only about 1%. The best running times are along the red line, when reporting timings as often as every 4th task.

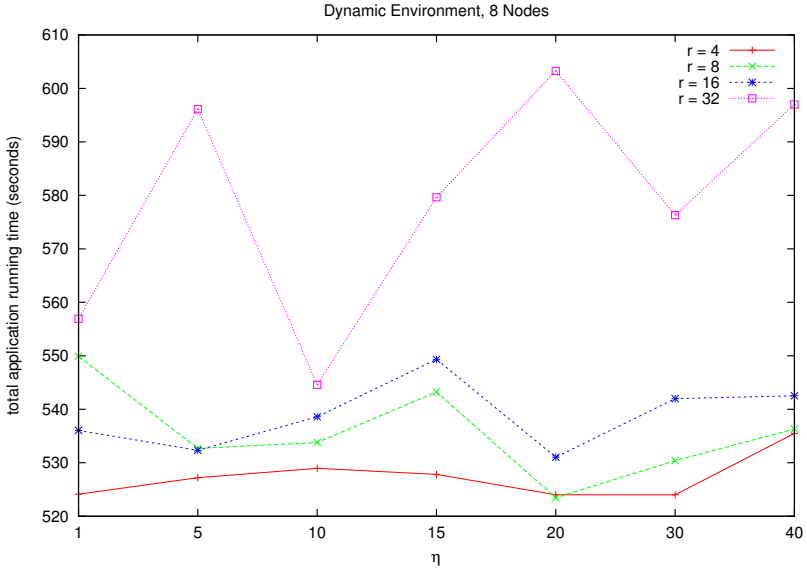


Figure 5.4: The graph shows the impact of varying r and η values on the application's total running time in our simulated dynamic environment. Image size 2048×2048 pixels, task size 64×64 pixels, 8 nodes. The results are as expected in a dynamic environment—they differ as much as 15%. The best running times are along the red line, when reporting timings as often as every 4th task.

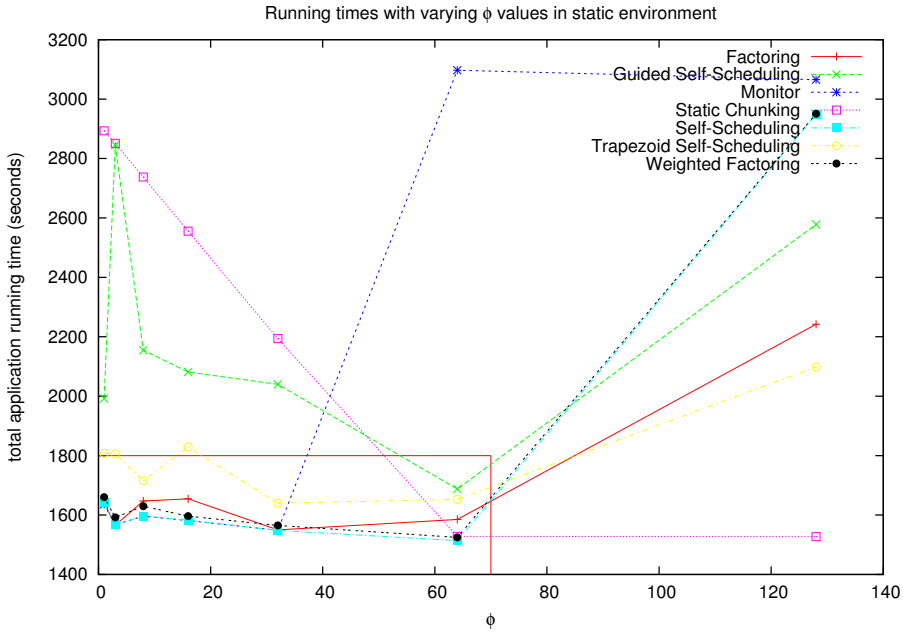


Figure 5.5: Comparison of scheduling strategies with increasing ϕ values, static environment. The area inside the red square is enlarged in Figure 5.6.

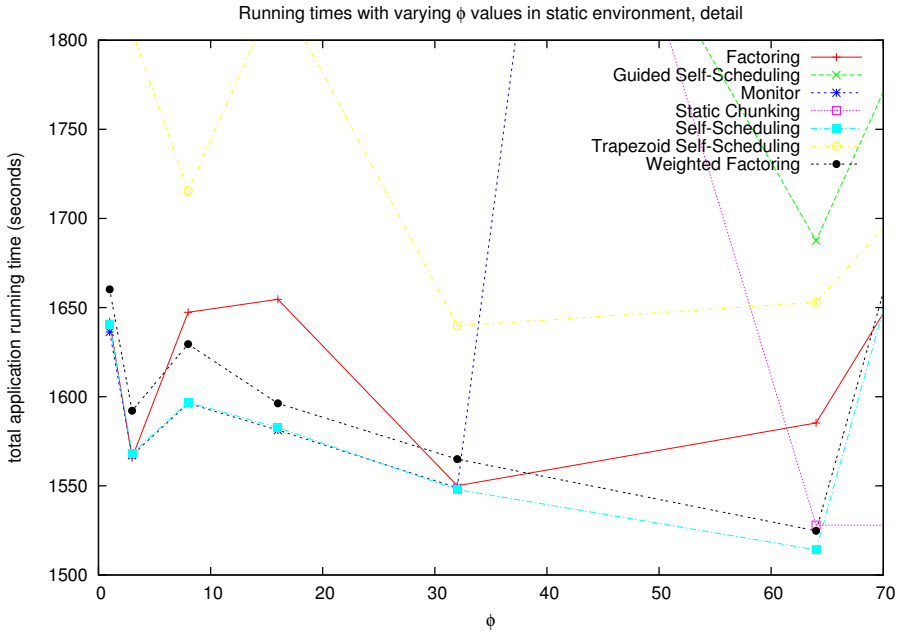


Figure 5.6: Detail of the fastest scheduling strategies from Figure 5.5.

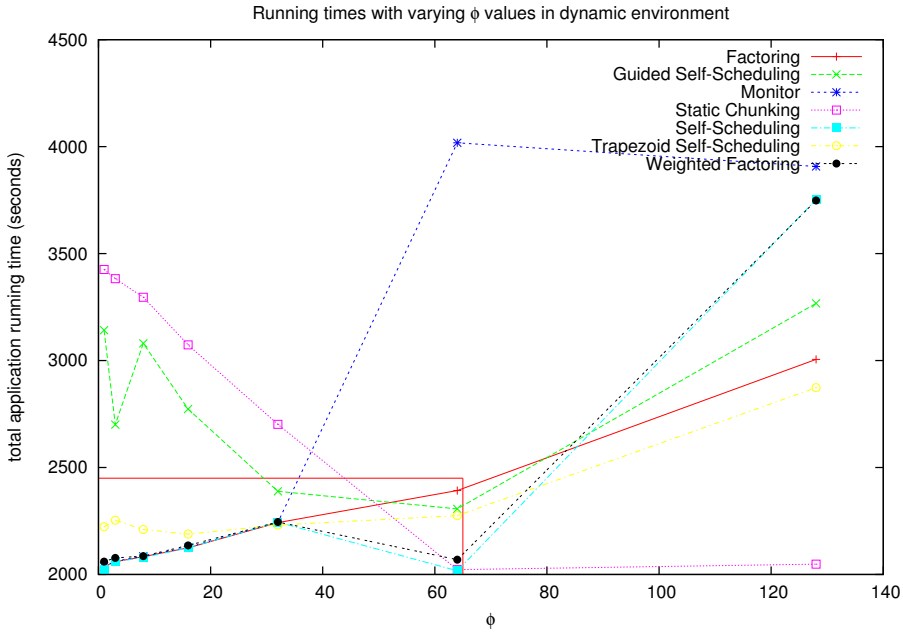


Figure 5.7: Comparison of scheduling strategies with increasing ϕ values, dynamic environment. The area inside the red square is enlarged in Figure 5.8.

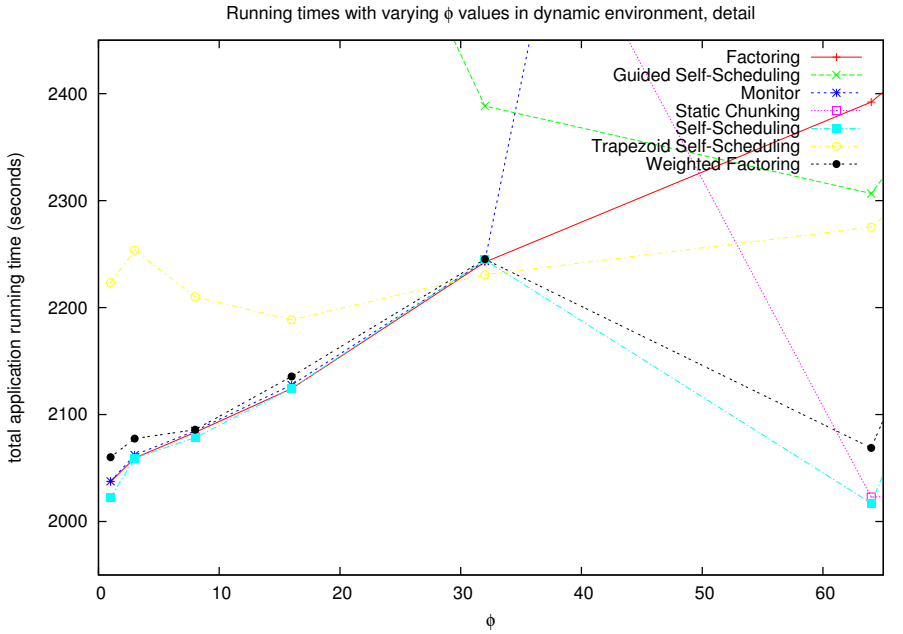


Figure 5.8: Detail of the fastest scheduling strategies from Figure 5.7.

Chapter 6

Analysis

6.1 Optimal Task Size

This section discusses the results of the experiments in Section 5.2.1.

6.1.1 CPU Cache

The optimal task size for a node depends very much on its CPU cache size. In our application, a task is a block from the input image, and its result is the corresponding block in the output image. The input image block's size is $b \times b$ pixels, and the pixel currently being computed is known as the *target pixel*, shown in black in Figure 4.1. When the images are stored row-by-row in memory, the pixel directly below the target pixel will be b pixels away from it, the pixel directly below that one again will be $2b$ pixels away, and so on. This means that memory access is non-contiguous, and if the entire block does not fit in the CPU cache, the algorithm will perform very badly.

If the input image, output image and filter arrays all fit in the CPU cache, performance will likely be much higher.

Fortunately, on our cluster, the nodes' CPU cache sizes are the same, so we won't have to adapt the task sizes to individual nodes. However, this would probably be necessary for good performance on many clusters, and an approach is outlined in Section 7.3.

6.1.2 Penalty Regarding Too Small Task Sizes

Another issue affecting the ideal task size, is the penalty involved with dividing the image too finely. As described in Section 4.1, one needs to read a border of $u/2$ pixels around each input image block, see Figure 4.2.

In Figure 6.1, the parallelized application would read 9 partially overlapping blocks of $(3+1+1) \times (3+1+1)$ pixels, a total of 225 pixels. If the application would filter the entire area as one task, it would read $(9+1+1) \times (9+1+1)$ pixels, a total of 100 pixels, less than half as much.

The optimal task size would thus be a trade-off between the requirements posed by the CPU cache (requires relatively small tasks) and the penalties involved when using too small tasks (suggests very large tasks).

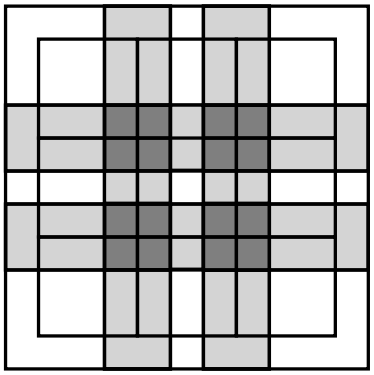


Figure 6.1: In this exaggerated example, the block size is 3×3 pixels and the filter is 3×3 pixels. Consequently, every block must have a 1-pixel frame around it. The white areas indicate data that has been read once, the light gray areas indicate data that has been read twice, and the dark gray areas indicate data that has been read four times into memory.

In our current implementation, the workers receive pointers to their data, and read each task into memory separately. A possible optimization would be to let each worker read all the data needed by its batch of tasks into memory first, and then copy this data into blocks of adequate size. In that case, data would never be read twice from disk unnecessarily, and the image could be filtered in blocks that would fit in the CPU cache.

To test the penalties of reading each task separately from disk, and to see the effects of the CPU cache, we conducted a simple experiment. We compared the performance of the SC strategy using the optimal task size found in Section 5.2.1 with the performance of the SC strategy when dividing the image evenly among the processors, with 1 large task per processor. Figure 6.2 shows the results.

Scheduling strategy	**Tot. time**	**Comm.**	**Comp.**	**I/O**
SC, task size 2048×256 px, 8 tasks	372.25	11.55	360.67	0.16
SC, task size 32×32 px, 1024 tasks	377.36	11.62	361.49	4.39

Figure 6.2: Comparison of running times with 128 tasks per worker or 1 task per worker.

As we can see, the difference in the total running time is only 1.4%, even though the second experiment reads considerably more data than the first. Intuitively, this probably means that the applications spends the majority of its running time computing the data, and only a fraction of its time is spent reading data into memory. The timings show that this is correct—the computation to I/O ratio is *very* high for this application.

The two runs spend almost the same time communicating and computing, but the second run spends 2600% more time reading and writing input data.

Still, since the application is multi-threaded, it seems that it is very capable of reading input data while computing at the same time. One would suspect, however, that the second run would spend less time computing, since it uses a block size more suitable for the CPU cache. This is not the case, and the cause is probably that the block size should have been even smaller, or that the image filtering code was poorly written with respect to cache optimizations.

6.2 Data Staging

To test the effects of our data staging implementation, we created another implementation of our application that does not use data staging, and compared the two. In the second implementation, the process with rank 0 reads the entire input image into memory, and scatters it onto all processes (including itself). The processes then compute their part of the image, and gather their results onto process 0, which normalizes the data and writes it to disk. The total running times of the two applications are shown in Figure 6.3. Note that for the master/worker application, the ϕ value was set to 128 to make the master's main thread and scheduler thread interfere as little as possible with the master's worker thread (see Section 6.5).

Scheduling strategy	**Total running time, 8 nodes**	
Master/worker, data staging, SC	372.25 s	100%
Scatter/gather, no data staging	374.30 s	100.6%

Figure 6.3: Comparison of running times between our master/worker implementation with data staging, and a scatter/gather implementation. Each of the 8 processes computes a task of 2048×256 pixels.

As we can see, the difference is marginal, only 0.6%. Our findings in the previous section show that the computation to I/O ratio for our application is *very* high, which means that the application will spend the vast majority of its time computing the data.

We've used the Jumpshot utility, [21], to create a graph of the call frequency of MPI functions throughout the applications' running times. These are shown in Figures 6.4 and 6.5. These graphs support our theory that the vast majority of the application's running time is spent computing data.

We can conclude that in our application, I/O is very quick compared to computations, and task sizes should hence be chosen mostly with regard to the maximized use of the CPU cache.

6.3 Consequences of Multi-Threaded Implementation

Our multi-threaded implementation of the worker processes creates some interesting results. We can see in Figures 5.3 and 5.4 that the Monitor strategy works best when reporting timings to the master often, even though this creates a lot of communication not directly related to the allocation of tasks. Also,

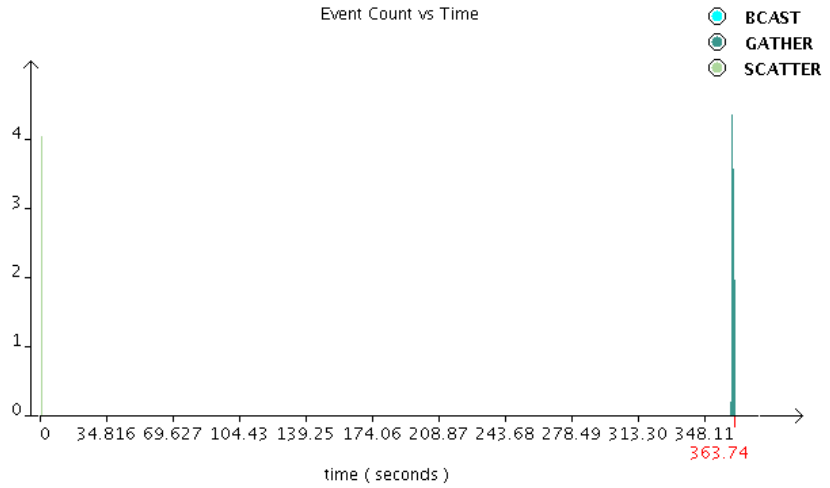


Figure 6.4: The graph shows the call frequency of the various MPI calls throughout the running time of the scatter/gather version of the application.

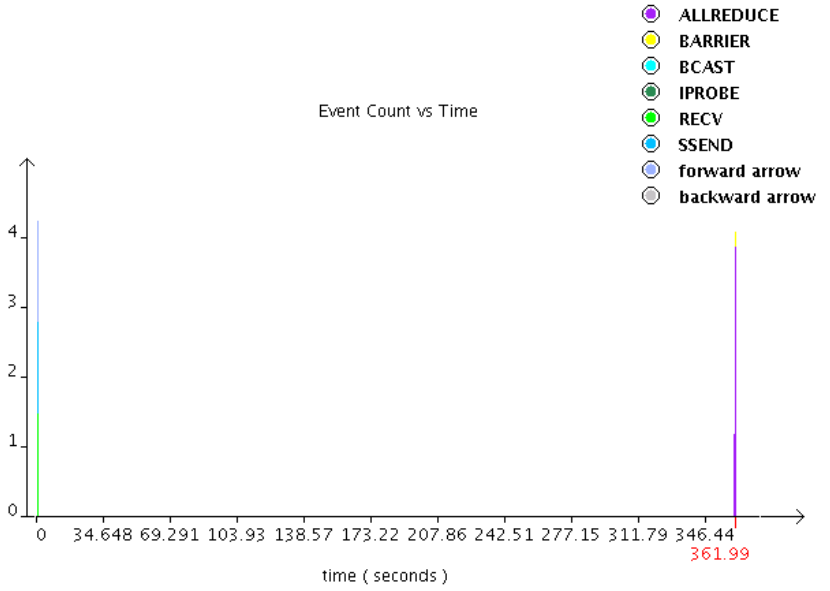


Figure 6.5: The graph shows the call frequency of the various MPI calls throughout the running time of the master/worker application, with $\phi = 128$.

Figures 5.6 and 5.8 show that the Self-Scheduling strategy clearly is among the fastest scheduling strategies, both in static and dynamic environments, despite the fact that this strategy has the highest amount of communication of all strategies.

These findings suggest that our multi-threaded implementation is able to *hide* the communication delays. The workers have one MPI communication thread and one input reader thread (see Section 4.3), that work in parallel to keep the task computation thread busy at all times.

In a single-threaded implementation, the experimental results clearly would be very different. The workers would have to send/receive data and compute data in a cyclic fashion. While communicating, a worker would be idle in all other respects. One possible improvement would be to use non-blocking communication, and call *MPI_ISend()* or *MPI_IRecv()* to initiate sending/receiving, and later call *MPI_Test()* to see if the operations had completed. However, achieving full parallelism between communicating, reading and computing would be much more complicated with this approach.

We believe that in a single-threaded implementation, the Monitor strategy would perform rather badly, as would the Self-Scheduling strategy.

6.4 Parameters for the Monitor Strategy

This section discusses the results of the experiments with the Monitor strategy in Section 5.2.2.

Workers report their task computation times every r tasks, as well as when they request a task. A low r value thus makes the workers report timings often, which generates a lot of communication between the master and workers. When our multi-threaded implementation is able to hide the penalties of communication, as discussed in Section 6.3, this might not be a problem for the workers. The master, however, might become congested if the communication load is too high, that is, too many workers report their timings too often. In this scenario, the master might not be able to allocate the tasks adequately, because of too many status updates.

Our experiments show that both in a static and a dynamic environment, reporting timings every $r = 4$ tasks is adequate when using 8 processes. All higher r values give worse results. The value of $r = 4$ is the lowest value we have tested, but we believe that using lower values might overload the master with requests.

The workers report timings which are mean values of their η last task computation times. Intuitively, $\eta = 1$ means that a worker reports only its last task computation time to the master. In a dynamic environment, the computation times can vary significantly, and having the master allocate a whole batch of tasks based on this single timing value might be unwise. At the other end of the spectrum, a high η value means that a worker will compute a mean value of many task computation times whenever it reports to the master. This might conceal changes in processing speeds, which is also non-optimal.

Our experiments show, however, that as long as the timings are reported often enough, η does not have a great impact. We chose to use $\eta = 20$ in our further experiments, since this gives good results both in a static and a dynamic environment, and seems to be a good trade-off between including too few and

too many values in the mean values reported.

6.5 Comparison of Scheduling Strategies

This section discusses the results of the comparison of the scheduling strategies in Section 5.2.3.

Using a low value for ϕ, e.g. $\phi = 1$, means that the workers' main threads will request a batch from the master when their task queues are empty. With very large values of ϕ, the workers would request tasks a very long time before they actually need them, and with large enough ϕ values, all tasks would be allocated in the beginning of the computation. This affects the performance of the various scheduling strategies differently.

However, a few effects are common to all scheduling strategies. Using a very large value for ϕ mainly implies two things. For the workers, it implies that their input reader threads would read all their tasks into memory very quickly, and the task computation thread would run without interference for most of the application's running time. For the master, the scheduler thread would allocate all tasks in the beginning of the run, and the main thread would spend only a very short amount of time sending/receiving messages. Thus, the master's worker thread would not have to compete for CPU time (see Section 4.2.1 for more about this issue). Thus, a large ϕ value has a positive effect on the task computation speeds both for the master and for the workers.

There are a few common effects when using a small ϕ value as well. The workers' main threads and input reader threads will be running throughout the entire computation, but they will sleep (and consume no CPU time) as long as the number of tasks in their queues is above ϕ. This makes the master's main thread run constantly waiting for incoming and outgoing messages, and if the master's scheduler thread is handing out large batches to workers, the main thread will run for extended periods of time and be of no use—it will only consume CPU time and interfere with the master's worker thread.

6.5.1 Static Chunking

One interesting finding in Figures 5.5 and 5.7 is that the Static Chunking strategy performs linearly better when using a larger ϕ value, both in a static environment and in a dynamic environment. We are using 8 processors and 1024 tasks, which would mean that each worker is allocated 128 tasks with the SC strategy. With $\phi = 64$ and above, the SC strategy seems to have reached a minimum running time.

The SC strategy obviously has the same task allocation to workers in all cases, so the improvement in total running time cannot come from an improved allocation of tasks to workers. It is well known that the SC strategy handles a heterogeneous environment rather badly, and with low ϕ values, the master's main thread and scheduler thread run continuously, so the master is exceptionally slow at computing tasks (see Section 4.2.1). Other scheduling strategies handle this heterogeneity quite well by allocating less tasks to the master's worker thread, but when using the SC strategy, all workers have to wait for the slowest worker to finish.

The reason why the SC strategy performs so well when using large ϕ values, is that the master's main thread and scheduler thread finish allocating tasks and sending/receiving messages in the beginning of the computation, and thus, the master's worker thread becomes equally fast as the other workers (if one does not regard the possible background processing load).

We used the Jumpshot utility, [21], to create graphs of the frequency of MPI calls throughout the application's running time. However, due to the high number of calls to *MPI_IProbe()* made by the master, we had severe problems creating such graphs from the full-size experiments described in the previous section (the log files grew too large, several gigabytes). However, Figures 6.7 and 6.6 show examples that support our theory. They show data from runs with the master and only two workers, a very small image of only 128×128 pixels, and $\phi = 128$ and $\phi = 1$, respectively.

Figure 6.6 shows a high activity of *MPI_IProbe()* throughout the entire computation, which takes 9.5 seconds. We can see in Figure 6.7 that with $\phi = 128$ there are almost no calls to *MPI_IProbe()*, which makes the computation go significantly faster, about 5.5 seconds.

Thus, it is fair to say that large ϕ values unleash more computing power, by letting the master's main thread and scheduler thread finish quickly. This would be very different if the master had been implemented with separate threads for send/receive, see Section 4.2.1.

6.5.2 Monitor

With ϕ values of up to 32, the Monitor strategy is among the very fastest scheduling strategies, both in a static and dynamic environment.

With $\phi > 32$, the Monitor strategy performs very badly. This is because batches are allocated in a weighted fashion, based on timings from very early in the computation. For the Monitor strategy to perform well, the master must allocate tasks throughout the computation, based on recent values of the workers' task computation times.

6.5.3 Self-Scheduling

It is quite surprising that the Self-Scheduling strategy, which has the highest amount of communication of all strategies, is among the very fastest scheduling strategies. This is because our multi-threaded implementation is able to hide the communication delays, as discussed in Section 6.3.

The SS strategy thus gives a quite optimal allocation of tasks.

6.5.4 Other Scheduling Strategies

For the other tested scheduling strategies, the general trend is that they perform well with ϕ values up until a certain point. All the scheduling strategies except Static Chunking perform very badly with high ϕ values. This is because the workers request tasks a very long time before they are actually needed, and the load is thereby excessively mis-allocated.

Due to the effects of having a large ϕ value with regard to the master's computing speed, as discussed in Section 6.5.1, it is not entirely fair to compare

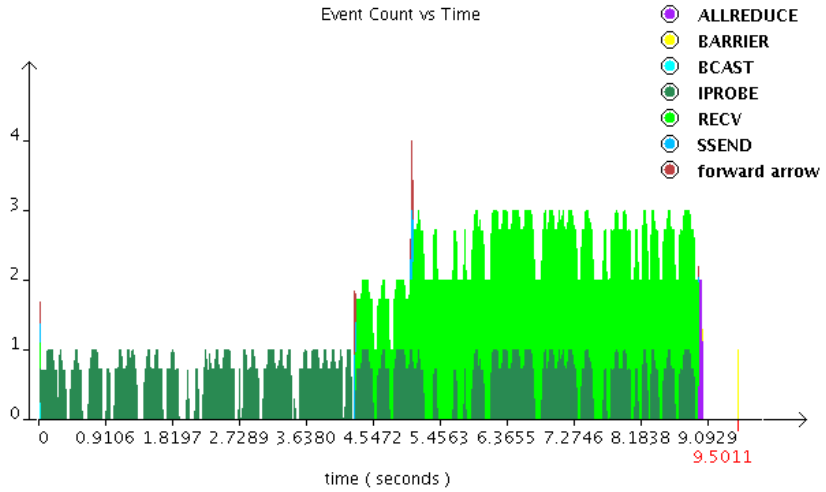


Figure 6.6: The graph shows the call frequency of the various MPI calls throughout the application's running time, with $\phi = 1$.

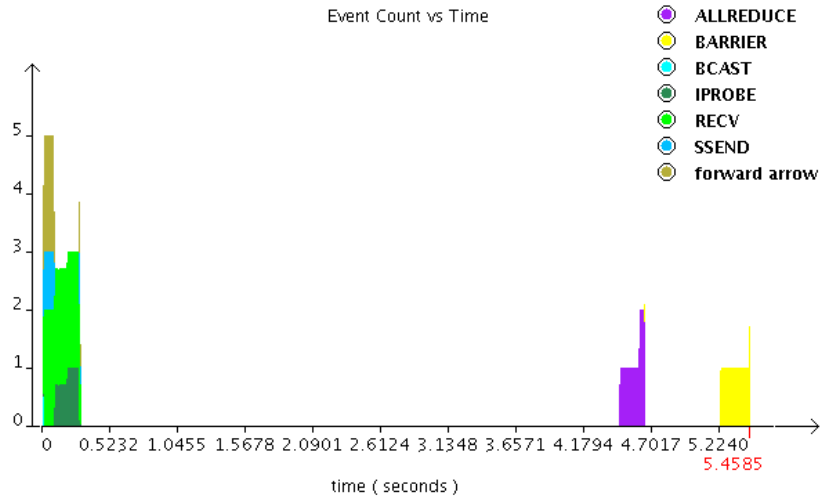


Figure 6.7: The graph shows the call frequency of the various MPI calls throughout the application's running time, with $\phi = 128$.

SC's running time at $\phi \geq 64$ with the running times of the other scheduling strategies.

With $\phi < 64$, the scheduling strategies have to allocate tasks to a heterogeneous set of workers, including the quite slow master's worker thread. With $\phi > 64$, the set of workers becomes much more homogeneous, because the master's worker thread speeds up drastically. In this perspective, it is quite impressive that the SS and Monitor strategies (and in part, Factoring and Weighted Factoring) with $\phi < 64$ are able to achieve a running time comparable to SC with $\phi \geq 64$.

6.5.5 Conclusive Remark

As a conclusive remark, it must be said that actually using a scheduling strategy with a very high ϕ value is of little interest. It is contrary to the basic idea of adequate scheduling that all tasks are allocated in the beginning of the computation, which is the consequence of a high ϕ value. Tasks should be allocated when the workers need them (or slightly before, to avoid idle time), and thus, ϕ must be kept relatively low.

The effect seen in the discussion of SC's performance is only an implementation-specific issue, and would be very different with an improved implementation of the master, or when computing tasks only on worker nodes. This is discussed further in the next chapter.

Chapter 7

Discussion

7.1 Improved Multi-Threaded Master Implementation

As we have seen in the analysis of the Static Chunking strategy's performance (Section 6.5.1), the implementation of the master's communication thread has a significant impact on the overall performance of the application.

A better implementation of the master's communication needs would be to have two threads, one for sending and one for receiving, as described in Section 4.2.1. With a future version of the MPICH library, or other MPI implementations, this will be possible. This would affect the performance of the scheduling strategies differently.

With the present implementation, the SC strategy seems to have reached a minimum running time with $\phi \geq 64$. With a new master implementation, we believe that the SC strategy would run at this speed regardless of ϕ value, as suggested by Figure 6.7.

With relatively low ϕ values ($\phi \leq 32$), the other scheduling strategies perform well at the present time. With a new master implementation, we believe that they will perform even better, but not significantly better. These scheduling strategies all adapt the total allocation of tasks based on the sequence that the workers request work (as opposed to SC), and the master's performance would improve the total running time only slightly.

However, as noted in the previous section, very high ϕ values have little practical interest. Tasks should be allocated to workers when they are needed, in several rounds.

7.2 Optimal Scheduling Strategy

Experiments with very high ϕ values serve only to illustrate the issues regarding our master implementation. To achieve adequate scheduling, one must use relatively low ϕ values. With $\phi = 3$, the Self-Scheduling, Monitor and Factoring strategies perform equally well both in a static environment and in a dynamic environment.

The fact that the Self-Scheduling strategy performs so well stems from the

effects of our multi-threaded implementation, as discussed in Section 6.3. In the literature of parallel scheduling, SS is known to perform very badly as long as the communication delays are non-negligible, [14]. This changes significantly when using a multi-threaded architecture.

It is also interesting to see that the Monitor strategy performs very well in both a static and a dynamic environment. This shows that our basic assumptions were correct, and that scheduling a batch of tasks based on the timings for the last η tasks is a good idea.

Because of the very small efforts needed to implement it, and its excellent performance in both static and dynamic environments when using a multi-threaded architecture, we would recommend using the SS strategy on a cluster.

In a single-threaded architecture, however, both the SS and the Monitor strategy would perform very badly. In such a scenario we would recommend the Factoring strategy.

7.3 Auto-Tuning of Optimal Task Size

An application's optimal task size is dependant on the nodes' CPU caches, but when using a job scheduler on a cluster, one cannot know which nodes will be reserved for running the computation. An auto-tuning procedure, which determines the optimal task size online, would thus be desirable. We will here present two such procedures, one for heterogeneous environments and one for homogeneous environments.

7.3.1 Heterogeneous Environment

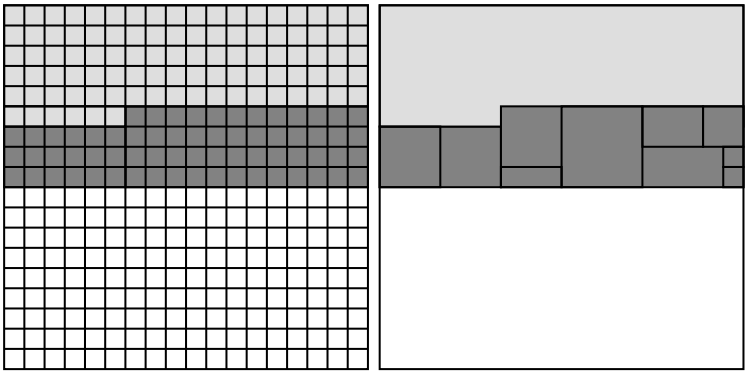
In a severely heterogeneous environment, the processors would have different CPU cache sizes. We will here briefly present a strategy for letting the workers determine their own optimal task size.

In our current implementation, the master allocates a batch of tasks to a worker w_i, which is a set of blocks in the input and output images. In Figure 7.1a, w_i has been allocated the dark gray blocks, and will compute them one-by-one.

If the block size is too large for worker w_i's CPU cache, the worker will compute the tasks very slowly due to cache misses. For worker w_i, it would be better with a smaller task size.

This can be achieved the following way: Workers are not allocated a batch of tasks with a fixed size, but instead, they are allocated an area in the input and output image. When the workers receive their first area, they find their optimal block size by dividing the area into blocks of various sizes, and measuring the time it takes to process the blocks. Figure 7.1b illustrates the approach. For the following rounds, the workers compute their allocated areas using their individual optimal block sizes.

Note that when using the Monitor strategy, the workers must report the time it takes to compute e.g. 100 pixels in the image, and not one block, since the workers use blocks of different sizes.



(a) The master allocates blocks of the image to requesting workers. The workers compute the blocks one-by-one.

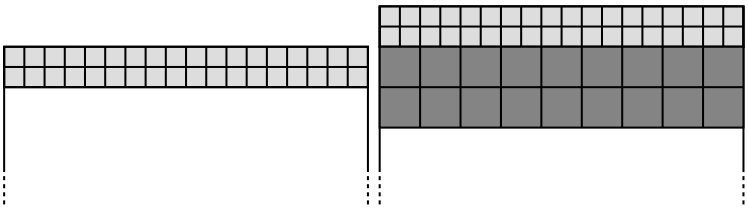
(b) Instead of being allocated a batch of fixed-size blocks, a worker can instead be allocated an area in the image and determine its optimal block size with regard to its CPU cache size.

Figure 7.1: Alternative ways of allocating image areas to workers.

7.3.2 Homogeneous Environment

In a homogeneous environment, one might be able to assume that the workers have the same CPU cache sizes, and the procedure presented in the previous section is thus unnecessary.

Determining the optimal task size online in a homogeneous environment can be achieved in the following way: The master initially allocates tasks of a relatively small size $s \times s$ from one horizontal line in the image, as seen in Figure 7.2a.



(a) The master initially allocates relatively small blocks from a horizontal line in the image.

(b) In the second round, the master allocates slightly larger blocks from a horizontal line in the image.

Figure 7.2: Approach for determining optimal common block size.

When all workers have finished computing their batches, they report their

timings to the master, which sums them into a total time T_s. The master then determines the processing speed in terms of pixels per second, v_s.

In the following round, the master allocates batches of tasks of size $t \times t$, which are slightly larger, as seen in Figure 7.2b.

Similarly, the master receives the workers' processing times, sums them into T_t, and computes the processing speed v_t.

Now, the master computes $\delta = v_s - v_t$. If $\delta > 0$, a block size of $s \times s$ is better, and the master will try a smaller block size in the next round. If $\delta < 0$, a block size of $t \times t$ is better, and the master will try a block size slightly greater in the next round. When $\delta \approx 0$, the two block sizes are equally fast, and the optimal block size is found.

Chapter 8

Conclusion

We have designed and implemented a scheduling strategy of our own, the Monitor strategy, and experimentally compared it to implementations of six other scheduling strategies found in the literature. Our experiments show that the Monitor strategy performs excellently compared to all the implemented previous strategies.

Our implementation has involved multi-threading and data staging, and both techniques have worked well to decrease processor idle time and increase master utilization.

We have shown that our test case application, the matched filtering algorithm, has a very high computation to I/O ratio, and that data staging probably isn't necessary for this application. Tests on our cluster show, however, that data staging is a valuable technique for applications whose computation to I/O ratio is lower.

We have shown that our multi-threaded implementation of the worker processes, using task queueing mechanisms, is able to hide communication delays and keep the workers processing data continuously. The excellent performance of both the Self-Scheduling and the Monitor strategy both substantiate this.

Based on our experimental findings, we would recommend using the Self-Scheduling strategy, because of its relatively effortless implementation, in combination with the use of data staging and a multi-threaded worker implementation using task queues, when scheduling bag-of-tasks applications on clusters.

Chapter 9

Future Work

Section 1.2.1 mentions fault-tolerance as a motivation behind the master/worker paradigm. This has not been included in our reference implementation, and would be a welcome addition.

The optimal task size has been found experimentally in this thesis. Section 7.3 discusses two procedures for determining the optimal task size online, but these have not been implemented. They would, however, have increased the usefulness of the application.

The Monitor strategy has two user-specifiable parameters, r and η. A procedure to determine the optimal values of these parameters while the application is running would be desirable.

In this experimental study, we have only tested our implementation on one cluster, ClustIS. Verifying the results on other clusters would be the next thing to do.

A thread-safe MPI library would enable us to implement the master process quite differently. This would increase performance, although not significantly for other strategies than Static Chunking. However, changing the master implementation requires relatively little effort.

Bibliography

[1] Para '04 Workshop on State-of-the-Art in Scientific Computing. http://www.imm.dtu.dk/~jw/para04/, June 20–23 2004.

[2] Altair Grid Technologies. OpenPBS. http://www.openpbs.org/, 2004.

[3] J. Basney, R. Raman, and M. Livny. High Throughput Monte Carlo, 1999.

[4] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems.* Computer Society, Aug 1996.

[5] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. *Cluster Computing*, 6(1), 2003.

[6] J. Blazewicz, M. Drozdowski, and M. Markiewicz. Divisible task scheduling—Concept and verification. *Parallel Computing*, 25:87–98, January 1999.

[7] S. Chaudhuri, S. Chatterjee, N. Katz, M. Nelson, and M. Goldbaum. Detection of Blood Vessels in Retinal Images Using Two-Dimensional Matched Filters. *IEEE Trans. on Med. Imaging*, 8(3):263–269, 1989.

[8] Frank Dabelstein and Peter Skaarup. Measuring Execution Characteristics of MPI Master/Slave Programs. Master's thesis, University of Aarhus, May 2003. http://www.daimi.au.dk/~piparum/download/master.pdf.

[9] M. Drozdowski and P. Wolniewicz. Experiments with Scheduling Divisible Tasks in Clusters of Workstations. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 311–319. Springer-Verlag, 2000.

[10] W. Elwasif, J. S. Plank, and R. Wolski. Data Staging Effects in Wide Area Task Farming Applications. In *IEEE Int'l Symposium on Cluster Computing and the Grid*, pages 122–129, Brisbane, Australia, May 2001.

[11] Free Software Foundation. The GNU General Public License. http://www.gnu.org/licenses/.

[12] Free Software Foundation. The GNU Scientific Library. http://www.gnu.org/software/gsl/.

[13] Richard Gerber. *The Software Optimization Cookbook.* Intel Press, 2002.

[14] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *Journ. of Parallel and Distr. Computing*, (47):185–197, 1997.

[15] Bryan Henderson. The NetPBM library. `http://netpbm.sourceforge.net/`.

[16] Robin Holtet. Developing Programming Examples for Parallel Computing Course. Technical report, NTNU, 2002.

[17] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Comm. of the ACM*, 35(8):90–101, Aug. 1992.

[18] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328. ACM Press, 1996.

[19] Intel Corporation. Intel Compilers. `http://www.intel.com/software/products/compilers/`.

[20] C. P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Trans. on Software Eng.*, 11(10):1001–1016, Oct. 1985.

[21] MPICH. `http://www.mcs.anl.gov/mpi/mpich/`. Argonne National Laboratory.

[22] NOTUR. `http://www.notur.org/`. The Norwegian HPC Consortium.

[23] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[24] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. on Comp.*, 36(12):1425–1439, Dec. 1987.

[25] Source Mage GNU/Linux. `http://www.sourcemage.org/`, 2004.

[26] T. G. Robertazzi. Ten Reasons to Use Divisible Load Theory. *Computer*, 36(05):63–68, 2003.

[27] NOTUR. Preparing for the Future HPC Infrastructure for Computational Science in Norway, December 2003.

[28] The Free Software Foundation. The GNU C library. `http://www.gnu.org/software/libc/`.

[29] The Free Software Foundation. The GNU Compiler Collection. `http://gcc.gnu.org/`.

[30] T. H. Tzen and L. M. Ni. Dynamic Loop Scheduling for Shared-Memory Multiprocessors. In *Proc. of the 1991 Int'l Conference on Parallel Processing*, pages II247–II250. IEEE Computer Society, 1991.

[31] Dimitri van Heesch. Doxygen. `http://www.doxygen.org/`.

[32] Veeravalli, Bharadwaj, Ranganath, and Surendra. Theoretical and experimental study on large size image processing applications using divisible load paradigm on distributed bus networks. *Image and Vision Computing*, 20(13–14):917–935, December 2002.

Appendix A

Source Code

The source code of the entire application is included on the CD attached to the back cover of this thesis. Note that the source code is licensed under the GNU General Public License, [11].

The code was developed on the cluster described in Chapter 2. It was compiled with the GNU Compiler Collection (gcc) version 3.3.1, and makes use of the following open-source libraries (besides the GNU C library):

- The GNU Scientific Library (GSL), [12], for solving the linear equations in the Monitor strategy.
- NetPBM, [15], a library for reading and writing the Portable Anymap (PNM) family of file formats. These are uncompressed binary image formats suitable for many scientific applications.

The code was written to be very portable, and with only minor modifications one should be able to compile and run it on most platforms that have the necessary libraries and compilers.

The code is documented with Doxygen, [31], and the documentation is included on the attached CD in HTML, LaTeX 2_ε and PDF format.

Note that since the NetPBM library was only available on the master node on ClustIS, we had to copy the source files *pgm_metadata.c* and *pgm_metadata.h* from the NetPBM source code into our source tree. These are unnecessary when the program is compiled on a system where the NetPBM library is available.

A.1 Matched Filtering Algorithm

The following Matlab code was written by Ole Christian Eidheim, and served as a base for implementing the same algorithm in C. The algorithm was first implemented as a serial application, and the most important functions were reused in the parallel version.

```
function R=matchedfilter(I);
%Executes matched filtering.

% Create a gaussian hill filter.
x=-10:10;
```

```
for y=1:21
    M(y,1:21)=gaussmf(x, [5, 0]);
end

% The result of using matched filters on a homogenous
% region should be zero. This is achived by subtracting
% the average from the filter.

avg=0;
for x=1:21
    avg=avg+M(1, x);
end

avg=avg/21;

M=M-avg;
M=M/21;

% Correlate the image I and the matched filter M while
% scaling and rotating filter M. The results are summed.

R=double(zeros(size(I)));

for size=0.2:0.1:0.8
    for ang=6:6:180
        R=double(imfilter(I, imresize(imrotate(M, ang), size)))
          +double(R);
    end
end
```

A.2 Compiling

As described in Section 4.2, the implementation of the master process is very modular. The included Makefile has seven targets, one for each scheduling strategy, which all build an executable file of their own. The only difference between them, is that they compile and link in their own implementation of the scheduling strategy module, all of which follow a common interface to the rest of the master code.

The default target in the Makefile compiles executables for all scheduling strategies.

A.3 Command-Line Arguments

The command-line arguments to the application are presented in Figure A.1. Note that the argument parser module is shared between all scheduling strategy implementations, and not all arguments are relevant in all scheduling strategies.

Argument	**Meaning**
--infile N	Input file name (image in PNM format).
--outfile N	Output file name (image in PNM format).
--prefetch N	Value of ϕ, for the workers' queues. Default value 2 if unspecified.
--xdiv N	Number of divisions in X direction. Default value 4 if unspecified.
--ydiv N	Number of divisions in Y direction. Default value 4 if unspecified.
--timing-updates N	Value of r (for Monitor strategy). Default value 0 (no updates) if unspecified.
--update-timings-once	Used to report timings to master only once (for Weighted Factoring).
--avg N	Value of η (for Monitor strategy). Default value 10 if unspecified.

Figure A.1: Command-line arguments to the application.