ABSTRACT

A lot of research has been conducted for studying cooperation in distributed systems. The abstract problem of performing a set of tasks by a set of distributed fault-prone processors is generally known as DO-ALL. In partitionable networks DO-ALL is known as OMNI-DO. Despite the active research on this subject, a lot of it remains theoretical and there are not any empirical studies on proposed solutions and their behaviour in realistic environments. We investigate recent research on this subject and implement an algorithm proposed for solving the OMNI-DO problem. The algorithm uses a group communication service to handle processor coordination when regroupings occur due to dynamic changes in the underlying network structure. In this thesis Ensemble GCS is studied and used in the implementation. A coordinator based approach is used for dissemination of knowledge, regarding completed tasks, within a group. A naturally random load balancing rule is used for inter-group task scheduling. Finally, we empirically evaluate the algorithm with respect to work, message and execution time metrics. The algorithm performs well and our results fall within the results of the theoretical analysis. An additional overhead during regroupings is identified (caused by the way new groups are formed by the group communication service) and an implementation solution is proposed.

IMPLEMENTATION AND EVALUATION OF AN ALGORITHM FOR THE OMNI-DO PROBLEM USING THE ENSEMBLE GROUP COMMUNICATION SERVICE

Ioanna Savva

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

At the

University of Cyprus

Recommended for Acceptance

By the Department of Computer Science

December, 2009

APPROVAL PAGE

Master of Science Thesis

IMPLEMENTATION AND EVALUATION OF AN ALGORITHM FOR THE OMNI-DO PROBLEM USING THE ENSEMBLE GROUP COMMUNICATION SERVICE

Presented by

Ioanna Savva

Research Supervisor

____

Chryssis Georgiou, Assistant Professor

Committee Member

____

Anna Philippou, Assistant Professor

Committee Member

____

George Pallis, Visiting Lecturer

University of Cyprus

December, 2009

ACKNOWLEDGEMENTS

I am grateful to many people for help, both direct and indirect, in concluding this thesis. First, I would like to thank my research supervisor, Assistant Professor Chryssis Georgiou, for believing in me and providing me the opportunity to work with him. He was always willing to offer motivation, guidance and even editing assistance in writing this thesis. I would also like to thank my friends and fellow students from my undergraduate years for brainstorming with me and providing assistance whenever they could. In alphabetical order they are: Andreas Savva, Angelos Constantinides, Pantelis Loizou and Stelios Erwtokritou. Finally, I would also like to thank my family for bearing with me, believing in me and supporting me all through these years.

TABLE OF CONTENTS

Chapter 1 .... **1**

Introduction **1**

1.1 Motivation and Related Work 1

1.2 Contribution 3

1.3 Document Structure 4

Chapter 2 .... **5**

Background **5**

2.1 Group Communication Services 5

2.1.1 Membership Service Safety Properties 7

2.1.2 Multicast Service Safety Properties 8

2.1.3 Properties for Safe Messages 9

2.1.4 Properties for Ordered and Reliable Services 10

2.1.5 Liveness Properties 11

2.1.6 Interaction primitives 13

2.2 DO-ALL and OMNI-DO problems 14

Chapter 3 .... **18**

The OMNI-DO Algorithm **18**

3.1 General Algorithm Description 18

3.2 Task Allocation Methods 19

3.3 Purpose of GCS 22

Chapter 4 .... **25**

The Ensemble Group Communication Service **25**

4.1 Description 25

4.2 Architecture 26

4.3 Characteristics 27

4.4 Installation & Compilation Instructions 28

4.5 Local processors 31

Chapter 5 33

Algorithm Implementation 33

5.1 C Language API 33

5.2 Tasks 35

5.3 Task Allocation 37

5.4 Messages 38

5.5 Regrouping mechanisms 40

5.5.1 Arbitrary Pattern 41

5.5.2 Specific Patterns (SGSM and OUG) 42

5.6 Client Logfiles 44

5.7 Compiling the Application 45

Chapter 6 48

Empirical Evaluation 48

6.1 Experimentation setting 48

6.2 Experiment 1: Effect of the number of Processors (P) and Tasks (N) 49

6.2.1 Scenario 1.1: Number of Processors 50

6.2.2 Scenario 1.2: Number of tasks 54

6.2.3 Summary 60

6.3 Experiment 2: Effect of Group_Max Variable 61

6.3.1 Scenario 2.1: Concurrent link failures and recoveries 61

6.3.2 Summary 63

6.4 Experiment 3: Effect of regrouping frequency (Limit_Pct variable) 65

6.4.1 Scenario 3.1: Regrouping frequency 66

6.4.2 Summary 69

6.5 Experiment 4: Simple/Intensive Tasks 69

6.5.1 Scenario 4.1: Simple vs intensive tasks 69

6.5.2 Summary 71

6.6 Experiment 5: Comparison of LBA1 and LBA2 ... 71
6.6.1 Scenario 5.1: LBA1 vs LBA2 ... 72
6.6.2 Summary ... 74
6.7 Conclusions ... 75
Chapter 7 ... 77
Epilogue ... 77
Bibliography ... 79

LIST OF TABLES

Table 1: Safety Properties of GCS Membership Service 8
Table 2: Safety Properties of GCS Multicast Service 9
Table 3: Indicators for Safe Messages 9
Table 4: Multicast Ordered and Reliable Services 11
Table 5: Liveness Properties 12
Table 6: Ensemble Routines [11] 34
Table 7: Ensemble Structures [11] 34
Table 8: Task Structure 36
Table 9: Task Types 37
Table 10: Task Allocation Type 38
Table 11: Message Tokens 38
Table 12: Message handling routines 40
Table 13: Arbitrary Regrouping mechanism elements 41
Table 14: Specific Regrouping mechanism elements 43
Table 15: Source files 46
Table 16: Script for gathering statistics 49

LIST OF FIGURES

Figure 1: Example of a Partitionable Network. 6
Figure 2: Interaction with a GCS [6] 13
Figure 3: Merge from two groups to one where m=1 15
Figure 4: Fragmentation from one group to three where f=3 15
Figure 5: Regrouping from four groups to three groups 16
Figure 6: Pseudocode of the algorithm 20
Figure 7: Allocating 7 incomplete tasks using LBA1 in a group of 5 processors 20
Figure 8: Allocating 7 incomplete tasks using LBA2 in a group of 5 processors 21
Figure 9: Comparison of LBA1 and LBA2 when two groups of 2 processors each merge 22
Figure 10: The sequence of events during adaptation [7] 25
Figure 11: A sample protocol stack [8] 27
Figure 12: Timeline of endpoints in a group, where A, B, C and D are endpoints [8] 28
Figure 13: Specific regrouping pattern – Singleton Groups Slowly Merge (SGSM) .. 42
Figure 14: Specific regrouping pattern - One Unstable Group (OUG) 43
Figure 15: Parameter file sample 44
Figure 16: Work - Effect of processor number 51
Figure 17: Message Complexity - Effect of processor number 55
Figure 18: Time – Effect of Processor Number 56
Figure 19: Work - Effect of task number 57
Figure 20: Message Complexity - Effect of task number 59
Figure 21: Time - Effect of task number 60
Figure 22: Work, MC - Effect of Group_Max 64
Figure 23: Time - Effect of Group_Max in SGSM 65
Figure 24: Work, MC - Effect of Limit_Pct 67
Figure 25: Time - Effect of Limit_Pct 68
Figure 26: Work, MC - Effect of Task Type in SGSM 70
Figure 27: Work, MC - Comparison of LBA1 & LBA2 with Intensive Tasks 73
Figure 28: Time - Comparison of LBA1 & LBA2 with Intensive Tasks 74

Chapter 1

Introduction

1.1 Motivation and Related Work

A lot of research has been conducted for studying cooperation in distributed systems. The abstract problem of performing a set of tasks by a set of distributed fault-prone processors is generally known as DO-ALL [24]. This problem has been considered in a variety of communication models, such as message-passing [5,24,9], shared-memory [2,14,19,20,22] and partitionable networks [12,10,8,21]. In the latter, DO-ALL is known as OMNI-DO [8]. It is set in a partitionable environment where links between asynchronous processors may fail and recover at any moment during the computation. In such a setting it is required that all processors know the results of all tasks, as opposed to the DO-ALL problem in which it is sufficient for each processor to learn that the tasks have been performed (but not necessarily know all task results).

Despite the active research on this subject, it remains theoretical. To the best of our knowledge, there are no empirical studies on proposed solutions and on their behaviour in realistic environments. The purpose of this thesis is to investigate recent research on this subject and implement and evaluate empirically an algorithm proposed for solving the OMNI-DO problem.

In the OMNI-DO problem, the concept of partitions is used. The underlying network structure is subject to dynamic changes that partition the communicating processors into groups. Processors in the same group can communicate reliably sharing information on the current computation. No processor can be in two groups at the same time and processors in different groups cannot communicate with each other. A partition transition occurs whenever the topology of the network changes, causing the processors to form new groups. These partition transitions are called *regroupings* and can occur multiple times during a

computation, creating what is called a *regrouping pattern*. Regroupings can be handled by a *group communication service* [23, 6].

The *efficiency* of the solutions for the OMNI-DO problem is measured in terms of *work* and *message complexity*. Work is defined as the total number of tasks executed by all processors during the computation. This includes redundant tasks that may be executed by multiple processors when the processors are disconnected due to link failures. Message complexity is the total number of point-to-point messages send by all processors during the computation.

In previous work, solutions have been provided for specific and arbitrary regrouping types. In [8], the OMNI-DO problem was introduced. The load balancing algorithm AF was proposed and analysed for partition changes triggered only by group *fragmentations*. Group fragmentations are restricted to forming groups of processors that were in the same group before the regrouping. Furthermore, it presents an effective scheduling strategy for minimizing the redundant tasks executed in partition changes triggered only by group *merges*. Group merges are restricted regroupings in which all processors of a group must join the same group after the regrouping.

The authors of [12] designed and analysed algorithm AX that solves the OMNI-DO problem efficiently under any pattern of fragmentations *and* merges. They propose the notion of *view graphs* to be used for studying distributed computing with group communication services and use it to analyse the complexity of their algorithm. View graphs are directed graphs that represent view changes at processors during executions. The authors in [12] present matching upper and lower bounds for work and message complexity of the algorithm, for any pattern of fragmentation and merges. Moreover, they improve the message complexity in relation to [8], for any pattern of *only* group fragmentations.

In [10], an algorithm is proposed that considers arbitrary regroupings beyond fragmentations and merges. It analyses and establishes bounds on the efficiency of a randomized algorithm, called RS. Algorithm RS employs random scheduling of incomplete tasks to solve the OMNI-DO problem. It compares the expected work of this algorithm to the

work of an optimal off-line algorithm (schedules tasks with full knowledge of the pattern of partitions) and concludes that this algorithm is work-optimal for any pattern of regroupings.

1.2 Contribution

To solve the OMNI-DO problem, recent research on the subject is studied and an algorithm is implemented that is in fact a combination of two previously proposed algorithms. The first one is algorithm AX [12]. This algorithm handles dissemination of knowledge between the members of a group and also provides a way to select the task that will be completed by each processor at any time. The group handling is done using a group communication service [23, 6]. A popular group communication service called Ensemble [28, 15] is studied and used in the implementation. The second algorithm is algorithm RS [10]. This algorithm provides a different load balancing rule that is ideal for disjoint groups that are merged at some point during the computation. Finally, part of the implementation of the new algorithm includes the implementation of a mechanism that simulates various types of regrouping patterns. Simulations are run on a single machine to conduct the experiments needed for empirical evaluation of the algorithm.

Experiments are conducted on specific regrouping patterns, additionally to the random regroupings, in order to evaluate the performance of the algorithm. Experiments are conducted with simple tasks (take little time to conclude) and computationally-intensive tasks in order to get a more clear idea on how the algorithm fairs in more realistic situations. Work and message complexity are measured in the experiments. We also measure the runtime of the algorithm, by averaging the time each processor needs to solve the problem.

It is important to note that the theoretical studies focus and assume that no processor failures occur but only link failures. The algorithms AX and RS were analyzed under this assumption. Our implementation is general enough to support processor failures by modelling a crashed processor as an isolated singleton group.

1.3 Document Structure

The remainder of this thesis is organised as follows. In Chapter 2 necessary background is provided. A description of Group Communication Services and their properties are given. Additionally, problems DO-ALL and OMNI-DO are defined. In Chapter 3 the algorithm that is implemented is described, as well as its relation to algorithms AX and RS. The properties that the GCS used for the algorithm must satisfy (in order to be applicable) are also mentioned. In Chapter 4 information for the Ensemble Group Communication Service is provided. The installation process is described in detail, the necessary requirements are specified and, any problems that were encountered and their solutions are mentioned. In Chapter 5 the specific implementation steps are given. The system that was built for the purposes of this thesis and running the simulations is specified. In Chapter 6 the experiments and simulations are specified and the results are analyzed. Final conclusions and future work are identified in Chapter 7.

Chapter 2

Background

In this chapter necessary background for this thesis is provided. We overview Group Communication Services and their properties and we specify DO-ALL and OMNI-DO problems.

2.1 Group Communication Services

In distributed computing it is essential that the participating processors can communicate and coordinate in executing common jobs. A way to do this is to use the notion of a group to represent the set of processors participating in the computation. Groups can be used to hide the complexity of the communication between the processors of the distributed system and improve its availability, efficiency and security.

Group Communication Services [6], or GCS for short, can provide such services for distributed applications. The processors that take part in a computation are organized into a group (or more depending on the requirements of the computation). They communicate via multicast messages, a capability provided by the GCS. This service guarantees a reliable and ordered dispatch of messages to all the members of the group. The multicast facility is one of the two main services provided by the GCS. The second important service is the Group Membership Service which is responsible for preserving a list of all the active processors in the group and making it available to each member. This list is called a *view* and the multicast messages are delivered to all the processors in the view.

The processors that participate may be prone to failures. They may crash, restart or lose the connection to the network. Furthermore, the network links are prone to failures which can result to the partitioning of the network. In Figure 1, an example of a partitionable network can be seen. The processors (represented by circles) of a distributed system can form groups (represented by squares) during a computation in such a setting.

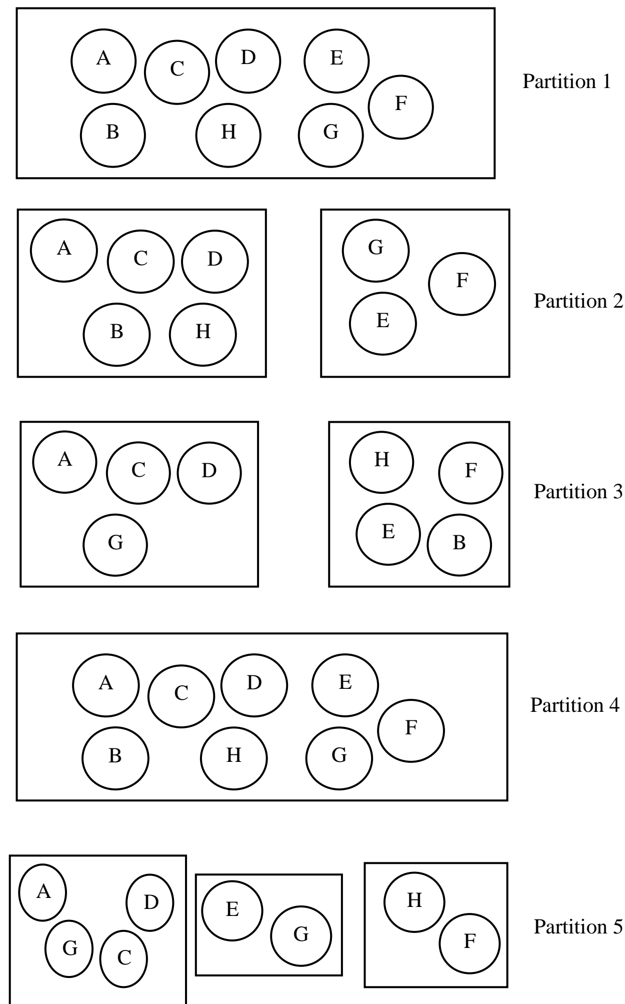


Figure 1: Example of a Partitionable Network.

At any time, the GCS attempts to realize the network situation and inform the members of the group using a "best effort" approach. There are various implementations of GCSs available for application developers. Each application developer should choose the GCS to use taking into account the guarantees it provides and the needs of the application.

There are many implementations of GCSs including Ensemble [15], Transis [7, 1], Isis [4], Horus [6], Totem [6] and Phonex [6]. The first two provide more functionality and hence are the most popular ones. Each GCS enforces a combination of properties and each application developer should choose to use the one that fits the distributed application better.

GCSs need to apply certain properties in order to be useful, including *safety* and *liveness* properties. A safety property is a property that ensures that "bad things" do not happen, e.g. the power plant will never blow up. A liveness property is a property that ensures that "good things" eventually happen, e.g. the power plant provides electricity. In the

following sections several properties of the GCSs will be presented, including safety and liveness properties.

2.1.1 Membership Service Safety Properties

When system failures happen, the view changes and the members of the group are informed with the new group situation by getting and installing the new view. The view also changes when new processors join the group or when members leave the group voluntarily. The group membership service of the GCS implements safety properties to assure that nothing goes wrong:

- **Self Inclusion** - The view that the processor knows to be the current view must include the processor itself.
- **Local Monotonicity** - Each view has an identifier and for each new view the identifier is monotonically increased.
- **Initial View Event** - When every processor starts it installs an initial view.

Most GCSs implement the above basic safety properties.

There are two types of group membership services in relation to the dependability of the underlying network. Primary component membership services assume that the network is dependable so all the processors in the system have the same view ordering, regardless if they are in the same group or not. To enforce this they implement the following optional property:

- **Primary Component Membership** – For every two consecutive views, a processor exists that is in both.

In a partitionable network each process group has its own view ordering. Most group communication services use partitionable membership service to cater for network failures. In Table 1, a summary of the safety properties that can be offered by the Membership Service of a GCS can be seen.

Membership Service Safety Properties
Self Inclusion (Basic)
Local Monotonicity (Basic)
Initial View Event (Basic)
Primary Component Membership (Optional)

Table 1: Safety Properties of GCS Membership Service

2.1.2 Multicast Service Safety Properties

The network that is used to deliver the messages between the processors is unreliable, meaning that messages can be lost. Additionally, there is no guarantee on the time it will take for a message to be delivered to its destination. The multicast service of the GCS implements the following basic safety properties:

- **Delivery Integrity** - Every message that is received by a processor was previously sent by another processor. This means that no message is generated on its own in the system.
- **No Duplication** - Each message is unique and is received only once by each receiving processor. The messages in the system are not duplicated.
- **Same View Delivery** - All processors that receive the same message receive it at the same view.

Some GCSs also implement the following optional safety properties:

- **Sending View Delivery** - The message is delivered at the processors that have installed the same view that the sender processor had when dispatching the message.
- **Virtual Synchrony** - When two processors have installed the same view and then they both change to another same view, the number of messages they received in their previous view is the same. Additionally, two sub-properties (Traditional Sets and Agreement on Successors) can be enforced to help indicate whether this property is applied.

In Table 2, a summary of the safety properties than can be provided by the Multicast Service of a GCS can be seen.

Multicast Service Safety Properties
Delivery Integrity (Basic)
No Duplication (Basic)
Same View Delivery (Basic)
Sending View Delivery (Optional)
Virtual Synchrony (Optional)
Traditional Set (Optional)
Agreement on Successors (Optional)

Table 2: Safety Properties of GCS Multicast Service

2.1.3 Properties for Safe Messages

In a distributed application the ideal functionality regarding message delivery is to either deliver the message to all the participating processors or to not deliver it to anyone. In that direction, some GCSs introduced the concept of Safe Messages to battle the unreliability of the underlying network to deliver messages. A Safe Message is a message that is delivered to the application only when the GCS knows with certainty that all the processors in the group received it. An extension to this is to deliver the message immediately to the application and wait for safe indications to arrive later.

- **Safe Indication Prefix** – A message is safe if it was received by all the processors in the view.
- **Safe Indication Reliable Prefix** – If a safe message is received by a processor and was sent by another processor in the view, then every previous message delivered to that processor is also received by all processors in the view.

In Table 3, a summary of the indicators for Safe Messages that can be provided by a GCS can be seen.

Safe Messages Properties
Safe Indication Prefix
Safe Indication Reliable Prefix

Table 3: Indicators for Safe Messages

2.1.4 Properties for Ordered and Reliable Services

The following ordering properties concern the order in which the messages are delivered to the group processors by the multicast service. The reliability properties are concerned with making sure that there are not any missing messages between the messages that are actually received in a view. Most group communication services provide one of the properties mentioned here:

- **FIFO Delivery** – Two messages sent by the same processor are received, in the same order they were sent, by the processors that receive both of them.
- **Reliable FIFO** – Two messages sent by the same processor in the same view are received, in the same order they were sent, by the processors that receive both of them.
- **Casual Delivery** – Two messages are received by the processors that receive both of them, in the same order they were sent. The difference from FIFO is that the messages are not necessarily sent by the same processor.
- **Reliable Casual** – Two messages sent in the same view are received, in the same order they were sent, by the processors that receive both of them.
- **Strong Total Order** – Messages are received in the same order by all the processors that receive them.
- **Weak Total Order** – Messages are received in the same order by all the processors that receive them if the processors remain connected. Connected means that the processors either stay in the same view forever or they move from the same previous view to the same new view.
- **Reliable Total Order or Atomic Order** – Two messages sent in the same view that have a timestamp denoting the time they were sent, they are received in the same order as indicated by their timestamp.

In Table 4, a summary of the Multicast Ordering and Reliability Properties than can be provided by a GCS can be seen.

Multicast Ordering and Reliability Properties
FIFO Delivery
Reliable FIFO
Strong Total Order
Weak Total Order
Reliable Total Order or Atomic Order
Casual Delivery
Reliable Casual

Table 4: Multicast Ordered and Reliable Services

2.1.5 Liveness Properties

By implementing safety properties the group communication services ensure that nothing goes wrong during a computation. However, this does not ensure that something useful is achieved from this computation. For example, if an application runs but does nothing and sends no messages, then the safety properties are enforced by default. For group communication services to be helpful in developing distributed applications, they need to be able to enforce liveness properties that ensure that something useful eventually happens.

To achieve liveness the group communication service attempts to be as correct and precise as possible when evaluating the situation of the network and processors in the group. In this context, the concepts of stable components and eventually perfect failure detectors are introduced. Stable component [6] is defined as a group of processors that ultimately become connected, meaning that they are active and have functioning network links with all the other processors in the group only. A failure detector [6] is a mechanism (external to the system) that attempts to detect if and when a part (processor or link) of the system failed. An eventually perfect failure detector is one that sooner or later reaches a point from which on it correctly detects the condition of each part of the system. The following basic liveness properties must be enforced for each stable component (if it exists), if an eventually perfect failure detector is available in the system.

Let *V* be the view that contains all the processors in a stable component:

- **Membership Precision** – For every processor in the stable component, the last view installed is V.
- **Multicast Liveness** – Every message that is sent by a processor, that is in a stable component and is sent in V, is received by all processors in the stable component.
- **Self Delivery** – A processor receives all messages that it sent in any view unless it the processor failed after sending the message.
- **Safe Indication Liveness** – Every message that is sent by a processor that is in a stable component and is sent in V, is indicated as safe by all the processors in the stable component. This property applies for group communication services that apply safe message properties as mentioned previously.

Some optional liveness properties require an eventually perfect failure detector but a stable component is not mandatory:

- **Membership Accuracy** – If at some point two processors are alive and the link connecting them is functional, then each processor installs a view that the other is included in, and in every view installed thereafter.
- **Termination of Delivery** – For every message sent in V, then each member of V either receives the message or the sending processor installs a new view.

In Table 5, a summary of the Liveness Properties that can be offered by a GCS can be seen.

GCS Liveness Properties
Membership Precision (Basic)
Multicast Liveness (Basic)
Self Delivery (Basic)
Safe Indication Liveness (Optional)
Membership Accuracy (Optional)
Termination of Delivery (Optional)

Table 5: Liveness Properties

2.1.6 Interaction primitives

The GCS needs to communicate with the application in each processor to be able to do its job. The following primitives are provided by the GCS to enable this interaction:

- **Send** – Send a multicast message to the processors of a group.
- **Receive** – Receive a multicast message from the group.
- **Unicast Send** – Send a unicast message to a processor in a group.
- **Unicast Receive** – Receive a unicast message from a processor in a group.
- **View Change** – Receive a notification about a view change.
- **Safe Prefix** – Receive safe prefix indications.

Events can occur that may cause a change in the network topology. This directly affects the group membership information kept by the GCS:

- **Crash** – A processor crashes or is disconnected due to a link failure.
- **Recover** - A processor recovers or a link is fixed.

In Figure 2, an example of the way a GCS interacts with an application is depicted. The application can send messages and receive messages from the GCS. The GCS sends view change notifications to the application to inform it of the current list of processors in the group. It can also send safe prefix indicators if the GCS uses Safe Messages. The external failure detector realizes any changes in the environment, such as processor or link failures and recoveries.

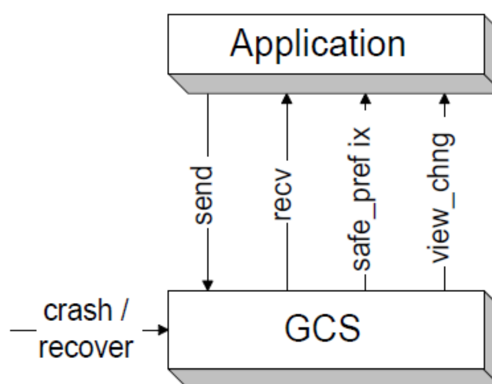


Figure 2: Interaction with a GCS [6]

In Chapter 4 we present in more detail the Ensemble GCS that was used in this work.

2.2 DO-ALL and OMNI-DO problems

DO-ALL is the problem of having a set of P asynchronous uniquely-identified processors cooperatively performing a set of N uniquely-identified tasks in the presence of failures [9]. The tasks are independent and idempotent, meaning that there is no particular order that the tasks must execute and the execution of any task is not dependent on the result of another. The tasks are known a priori to all the processors and the processors must cooperate to execute them all. At the end of the computation a processor does not necessarily need to know the results of all the tasks but it needs to know that all the tasks are completed. If a request for the result of a task is received from a client, the queried processor can obtain the result from another processor and be able to respond to the client. During the computation, any of the processors may fail either temporarily or permanently. They may lose connection to the distributed system, crash or just be too slow to respond in a manner that the system considers them disconnected. This problem has been studied in various failure models in both message-passing [13] and shared-memory [19] models, such as crash (the processor crashes and does not recover), crash/restart (the processor crashes but it may recover) and Byzantine (the processor exhibits malicious or arbitrary behavior).

A variation of the above problem is the OMNI-DO problem [8]. As in DO-ALL, the purpose of the distributed system is to complete all independent tasks with a set of connected processors which are prone to failures. OMNI-DO is studied in a partitionable network environment, meaning that link failures are also present (additional to processor failures). The processors may form various groups during the computation due to loss of their connection to (some of) the other processors. Later they may connect back to the initial group that started the computation or form other groups. Due to the existence of the partitionable environment, the processors must be omniscient (thus OMNI): they must know the results of all tasks before the computation completes. A processor may not be able to query the distributed system for a result at any time (since it may not be connected with some of the other processors). Hence, as soon as a processor is requested for the result of a task, it must know the result to be able to respond to the client.

A *regrouping* is defined as a transition from one network partition to another, which causes the formation of new groups of processors. Fragmentation and merges are specific patterns of regroupings.

Merge is formally defined as follows, where g is a group of processors:

$$Partition_V(g_1, g_2, \dots, g_k) \rightarrow Partition_{V+1}(g_x), such\ that\ g_x = \bigcup_{i=1}^{k} g_i$$

The formal definition of *fragmentation* is given as:

$$Partition_V(g_x) \rightarrow\ Partition_{V+1}(g_1, g_2, \dots, g_l), such\ that \bigcup_{i=1}^{l} g_i = g_x\ and\ g_i \cap\ g_j\ =\ \emptyset$$

Given a computation that undergoes a series of regroupings, we define f to be the fragmentation number, that is, the number of new groups created due to fragmentations. Similarly, we define m to be the merge number, meaning the number of new groups created due to merges. In Figure 3, a group merge with m=1 is depicted.

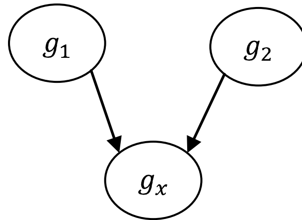


Figure 3: Merge from two groups to one where m=1

In Figure 4, a group fragmentation with f=3 is depicted.

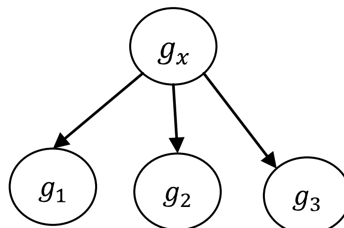


Figure 4: Fragmentation from one group to three where f=3

In Figure 5, a general type of regrouping is depicted. As can be observed, this is neither a merge nor a fragmentation. It is not a merge since, for example, in g_y processors

from both g_2 and g_4 join the group but processors from those groups also join other groups. It is not a fragmentation, since g_2 fragments to g_x and g_k, but members from other groups also join these two groups.

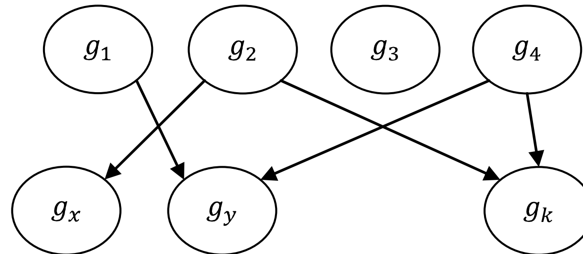


Figure 5: Regrouping from four groups to three groups

An off-line task scheduling algorithm is considered to be one that knows the pattern of regroupings beforehand and can optimally schedule tasks to minimize work. An on-line algorithm does not have this knowledge and must schedule tasks by realizing the situation on the go. When an on-line algorithm is analyzed for *competitiveness* [26, 3], its performance is compared to the performance of an optimal off-line algorithm. An algorithm is said to be *a-competitive*, if its competitive ratio [26] (the ratio between its performance and the offline algorithm's performance) is less or equal than *a*.

In [8], the first study of the OMNI-DO problem was conducted were regrouping patterns were limited to only group merges or only group fragmentations. It introduces a lower bound on the worst case competitive ratio of the termination time of an on-line algorithm relative to an off-line algorithm. Based on that conclusion, they propose algorithm AF that guarantees completion with total work $O(N \cdot f + N)$ for any pattern of fragmentations. Algorithm AF uses an abstract GCS to handle inner-group communication and coordination. Additionally, the authors in [8] present a scheduling strategy for minimizing the task execution redundancy, for any pattern of merges. This strategy can schedule $\Theta(N^{1/3})$ tasks with at most one task execution overlap for any two processors.

In [12, 13], algorithm AX was introduced that uses a group communication service and a rank-based load balancing rule. Upper bounds on work and message complexity are shown for this algorithm in respect to a system of processors that start in a single group and

regrouping patterns that are limited to merges and fragmentations. Specifically, it was proved that:

1. For any pattern of fragmentations and merges
   - Work is at most $\min\{N \cdot f + N, N \cdot P\}$
   - Message complexity is at most $4(N \cdot f + N + P \cdot m)$
2. For any pattern of only fragmentations
   - Work is $O(\min\{N \cdot f + N, N \cdot P\})$
   - Message complexity is $O(N \cdot f + N)$

For any pattern of fragmentations and merges, Algorithm AX is rendered to be work-optimal since its upper bound matches the lower bound of OMNI-DO, which was shown to be $\Omega(\min\{N . f, N . P\})$.

In [10, 13], the task scheduling algorithm RS is introduced that handles arbitrary regrouping patterns and it uses a random allocation strategy based on permutations of tasks. In that work, the notion of *computation width* (cw) is defined as the maximum number of independent groups that can exist concurrently in a partition (e.g., the regrouping pattern in Figure 5 has cw equal to four). Matching lower and upper bounds are established on the competitive ratio of the algorithm's work for all computation patterns with a given computation width. Consequently, RS achieves optimal competitive ratio. Specifically:

- (Upper bound) For any computation pattern C, the randomized algorithm RS is (1 + cw(C)/e)-work competitive, where e is the base of the natural logarithm and cw(C) the computation width of C.
- (Lower bound) For any scheduling algorithm A that is designed to solve the OMNI-DO problem with P processors and N tasks, there exists a computation pattern C with cw(C)=k such that algorithm A is at least (1 + k/e)-work-competitive.

Further details on algorithms AX and RS are given in the next chapter.

Chapter 3

The OMNI-DO Algorithm

The algorithm described here is concerned with solving the OMNI-DO problem as is described in the previous chapter and is a hybrid of the two algorithms mentioned previously, AX and RS. Inner-group task scheduling is handled by algorithm AX while inter-group task scheduling is handled by algorithm RS. This is explained further in the following sections.

3.1 General Algorithm Description

The core of the algorithm is mostly based on algorithm AX. It uses a GCS to handle coordination of processor activity in a network that suffers regroupings due to failures of communication links. At any time during the computation each processor is included in only one group (unless it failed). The initial regrouping is considered to be the initial state of the system, in which all the processors are connected and take part in the computation in a single group. Each processor that participates knows all the initial tasks that need to be executed to finish the computation. The algorithm is executed in ordered rounds, always starting from number 1 each time a new group is formed.

During the first round, the GCS notifies the processors with a list (sorted in ascending order) of the processors in the newly formed group. Based on that list, the processors are assigned an identifier, called rank. Within the group, the processor with the highest rank is decided to be the coordinator of the participating processors. Since all the processors in the group know the ranked list of the group processors, they can easily know the coordinator without further communication. The coordinator is responsible for handling task allocation and dissemination of information regarding the completed tasks. The allocation of tasks to each processing unit is done using a load balancing rule. After that, the following steps are followed for each round including the first one:

1. Each processor sends the task results it knows to the coordinator, via unicast.

2. The coordinator receives results from all the processors it knows are in the group. It updates the tasks with the results and then sends all the results it now knows to all the processors in the group, using multicast.
3. Each processor updates its completed tasks and results variables and uses the load balancing rule to start computing a task (or not) from the list of the unfinished tasks.
4. Round changes. Continue to 1.

At the beginning of each round it knows which tasks are already completed their results and which processors are currently connected in this group. The algorithm executes within each group until each of the participating processors, regardless of the group they are in, knows all the results (provided the processor has not failed, otherwise it is ignored). As soon as a processor knows the result of all the tasks it remains idle, until the next regrouping. If requests arrive for task results, the processors have the results locally stored and can send them immediately. In Figure 6, the pseudocode of the algorithm can be seen.

3.2 Task Allocation Methods

The load balancing rule proposed by algorithm AX is very simple and will be referred from now on as Load Balancing Allocation 1 (LBA1). The processors are sorted in ascending order within the group they are in and are given a rank, as mentioned previously. Each task is given a rank and is sorted in ascending order. Each processor, regardless of the group it is in, has the same task sequence. With this rule, each processor gets assigned a task depending on its own rank as follows:

- If the rank of processor x is less or equal to the number of tasks, the processor is assigned the task that has the same rank.
- If the rank of processor x is greater than the number of tasks, the processor does nothing for that round.

```
At each client C:
Initialize CompletedTasks, KnownResults, Processors, Phase=Send, Round=1
Permutation(C) Tasks   //for LBA2 Task allocation
WHILE Phase!=Sleep
   IF newview(members) THEN
      Round=1, Phase=Send, Coordinator=Max(Rank), Group=members
   ENDIF
  IF Phase=Send THEN
      IF Coordinator THEN
         Phase = Receive
         MsgReceived=0
      ELSE
         SEND(Coordinator) CompletedTasks
         Phase = MReceive
      ENDIF
  ENDIF
  IF Phase = Receive AND RECEIVE(Group) CompletedTasks THEN
     MsgReceived= MsgReceived+1
    IF MsgReceived=members THEN
       Phase=MCast
    ENDIF
   ENDIF
  IF Phase=Mcast THEN
      CAST(Group) CompletedTasks
      Phase=MReceive
   ENDIF
   IF Phase=MReceive THEN
        Update CompletedTasks, KnownResults
        IF CompletedTasks=Tasks THEN
            Phase=SLEEP
        ENDIF
   ELSE
         EXECUTE Task(Rank(C))
         Update CompletedTasks, KnownResults
         Round=Round+1
         Phase=Send
   ENDIF
ENDWHILE
```

Figure 6: Pseudocode of the algorithm

This rule handles inner-group task scheduling. An example of LBA1 in action can be seen in Figure 7, were R represents Rank.

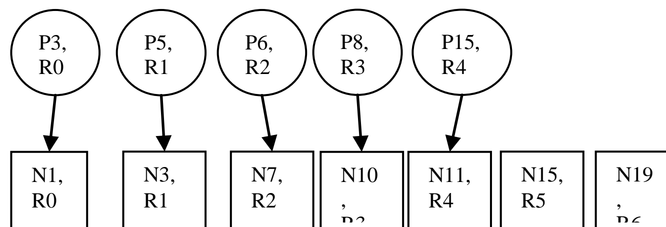


Figure 7: Allocating 7 incomplete tasks using LBA1 in a group of 5 processors

In the second algorithm RS, the load balancing rule proposed, referred to as Load Balancing Allocation 2 (LBA2), employees a random allocation method. Each processor in the distributed system has a different permutation (ordered arrangement) of the set of tasks that need to be executed. Each group knows which tasks are already completed. Given that knowledge, each group next executes incomplete tasks in the sequence of tasks that the coordinator of the group has. This rule is responsible for allocating different tasks between groups, reducing execution of redundant tasks. The resulting effect of this inter-group task scheduling algorithm can be seen in Figure 8, if LBA1 is used for inner-group scheduling (each processor gets assigned a task based on its rank). R represents Rank.

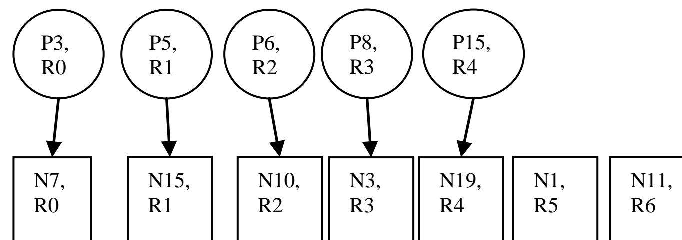


Figure 8: Allocating 7 incomplete tasks using LBA2 in a group of 5 processors

For both LBA rules, the task given is taken from the set of incomplete tasks and no two processors get assigned the same task within the same group.

With LBA1, if two groups exist, both groups will get assigned the tasks in the same order (since they have the same task sequence), meaning both processors of rank 0 will get assigned the task with rank 0 in the first round etc. In case those groups merge, at some point during the computation, no new knowledge will be earned since both groups allocated the tasks in the same order. Using LBA2, it is possible to gain additional knowledge, in comparison to LBA1, in case the two groups merge since the order of assigned tasks is different. Consider the situation were an initial group is fragmented to two groups. It is unlikely that both processors with rank 0 of the two groups will get assigned the same task, since both get a task in a different permutation from the set of incomplete tasks. If the two groups merge, before the computation is completed, it is highly possible that each group will

have a different set of completed tasks and thus reducing the amount of redundant tasks performed and the set of incomplete tasks of the new group.

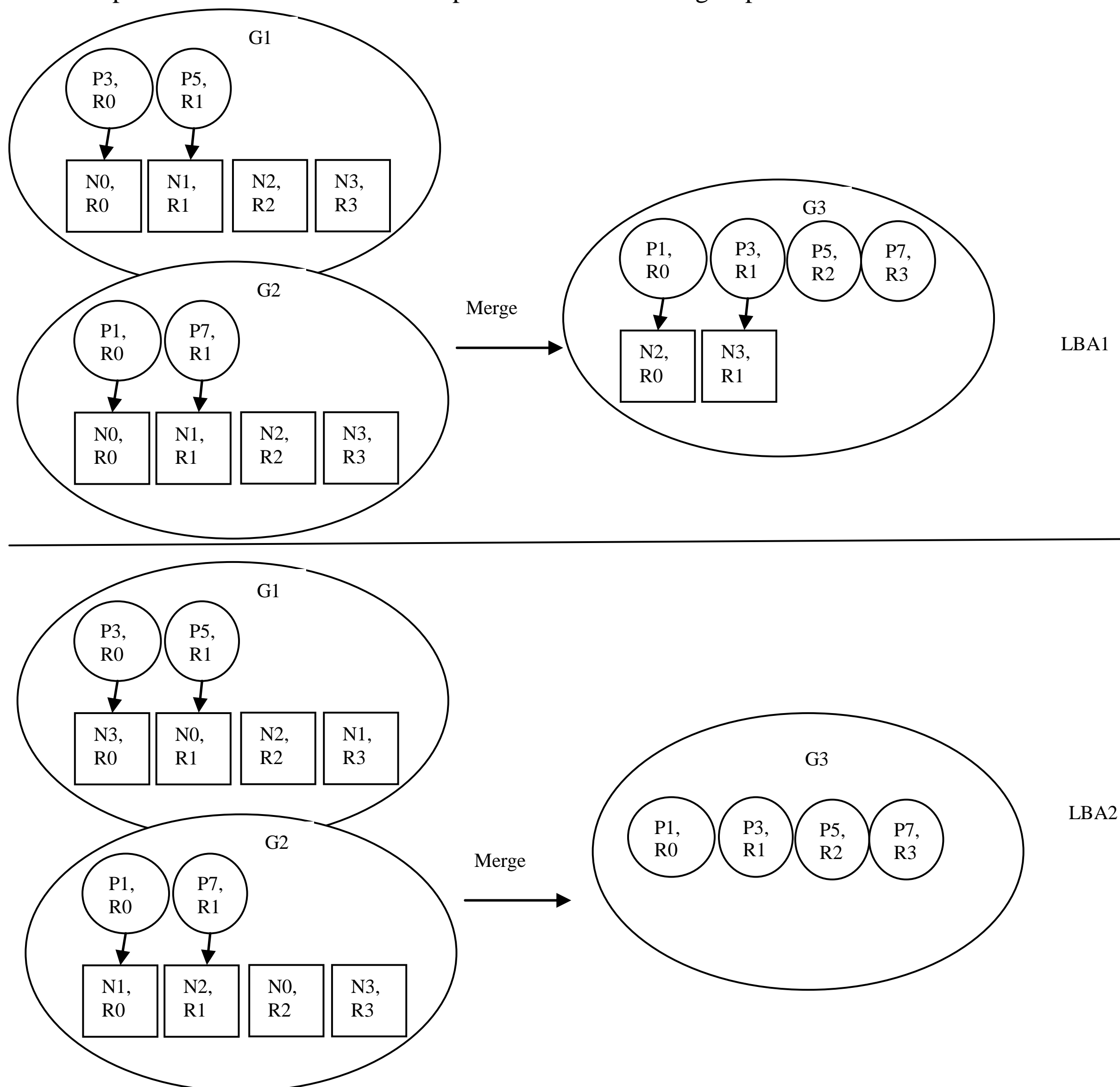


Figure 9: Comparison of LBA1 and LBA2 when two groups of 2 processors each merge

In the example of Figure 9, work done is less using LBA2, since the work is 4 while with LBA1 the work is 6. During the experimentation phase the two task allocation methods are compared so to empirically validate these observations.

3.3 Purpose of GCS

The algorithm we implemented depends heavily on using a GCS. The GCS needs to satisfy some basic properties, that will enable it to handle the group memberships and the communication between the processors used during the algorithm execution. The following

safety properties need to be satisfied during any run of the algorithm (see their definitions in Chapter 2):

- Self Inclusion
- Local Monotonicity
- Initial View Event
- Delivery Integrity
- No Duplication
- Same View Delivery

Furthermore, the following liveness properties need to be satisfied:

- Termination of Delivery
- Membership Accuracy

The following actions need to be provided by the GCS to be able to interact with the processors that are running the algorithm:

- View Change
- Send
- Receive
- Unicast Send
- Unicast Receive

Both Transis [7, 6] and Ensemble [15] GCSs satisfy all of the above requirements. We initially chose Transis for our algorithm implementation. An attempt to install the Transis system failed due to technical difficulties and incompatibilities with newer versions of the libraries and compilers needed. Furthermore, it seems that it does not have active support any longer that could help us overcome these issues (we attempted to contact the vendors unsuccessfully). Therefore, Ensemble was used, a GCS that is implemented as a client-server system. For the purposes of this thesis, an Ensemble Client written in C language was implemented that communicates with the server to use the services it provides. The client is in essence the application that uses the group membership and multicast services to

communicate with other identical clients in order to solve the OMNI-DO problem. Details on this particular GCS are given in the following chapter.

Chapter 4

The Ensemble Group Communication Service

Information about the architecture and characteristics of the Ensemble Group Communication Service is given in this chapter. Furthermore, the installation procedure followed for this implementation is described. We also list the problems we encountered during this procedure and the actions we took to overcome them.

4.1 Description

The Ensemble system aims in aiding the implementation of reliable distributed systems. In distributed computing, it is common for applications to be executed while the underlying network structure is subjected to dynamic changes. It is essential that the applications are adaptive and flexible to continue working properly under these variable conditions.

When a distributed application begins to execute, it takes into account certain conditions of the underlying distributed system. The application bases its execution on these conditions. When these conditions change, the distributed system needs to modify its configuration and provide the necessary information to the application in order to continue working properly and effectively. In Figure 10, the adaptation process can be seen.

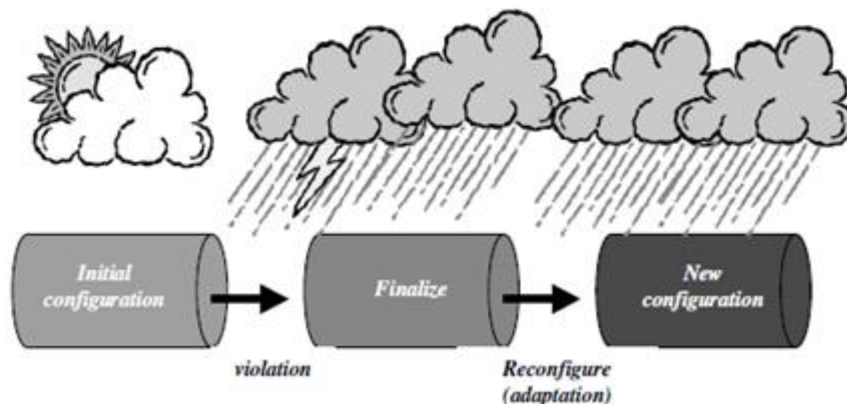


Figure 10: The sequence of events during adaptation [7]

Examples of adaptive behavior for which the Ensemble system can provide solutions includes group membership maintenance, replication of data or transfer of execution of an application between secured and unsecured environments. In this thesis we are interested in Ensemble as a group communication service, since this is what our algorithm implementation needs: an effective GCS.

4.2 Architecture

The Ensemble architecture is based on the use of layers (also referred to as micro-protocols) to dynamically compose protocol stacks. The application can select which layers (that correspond to properties the system will provide) it needs and create a protocol stack to use. Examples of layers include failure detectors, flow control, group membership, security, send and receive messages etc. Furthermore, the application does not reside on the top of the protocol stack (as in the internet protocol stack) but is in fact represented by another layer. Each layer communicates with the neighboring layers using events and so the communication goes from the top layer to the bottom and vice versa. The protocol stack is replaced by another one with every reconfiguration that happens when the system environment changes. The new configuration is applied by sending same View State records to all the participating processors. In Figure 11, the protocol stack created when using the default Ensemble properties can be seen.

Processors are of course important since they execute applications and communicate between one another. They communicate by sending and receiving messages via the network. The network is unreliable on its own but the Ensemble system has the ability to enhance the communication and provide reliability and security. Since this is a group communication service, the concept of groups is pretty important. Communication groups have unique names that serve as identifiers for each group. Processors can create one or more groups that contain one or more endpoints which have unique identifiers too.

protocol	description
Top	top-most protocol layer
Heal	partition healing
Switch	protocol arbitration and switching
Migrate	process migration
Leave	reliable leave
Inter	multi-partition view changes
Intra	single partition view changes
Elect	leader election
Merge	reliable merge protocol
Slander	failure suspicion sharing
Sync	view change synchronization
Suspect	failure detector
Stable	broadcast stability detection
Appl	application representative
Frag	fragmentation/reassembly
Pt2ptw	point-to-point window flow control
Mflow	multicast flow control
Pt2pt	reliable, FIFO point-to-point
Mnak	multicast NAK protocol
Bottom	bottom-most protocol layer

Figure 11: A sample protocol stack [8]

In Figure 12, an example of endpoints changing groups due to failures or merges can be seen.

4.3 Characteristics

The Ensemble GCS is built as a Client-Server system. The server provides the group communication services. The Client must connect to the server to be able to use the group communication services via message passing. The application is in essence the client and is responsible for allocating and freeing any necessary memory to use with the actions provided by the Ensemble server. The application is informed by the server if any new messages exist in order to receive them.

When developing Ensemble a great effort was put into providing platform independence. For this reason clients can be developed in various programming languages such as ML, C, C++ and Java. Our algorithm was implemented using the C programming language. A detailed description of implementing an Ensemble Client that uses the provided group communication services will be given in another chapter.

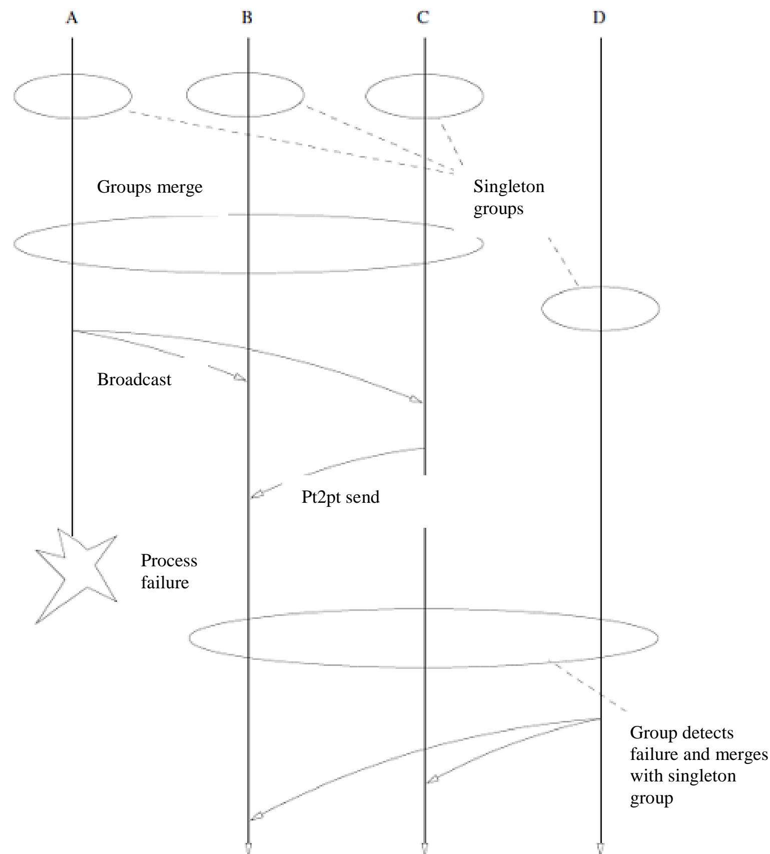


Figure 12: Timeline of endpoints in a group, where A, B, C and D are endpoints [8]

4.4 Installation & Compilation Instructions

According to the Ensemble installation instructions, it can be installed on Unix-like as well as on Win32 platforms. The operating system chosen for this installation was Ubuntu 8.04. The following packages were necessary and were installed during the procedure of installing Ensemble and implementing the algorithm:

- Essential packages (build-essential) – This includes a version of GNU-make and the C compiler. They were necessary for compiling Ensemble and were also used during the implementation of the algorithm.
- Packages tlc8.4 & tk8.4 – necessary to compile Ensemble.
- Emacs – used during implementation of the algorithm.
- C shell (csh) – used during implementation of the algorithm.
- Ocaml 3.08 or newer version – necessary to compile Ensemble.
- Java – necessary to compile Ensemble.

The Ensemble project has been in development since 1991 although it was a different version then called Horus/C. It has been completely re-implemented since then but even so, the versions the tools used for it have become obsolete (since its second implementation). One example is the C compiler used and some modifications that were needed to be done in some files in order to be able to compile this distribution. Another example is the version of GNU-make which caused a necessity to modify the make files. Additionally, some configuration changes needed to be made to inform Ensemble of the platform that was going to be used. The following exact procedure was followed for compilation of Ensemble (this is a combination of instructions found in the Ensemble official Installation document and modifications made during the installation used for this thesis):

1. Download the Ensemble distribution from http://dsl.cs.technion.ac.il/projects/Ensemble//ftp.html. Unzip it. The folder named "ens2_01" is created that contains the sources. This will be called the root folder of Ensemble.

2. Edit shell file – In user root, edit the ".bashrc" file using Emacs. Add the following lines (in order to setup some environment variables) in the file and save:

```
export PATH = $PATH:.:/usr/local/bin:
export OCAMLLIB = /usr/local/lib/ocaml
export ENS_CONFIG_FILE=$HOME/ensemble.conf
export JAVA_HOME = .... (to run the java client if desirable)
```

3. From the root of Ensemble open the file "mk/config.mk". This is the Ensemble configuration file. At line 120 (the Ensemble configuration section) make sure that "HSYS_TYPE=unix". This means that Ensemble will use the Unix library since Ubuntu is a unix based system (The alternative is to use the socket library supported for win32 systems). At lines 163 and 167 (configuration macros section), change "MAKE=gmake" to "MAKE=make".

4. From the root of Ensemble open the file "server/socket/s/mm.c". At line 27, replace

```
mm_Cbuf_val(cbuf_v) = NULL;
```

with

```
Field(cbuf_v,0) = NULL;
```

5. From the root folder of Ensemble run the following commands:

   ```
   configure
   ```

   This will compute the system settings and write them in file "mk/env.mk" in the format "machine type"-"operating system type". For this installation "i386-linux" was written.

   ```
   make depend > depend.txt 2>&1
   ```

   This will make the Ensemble system dependencies and redirect the output in a file named "depend.txt" located in the Ensemble root folder.

   ```
   make all > log.txt 2>&1
   ```

   This will make the Ensemble system and redirect the output in a file named "log.txt" located in the Ensemble root folder. Besides the Ensemble daemon (server), this will also compile the sample C client.

   ```
   make tests > tests.txt 2>&1
   ```

   This command should be run if compilation of the additional tests is desirable. It will redirect the output in a file named "tests.txt" located in the Ensemble root folder.

   ```
   make clean
   ```

   This will leave only the binaries and libraries and remove any other files created during the compilation. These are located in "Ensemble root/bin/ i386-linux" and "Ensemble root/lib/ i386-linux" respectively.

6. Additionally, to built the java client run the following commands:

   ```
   cd client/java
   make all > javalog.txt 2>&1
   ```

7. Create a file named "ensemble.conf" and put it in $HOME (user root). This is the configuration file that will be used by the Ensemble server and client in order to be able to communicate. This will be explained in further detail later and when appropriate.

The Ensemble distribution came with some sample application clients in languages supported by Ensemble specifically in C, Java and C-sharp. For this thesis the C client was used as the starting point of the algorithm implementation. The source file of the sample client is located under Ensemble root/Client/c and the main source file is "c_mtalk.c". The client implements the basic functions of connecting to the server, forming groups, keeping

membership information and sending multicast messages. At this point the configuration file ("ensemble.conf") mentioned above will need to be created in order to test the client.

The mandatory lines that must be included are:

```
# The set of communication transports.
ENS_MODES=UDP

# The user-id
ENS_ID=ioanna

# The port number used by the system
ENS_PORT=6789
```

There are other configuration parameters to include that are not necessary for now. To run the Ensemble server and client(s):

- To run the Ensemble server one must go to "Ensemble root/bin/i386-linux" and run the command "ensembled". This will start the ensemble-daemon which is responsible for providing the group communication services.
- To run the client one must go to "Ensemble root/bin/i386-linux" and run the command "c_mtalk". The client must be executed as many times as necessary and it can be seen that they all merge in the same group. Typing a message and pressing "Enter" will cause all the members of the group to receive it.
- To be able to run the server/clients as soon as the terminal is opened, the path of the binaries must be added in PATH variable (in shell file), similarly to the other variables in point 2 of the installation instructions above.

4.5 Local processors

Ensemble is a general-purpose communication system indented for constructing reliable distributed applications and as such it is able to support processors connected via LAN or WAN. We initially planned to run our experiments on processors in different machines connected via LAN. There are some guidelines in [16, 17] for configuring the application to locate the membership service (by adding configuration parameters in file

“ensemble.conf”). However, parts of the documentation seem to be obsolete and incomplete (at some locations it even mentions so). After various unsuccessful attempts to run our application on different machines we attempted to get support from the creators of Ensemble. We did not manage to receive adequate support so we decided to run our experiments locally (on a single machine). This does not negate the results of our empirical evaluation. On the contrary, it is best to conduct simulations first in order to realize the important parameters that affect the algorithm, so they can be consequently used in a real deployment.

Chapter 5

Algorithm Implementation

In this chapter a description of the C Language API offered by the Ensemble system is described, along with the decisions and steps taken during the implementation of the algorithm. The system that was built for the purposes of this thesis and running the simulations is specified.

5.1 C Language API

The Ensemble distribution provides a Tutorial [17] and also a sample Client that can be used by developers as a starting point in developing their own Ensemble Clients. In this implementation the Client was written in C language so we will focus in C Language Client API only. Additional information about Server side implementation and the Java Client API is included in the Ensemble Tutorial. To be able to use the C Client API it is necessary to include the necessary libraries by including the following header files in the code:

```
#include "ens.h"
#include "ens_threads.h"
#include "ens_utils.h"
#include "ens_comm.h"
```

The important interaction routines provided by the API are summarized in Table 6 and the structures in Table 7.

Calling routine ens_Init is the first step in connecting and communicating with the Ensemble server. Routine ens_poll is necessary to inform of any pending messages on the server and needs to be called in regular intervals. Routines ens_Join and ens_Leave are called by the clients when they want to join or leave a group. Routines ens_Cast, ens_Send and ens_Send1 are used to communicate messages between the members of the group. Specifically, ens_Cast is used to multicast messages to the members of the group, ens_Send is used to send point-to-point messages to all the members of the group and ens_Send1 is used to send a point-to-point message to one member in the group.

Interaction routines
ens_conn_t *ens_Init(void);
ens_rc_t ens_Poll(ens_conn_t *conn, int milliseconds, /*OUT*/ int *data_available);
ens_rc_t ens_Join(ens_conn_t *conn,ens_member_t *memb,ens_jops_t *ops,void *user_ctx) ;
ens_rc_t ens_Leave(ens_member_t *memb) ;
ens_rc_t ens_Cast(ens_member_t *memb, int len, char *buf) ;
ens_rc_t ens_Send(ens_member_t *memb, int num_dests, int *dests, int len, char* buf) ;
ens_rc_t ens_Send1(ens_member_t *memb, int dest, int len, char* buf) ;
ens_rc_t ens_BlockOk(ens_member_t *memb) ;
ens_rc_t ens_RecvMetaData(ens_conn_t *conn, ens_msg_t *msg);
ens_rc_t ens_RecvView(ens_conn_t *conn, ens_member_t *memb, /*OUT*/ ens_view_t *view);
ens_rc_t ens_RecvMsg(ens_conn_t *conn, /*OUT*/ int *origin, char *buf);

Table 6: Ensemble Routines [11]

C Data Structures
typedef enum ens_rc_t ENS_OK = 0, ENS_ERROR = 1 ens_rc_t;
typedef struct ens_jops_t char group_name[ENS_GROUP_NAME_MAX_SIZE] ; /* The group name */ char properties[ENS_PROPERTIES_MAX_SIZE] ; /* The set of properties */ char params[ENS_PARAMS_MAX_SIZE] ; /* The set of parameters */ char princ[ENS_PRINCIPAL_MAX_SIZE] ; /* My principal name (security) */ int secure ; /* Do we want a secure stack (encryption + authentication)? ens_jops_t ;
typedef enum ens_up_msg_t VIEW = 1, /* A new view has arrived from the server. */ CAST = 2, /* A multicast message */ SEND = 3, /* A point-to-point message */ BLOCK = 4, /* A block requeset, prior to the installation of a new view */ EXIT = 5 /* A final notification that the member is no longer valid */ ens_up_msg_t;
typedef struct ens_msg_t ens_member_t *memb; /* endpoint this message blongs to */ ens_up_msg_t mtype ; /* message type */ union struct /* The variant for VIEW: */ int nmembers; /* the number of members in a view */ view; struct /* The variant for a point-to-point message */ int msg_size; /* length of a bulk-data */ send; struct /* The variant for multicast message */ int msg_size; /* length of a bulk-data */ cast; u; ens_msg_t;

Table 7: Ensemble Structures [11]

The routine BlockOk is used to inform the system that no messages will be sent in this view by this member from that point on. It is used when a block command message is received i.e. when another member joins or leaves the group. The routine RecvMetaData is used after the client knows for sure that pending messages wait to be received from the server i.e. after a positive response from ens_poll. It brings to the application information about the message but not the message itself, i.e. type and size of the message etc. Once the information regarding the messages is received, it is time to call one of RecvView or RecvMsg depending on the information returned by RecvMetaData. RecvView is called when a view change event must occur while RecvMsg is called to actually receive a message.

When the ensemble system is compiled a sample C Client is compiled and ready to be used as well. It is a simple program implementing a multi-person talk and it is a good starting point for implementing a client that connects to the Ensemble server and uses the group communication services.

5.2 Tasks

The application needs to handle information regarding the completed and pending tasks. It maintains an array TASK *Tasks_Array and for each task it keeps information regarding its status and its result if it is known. It also maintains an array TASK *Inc_Tasks_Array which includes the incomplete tasks and is only created every time a new group is formed. It also maintains a counter Local_Tasks_Completed that has the number of tasks the processor knows are completed at any time. Another counter TasksDoneLocally is maintained, that has the number of tasks the processor itself has executed. The lstatus of all tasks is initially marked as INCOMPLETE and when the processor executes it or receives news from other members of the group that it was executed, it marks it as COMPLETE. The variable tid in the Task structure is used to identify each task between Tasks_Array and Inc_Tasks_Array, meaning that tid for a task in Tasks_Array will be equal to the index of the corresponding task in Inc_Tasks_Array (if it is incomplete) and vice versa. In Table 8, the Task structure can be seen.

Important Structure

typedef struct Task_Info{
int lstatus; //local status
int result;
int tid;
}TASK;

Table 8: Task Structure

There are two types of tasks that the processors can complete. The first type is simple tasks that their processing essentially constitutes of a simple assignment that the task is completed. To complete this type of task the routine void Complete_Task(int task_pos) needs to be called with the parameter being the position of the task in the array of tasks.

The second type of tasks is the computationally-intensive tasks and these will be used to extract useful statistics for realistic situations. It does not really matter what the results of these tasks are. What matters is that they take some time to compute, this way creating realistic conditions for the simulations. For this purpose the task chosen is the computation of π digits. The number of digits will determine the complexity of the task and the time it will take to be executed. To calculate π digits first we need an array the size of the number of π digits we want to calculate. Then we call routine void arctan(int multiplier, int denom, int sign) twice with the following parameters:

```
arctan(16, 5, 1);
arctan(4, 239, -1);
```

The implementation of π digits calculation is based on code found in [28].

All the tasks will essentially have the same result but as mentioned above we only care that each task will take some time to be executed. To complete this type of task the routine void Complete_Realistic_Task(int task_pos) needs to be called with the parameter being the position of the task in the array. In Table 9, the routines that need to be called for executing simple and computationally-intensive tasks can be seen.

Task Types	
Simple	void Complete_Task(int task_pos)
Computationally - Intensive	void Complete_Realistic_Task(int task_pos)

Table 9: Task Types

5.3 Task Allocation

As mentioned previously, the algorithm proposed that solves the OMNI-DO problem is a combination of two algorithms, AX and RS. Both algorithms offer a different way of allocating tasks to the coordinating processors, LBA1 and LBA2. LBA1 uses rank to determine which task is assigned to which processor. Specifically, since both tasks and processors are ranked, each processor gets assigned the task that corresponds to its rank. If more tasks remain they will be assigned in a following round, based again on their rank order. To invoke this type of assignment the routine int Assigned_Task(int rank, int size) must be called which takes as parameters the rank of the processor that is about to get assigned a task and the size of the tasks array.

LBA2 assigns a random task from the set of INCOMPLETE tasks. This random allocation needs to apply the rule that no two processors within the same group get assigned the same task. In order to achieve this, the RS selection algorithm runs at the beginning of the execution and assigns each processor with a different permutation of the initial tasks. This ensures that any processor in the same group gets assigned a unique task and random selection between tasks still applies. At the beginning of round 1, the coordinator sends the multicast message with the completed tasks in the group and includes its permutation of the tasks (assigned to him at the beginning of the execution). The processors in this group get assigned tasks based on this permutation. In essence, the actual selection of the tasks is identical to the first load balancing rule and achieved by calling the same routine int Assigned_Task(int rank, int size).

The routines that are used to create the permutation are void swap_tasks(TASK *a, TASK *b), were parameters are the two tasks to be swapped and void shuffle(TASK *p,int

size), were the parameters are the array to be randomized and its size. In Table 10, the routines used for assigning tasks and creating permutations of the initial task sequence can be seen.

Task Allocation routines
int Assigned_Task(int rank, int size)
void shuffle(TASK *,int)
void swap_tasks(TASK *a, TASK *b)

Table 10: Task Allocation Type

5.4 Messages

Communication between the processors, for keeping track of those that are alive and connected during fragmentations and merges of groups, is handled by the group communication service. During the execution of the algorithm though, certain messages need to be exchanged that are part of them. Table 11 lists the identifiers for parts of the messages.

Identifier	**Actual String**	**Explanation**
REPORT	REP	Beginning of an algorithm report message - For validation purposes
SEPARATOR	$	Separator between each useful data in the message
DONE	DON	End of first part of the message
RESULT	RES	Result of a task
END	END	End of an algorithm report message - For validation purposes
RND	Number in string form	Number of current round
NUM	Number in string form	Number of tasks
STRING-SEP	String of numbers between SEP	Numbers alternating with SEPARATOR
PERMUTATION	PER	Beginning of part of the message that gives the order of the tasks

Table 11: Message Tokens

The first type of message is the message a processor sends to the leader of the grouping as soon as it completes its currently appointed task. This message is unicast to the leader by each processor in the beginning of each round. The second type of message is the one the leader multicasts to all the members in the group (to which it is leader). Both messages have the following format (without the spaces):

```
REPORT SEP RND SEP NUM SEP DONE SEP STRING-SEP RESULT SEP
STRING-SEP END
```

The SEP is a character used to easily encode and decode the useful parts of each message. The first part of the message is from REPORT until DONE. It contains the number of the current round for which the processor sends this message (RND) and the number of tasks (NUM) that the processor knows are completed and sends information for in this message. The second part of the message starts after DONE and finishes at END. It contains information for each completed task known by the processor sending the message. The first STRING-SEP after DONE contains the numbers of the tasks that it knows are completed. The second STRING-SEP after RESULT contains the result of each task in the position of the corresponding task.

To create this message, routines Create_Completed_And_Results_String() (creates and returns the second part of the message) and Create_Report_Msg() (creates the first part and concatenates it with the second part) need to be called. Routine Collate_Report() decodes the message that is received from the leader and Update_Completed_And_Results() is responsible for updating the local structures of the processor with the completed tasks.

A third message type exists, that is only sent at the beginning of the first round by the coordinator. It contains the permutation of the order of the tasks that the processors in the group need to use in order to get assigned tasks randomly. This message has the following format (without the spaces):

```
    REPORT SEP RND SEP NUM SEP DONE SEP STRING-SEP RESULT SEP
STRING-SEP SEP PERMUTATION SEP NUM SEP STRING-SEP END
```

Until PERMUTATION the format of the message is the same as the previous two types of messages. The third part of the message is from PERMUTATION until END. If LBA2 is used, the processors will expect to receive the permutation from the leader on round 1. The first NUM after PERMUTATION denotes the number of incomplete tasks that will follow. The rest of the numbers are the number of the tasks in their new execution order. The leader creates this message by calling routine Create_Completed_And_Results_String() (creates the second part of the string), Create_Permutation_And_Msg() (creates the third part of the string) and Create_Report_Msg2() (attaches the other two parts in the first part and creates the final message). The other processors use routine Collate_Report2() to decode the message that is received from the leader. Routine Update_Completed_And_Results2() is responsible for updating the local structures of the processor with the completed tasks and creating the array of the incomplete tasks with the received permutation. In Table 12, the routines that handle messages are presented.

Routines that handle messages
char* Create_Completed_And_Results_String()
char *Create_Report_Msg()
void Update_Completed_And_Results(int *, int *, int)
void Collate_Report(char *)
void Collate_Report2(char *)
char *Create_Report_Msg2()
void Update_Completed_And_Results2(int *, int *, int, int*)
char* Create_Permutation_And_Msg()

Table 12: Message handling routines

5.5 Regrouping mechanisms

In order to get comparable measurements it was necessary be able to reproduce regrouping patterns, so specific regrouping patterns were designed and implemented to be used in the experimentation phase. Additionally, a mechanism to create arbitrary patterns of regroupings was designed and implemented in order to provide a more completed solution.

5.5.1 Arbitrary Pattern

As the OMNI-DO problem is studied in partitionable networks we needed to develop a mechanism (that will run on each processor) to simulate network partitions. It essentially creates random patterns of regroupings while executing the implemented algorithm.

The idea is that the processors will be initially connected into a single group and will have all the knowledge needed to solve the problem. The system will start executing the algorithm and at random times processors may or may not leave the group and connect to other groups, this way simulating a processor or link crash. At some point (and again randomly) some processors may connect back to the initial group, stay connected to their current group or connect to yet another group. Table 13 includes the variables and routines that are used to achieve this.

Arbitrary Regrouping mechanism variables & routines	
ActivateVar	If this variable is set to 1 the processor will leave the group it is in, in the current round
Limit_Min	Set to 1 (part of facility that decides if ActivateVar is set to 1)
Limit_Max	Set to 100 (part of facility that decides if ActivateVar is set to 1)
Limit_Pct	The percentage that ActivateVar is set to 1 (part of facility that decides if ActivateVar is set to 1). If its value is 10 then ActivateVar has 10% possibility to be set to 1.
Group_Min	The minimum number of groups that can exist concurrently
Group_Max	The maximum number of groups that can exist concurrently
GroupVar	The number of the group the processor will join
void join_group()	To join a group
void GetSeed(int rank)	To initialize routine that provides random numbers
int Random_Integer(int low_num, int high_num)	Routine that provides random numbers between to limits

Table 13: Arbitrary Regrouping mechanism elements

While the OMNI-DO algorithm is being executed, each processor will set ActivateVar with a random number between Limit_Min and Limit_Max inclusive. If that random number is between Limit_Min and Limit_Pct then the processor will attempt to leave the group and join another one. To that effect, it will compute a random number between

Group_Min and Group_Max and set it in GroupVar. If the new number corresponds to a different group to the one the processor is already in, then it will leave the current group and join the one corresponding to GroupVar. To get random numbers, routine Random_Integer(int low_num, int high_num) is called with parameters being the lower and upper limit between which the number must be. It is called when deciding the values of both ActivateVar and GroupVar. This routine is based on using srand() which is pseudo-random number generator and a seed. Since we don't want all the processors to get the same order of numbers each time, a unique seed is given to each processor when they are initialized. To achieve that, the seed of each processor is computed using routine GetSeed(int rank) that uses the rank of the processor and current clock time to compute the seed.

5.5.2 Specific Patterns (SGSM and OUG)

In order to get some useful results it is necessary to design specific regrouping scenarios to examine specific situations.

One such scenario is when each processor is disconnected from all others and is included in a singleton group. At some point a regrouping occurs that merges some of those groups so that some processors are connected together. The regrouping pattern continues to slowly merge the remaining groups until only one group remains. We will call this pattern Singleton Groups Slowly Merge (SGSM). An example of this pattern is shown in Figure 13.

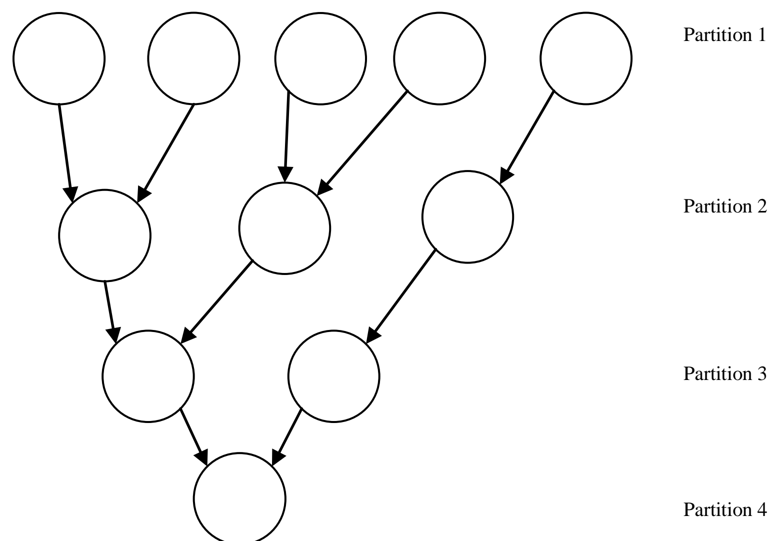


Figure 13: Specific regrouping pattern – Singleton Groups Slowly Merge (SGSM)

Another interesting scenario is when all processors start the computation in the same group and the regrouping simulation alternates between one group and many groups until the computation of tasks is completed. We will call this pattern One Unstable Group (OUG). An example of this regrouping pattern is shown in Figure 14.

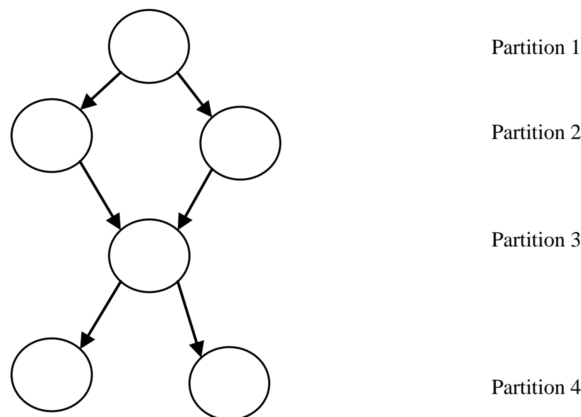


Figure 14: Specific regrouping pattern - One Unstable Group (OUG)

For these scenarios, some of the parameters used for arbitrary regroupings are used as well and can be seen in Table 14.

Specific Regrouping: variables & routines	
Limit_Pct	The number of tasks after which the transition will occur
Group_Max	The number of groups in which the one group of processors will transition to or the singleton groups that will be joined in each partition
GroupVar	The number of the group the processor will join
void join_group()	To join a group

Table 14: Specific Regrouping mechanism elements

Group_Max limits the number of groups in any partition when using the arbitrary regroupings pattern. We use the same variable when using the specific regrouping patterns for similar reasons. For SGSM, this variable defines how many groups will be joined into one during each merge. Put simply, it defines how quick the singleton groups will be merged into one group. In OUG, the number of groups formed after a fragmentation of the unstable group, are defined by Group_Max. In figures 13 and 14 Group_Max = 2.

Limit_Pct specifies the number of tasks that a processor completes after which a regrouping occurs, when used for SGSM and OUG regrouping patterns.

5.6 Client Logfiles

When a client is executed a method is required to distribute to all clients information regarding the current execution of the algorithm. It is essential to know how many processors are participating, how many tasks need to be completed and how to complete them. For this purpose, each client is initialized with some default values but the option is available to provide a configuration file that the processor will read during initialization. This input file should be called "parameters.param" and should include ten lines that correspond to the values that can be parameterized for each execution of the algorithm. Each line can have maximum thirty characters and must have the following format:

```
"Parameter Value" "Parameter Description"
```

Additionally to the number of processors, number of tasks and task type, some configuration variables regarding the regroupings algorithm need to be set as well. Figure 15 contains the configuration file for a setup of two processors and fifty tasks, were the processors may join groups 1, 2 and 3 during random regroupings. The Task type is 2, meaning computation of realistic tasks (for simple tasks the value should be 1) with the number of π digits to compute for each task is 2000.

The rest are variables that enable regroupings and are described in the previous section.

Parameter file sample
2 Processors
50 Tasks
1 Limit Min
100 Limit Max
10 Limit Pct
1 Group Min
3 Group Max
2 Task Type
2000 pi digits

Figure 15: Parameter file sample

Each processor creates an output report file that contains a trail of its execution and is restricted to the knowledge the particular processor had during the execution. This is

sensible since the processor only has information for the processors of its group at any given moment and cannot know information on crashed processors or processors connected to other groups. At initialization the processor prints out the parameters it reads from the input file for verification purposes. Each time the processor joins or leaves a group it records this event in the file. As soon as it joins a group it reports whether it was chosen to be the coordinator of the group. It also reports which task it executed in each round. When it knows that all tasks are completed it reports the following statistics:

- Number of tasks the processor itself executed
- Number of messages the processor sent
- Time in seconds that the algorithm needed to execute on this processor (at what time the processor became idle forever).

The file name of the statistics file of each processor has the following format where RANK is the rank of the processor that creates the file:

```
Processor RANK output file.txt
```

The algorithm ends when all non-faulty processors become idle. The application checks if a statistic report is created for all processors in the computation, to verify that all processors have finished their computation and exit. The runtime we count is the average completion time over all processors. Note that this check is only done for experimentation purposes. In a real distributed application this check is neither possible nor necessary, since the applications purpose is to be able to give response to queries about task results.

5.7 Compiling the Application

For the implementation of this application, the sample c client was used as a starting point. Several modules were created to expand the existing functionality to include the OMNI-DO algorithm, the regrouping mechanisms and the handling and execution of two types of tasks. A summary of the modules and the files created for each can be seen in Table 15.

Source files	
AlgAX_Client.c	Main C application file
Tasks_Module.h Tasks_Module.c	Header and source file for task handling module
GroupSim_Module.h GroupSim_Module.c	Header and source file for fragmentation/merge algorithm module
AlgAX_Module.h AlgAX_Module.c	Header and source file for algorithm AX module

Table 15: Source files

The source files are located under "Ensemble root/Client/c". To compile the client using the new application source file and the module files it is needed to edit the "makefile" of the client that is located under the same path. Initially, the following section exists in the file:

```
OBJECTS = \
        ens_utils$(OBJ) \
        ens_hashtbl$(OBJ) \
        ens_connection$(OBJ) \
        ens_comm$(OBJ) \
        ens_threads$(OBJ)
```

It will need to be modified to include the objects for the additional modules. For this implementation it looks like this:

```
OBJECTS = \
        ens_utils$(OBJ) \
        ens_hashtbl$(OBJ) \
        ens_connection$(OBJ) \
        ens_comm$(OBJ) \
        ens_threads$(OBJ) \
        AlgAX_Module$(OBJ) \
        GroupSim_Module$(OBJ) \
        Tasks_Module$(OBJ)
```

There is also another section that denotes that c_mtalk is an executable.

```
DEMOS = \
        $(ENSBIN)/c_mtalk$(EXE)
```

Since c_mtalk is a sample application it is denoted as DEMO. For this implementation another section similar to the above is inserted to denote which is the executable for this application. It looks like:

```
IS   = \
     $(ENSBIN)/AlgAX_Client$(EXE)
```

After the static library section a section with paragraphs that start from $(ENSBIN) can be seen. c_mtalk has its own section in this location. It is necessary to create one such section for the AlgAX_Client application. It looks like:

```
$(ENSBIN)/AlgAX_Client$(EXE):  AlgAX_Client$(OBJ) $(ENSLIB)/libens$(ARC)
        $(CC) $(CFLAGS) -o $(ENSBIN)/AlgAX_Client$(EXE)\
         AlgAX_Client$(OBJ)\
         $(LIB_PATH)$(ENSLIB) $(LIB_PREF)ens$(LIB_SUFF) $(LINK_FLAGS)
$(THREAD_LIB)
```

After the above modifications are made running the “make all” command from Ensemble root, will compile the source files and the client will be ready for use.

Chapter 6

Empirical Evaluation

In this chapter an empirical evaluation of the algorithm is presented. In particular, five experiments consisting of several scenarios are conducted and associated plots depicting the results are analyzed.

6.1 Experimentation setting

For our experiments we used an Intel Core 2 Quad 2.5GHz CPU-machine running the Ubuntu 8.04 operating system. Running experiments locally adds an extra margin for discrepancies since processes may be scheduled in different order or for different duration by the operating system, each time. For the specific regrouping patterns (SGSM and OUG) each scenario is run three times, and each plot point in the graphs represents the average of the three runs. For arbitrary regrouping patterns, where each run might be executed on a slightly different regrouping pattern, each plot point represents the average over ten runs. Some test scenarios were run during a pre-experimentation phase to verify that the implementation is correct and define which experiments to conduct.

In order to run scenarios with as many processors as necessary, a script code is written to start the clients easily. The script can be used as follows:

- Create a file named for example “run.sh” and enter the line “AlgAX_Client &” as many times as the number of the processors. For example for a simulation with five processors there should be five lines.

Each processor creates a file which has the name RANK.out, were RANK is the number of its rank when the simulation is completed. To gather all the statistics of the run together and easily import them into MS Excel to create graphs, another script is written to create a file named “all.stats”. An additional script is written to remove log and statistic files

created by each processor in order to run other experiments quickly. The scripts can be seen in Table 16.

Create "all.stats"	Remove client output files
#!/bin/bash FILES="*.out" rm all.stats for file in $FILES do cat $file >> all.stats done	#!/bin/bash rm *.out rm {*.txt rm run.txt

Table 16: Script for gathering statistics

6.2 Experiment 1: Effect of the number of Processors (P) and Tasks (N)

In our first experiment we focus on investigating how the number of processors P and number of tasks N affect the performance of the algorithm. This experiment is also important to help us identify sensible values for N and P to use in further experiments. We run this experiment for specific regrouping patterns (SGSM and OUG) and arbitrary regrouping patterns. For this experiment we run scenarios with the following parameters:

- Task Allocation = LBA2 – We choose LBA2 since the theoretical findings in [10] suggest that it results to reduced task execution redundancy when merges occur (in comparison to LBA1).
- Limit Pct = 10 – Recall that in the case of specific regroupings this represents the number of tasks that each processor will execute before a regrouping occurs. We want this value to be low enough such that regroupings occur during our experiments, since our goal is to evaluate the algorithm under various patterns of regroupings. We also want this value to be high enough such that the processors are able to execute some tasks before regroupings occur. If regroupings occur too often then communication overheads are heightened and results do not reflect the performance of the algorithm. In the case of arbitrary regroupings, Limit_Pct is the probability percentage that a processor will leave its group in each round. Imagine having 30 processors with each having 10% possibility to change group in each round. We would certainly have a regrouping each round. During the pre-experimentation phase we discovered that 0.4% is a good value to set when executing

scenarios for arbitrary regroupings. So for arbitrary regroupings we set Limit_Min = 1, Limit_Max=1000 and Limit_Pct=4.

- Group Max = 2 or P – Recall that in the case of the SGSM pattern this variable represents how many groups will be joined in each regrouping, in other words the rate that singletons will merge to one group. We do not want this to be too soon, as to be able to better observe the performance of the algorithm. In the case of the OUG pattern, this variable represents the number of groups that the unstable group will fragment to. During the pre-experimentation phase we noticed that it does not affect the performance for that particular pattern so we leave it at 2. For arbitrary regroupings we set Group Max = P, that is we allow the possibility of having partitions of P singleton groups.
- Task Type = 1 (simple) – We do not want the execution of the tasks to dominate the computation. Using simple tasks allows us to observe the effect of P and N on the performance of the algorithm.

We present two scenarios for this experiment.

6.2.1 Scenario 1.1: Number of Processors

In this scenario we investigate how the number of processors affects the OMNI-DO algorithm in SGSM, OUG and arbitrary regrouping patterns. The scenario is run on a system with 10, 20, 30, 40 and 50 processors. We run the same scenario for different numbers of tasks as well, specifically for 100, 300, 600 and 900 tasks.

In all three regrouping patterns we anticipate increase of work, message complexity and execution time in relation to processor number increase. In OUG pattern, we anticipate less work and message complexity in comparison to SGSM regrouping pattern. In the latter, each processor works in a singleton group for some time, before it starts merging with others introducing communication costs. In OUG, processors start in the same group for a while, they share knowledge regarding completed tasks resulting to less work and coordinating messages.

In Figure 16 we can see how work changes when we increase the number of processors for the different types of patterns.

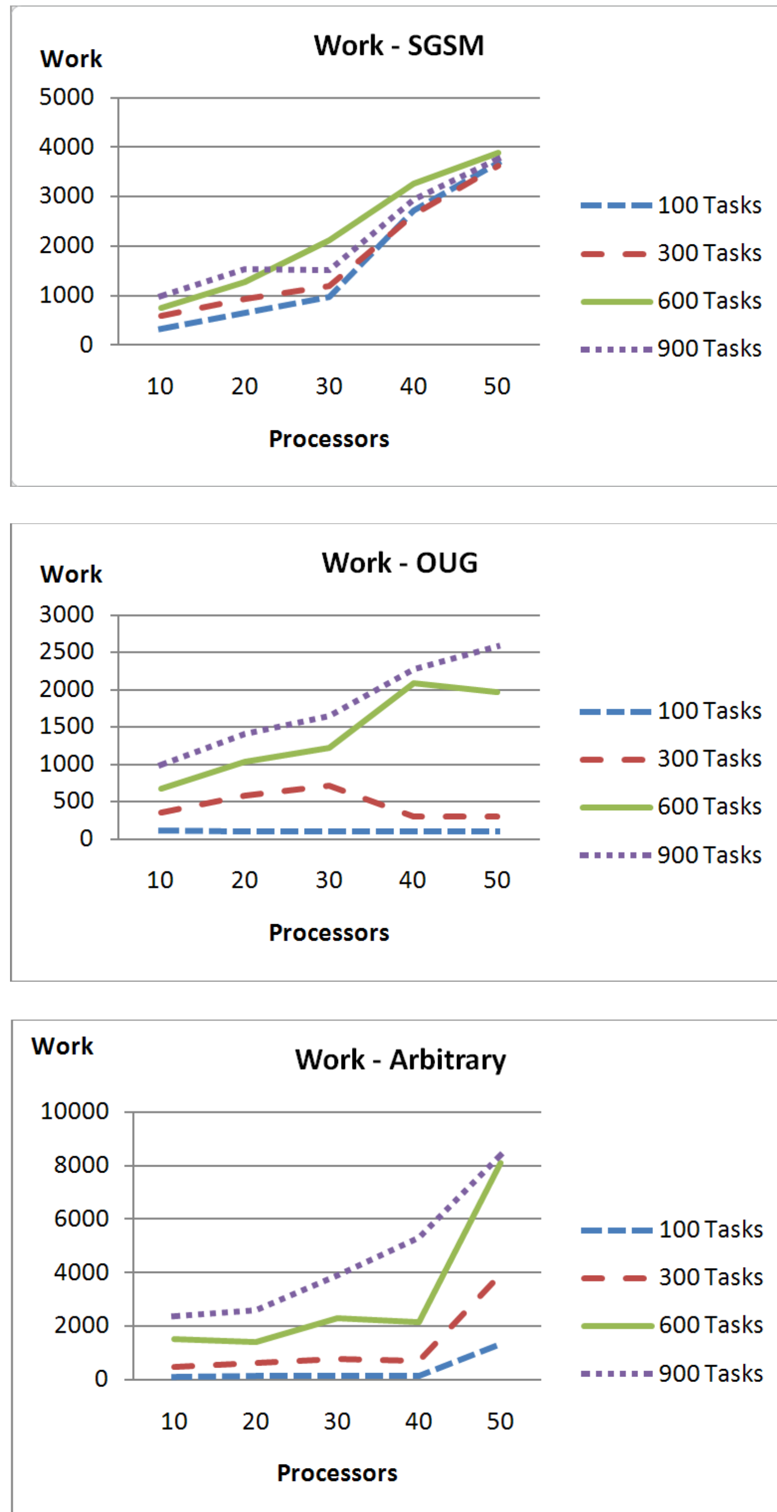


Figure 16: Work - Effect of processor number

When increasing the number of processors in all pattern types, work increases due to increased communication costs (as explained in the next paragraph). Depending on the

regrouping pattern, there is higher task execution redundancy as the number of processors increases. In the graph depicting the OUG pattern it can be seen that for 100 and 300 tasks, the work is not particularly affected by the number of processors in contrast to higher numbers of tasks. This behaviour is a consequence of the regrouping pattern. All processors start executing in the same group and complete most of the tasks before splitting. For example, when P=20 and N=300, since Limit_Pct = 10 they complete 200 tasks before the first transition occurs. This keeps work relatively steady. For 600 and 900 tasks we see that work is increased when the processor number increases, due to increased communication costs since the regrouping pattern gets a chance to have effect. As expected, the work from the OUG pattern is less than the work from the SGMS pattern. In arbitrary regrouping patterns we see much higher work and message complexities. By reviewing the data we gathered we see that some processors permanently join singleton groups causing this behaviour.

Have in mind that while in theory two groups merge at the same time, in reality it takes some time for the group communication service to create a group that contains all processors of the two groups. During this group stabilization phase processors keep executing tasks as the algorithm dictates and work is increased. In the SGSM pattern (where only merges occur) we notice that the slope becomes more steep after P=30, except in the case of N=600 where the slope is smooth (but work is greater). The increased steepness of the slope after P=30 (inclusive) indicates that work done during the group stabilization phase is greatly increased due to more processors. The smoothness of the slope in the case of N=600 indicates that the work done during the group stabilization phase is higher for all numbers of processors for that number of tasks. The combination of the parameters used for this experiment and N=600 is the worst case scenario for work, for this pattern.

Recall that for the work complexity of algorithm AX (the basis of the implemented algorithm), for any pattern of fragmentations and merges, the upper bound of $\min\{N \cdot f + N, N \cdot P\}$ was established. In the case of OUG (where there are both fragmentations and merges), each partition occurs after 10 tasks are completed by each processor. If two processors are in the same group, then the 10 tasks are executed concurrently on each

processor. Therefore when P=40 and N=900 the first partition occurs after 400 tasks are completed. The first partition is a fragmentation and since Group_Max=2, two new groups will be created. The two groups complete concurrently 200 tasks each and then merge into one group. After that, another fragmentation and another merge occur before the system completes its computation with work = 2278. As a consequent we see that for that particular scenario m=2 and f=2. We see that $N \cdot f + N = 2700 < N \cdot P = 36000$ and 2278 < 2700 which satisfies the upper bound. Hence, our experimentation results fall within the proved upper bound results.

In Figure 17 we can see how message complexity changes when we increase the number of processors for the different types of patterns. Notice that message complexity graphs are identical to work graphs for all patterns, with message complexity being slightly higher than work. This is natural since for each task it completes the processor sends a message in the group. Message complexity is slightly higher because when a new group is created (including the initial group), coordination messages are sent that are additional to the messages sent for sharing results of executed tasks.

For algorithm AX, the upper bound of $4(N \cdot f + N + P \cdot m)$ was established for message complexity. For OUG with P=40 and N=900 we showed that m=2 and f=2. Therefore as 2457 < 11120 the upper bound is satisfied.

In Figure 18 we can see how execution time changes when we increase the number of processors for the different types of patterns. In the case of SGSM we observe that while execution time increases as the number of processors increases, until P=30, it is decreased for more processors. Until P=30 execution time is increased since merging communication cost is increased. However, when even more processors are available to the system, execution time needed for all the tasks to be completed is reduced (when merges occur more completed tasks are known to the system). In any case, the particular combination of parameters, having P=30 and in the particular regrouping pattern the algorithm has the worst performance regarding execution time. In the case of OUG we observe that for few processors (P=10 and P=20) more time is needed to execute the algorithm than with more processors (P=30 and P=40). This

observation can be made in the case, of N=600 and N=900 where the regrouping pattern can affect the algorithm. This indicates that with more processors tasks are completed faster. However, in the case of P=50 we observe a high increase of execution time. This indicates that this number of processors causes a high overhead in communication (e.g., all processors in a group must send a unicast message to the coordinator each round) that overshadows the reduced task redundancy benefit of the algorithm in these numbers of tasks. Recall that in this pattern all the processors are in the same group most of the time. In the case of the Arbitrary regrouping pattern we observe that for P=50 the execution times are much higher than the rest. This reinforces the indication that this number of processors (with these numbers of tasks) causes a high overhead in communication.

From this scenario we realize that we should select 40 processors for further experiments to allow the regrouping patterns to have effect and avoid high communication overheads.

6.2.2 Scenario 1.2: Number of tasks

As an extension of the previous scenario, we investigate how the number of tasks affects the OMNI-DO algorithm in SGSM, OUG and arbitrary regrouping patterns. We use data gathered from Scenario 1 and create different plots that show how work, message and execution time are affected when the number of tasks changes. We expect to see increase of work, message and execution time when the task number increases. In Figure 19 we can see how work changes as the number of tasks increases.

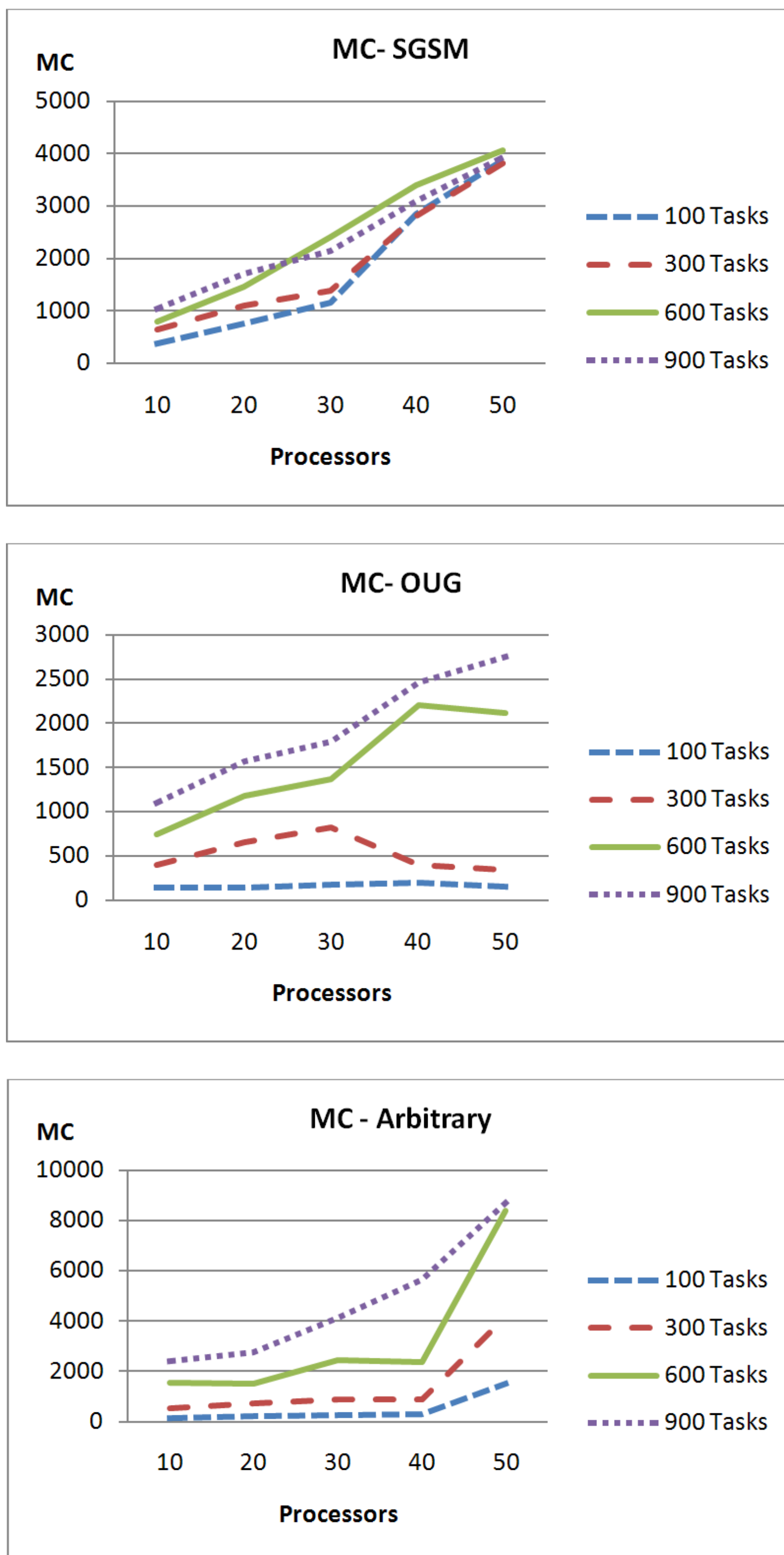


Figure 17: Message Complexity - Effect of processor number

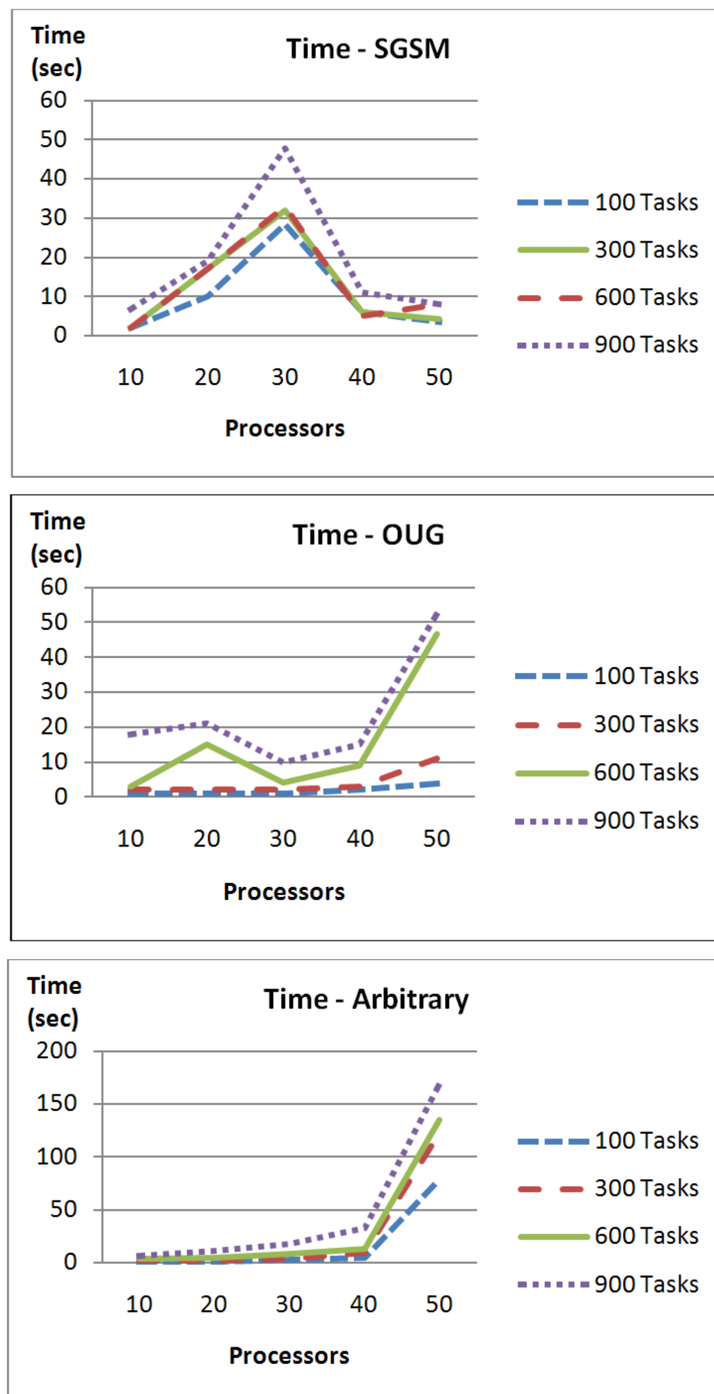


Figure 18: Time – Effect of Processor Number

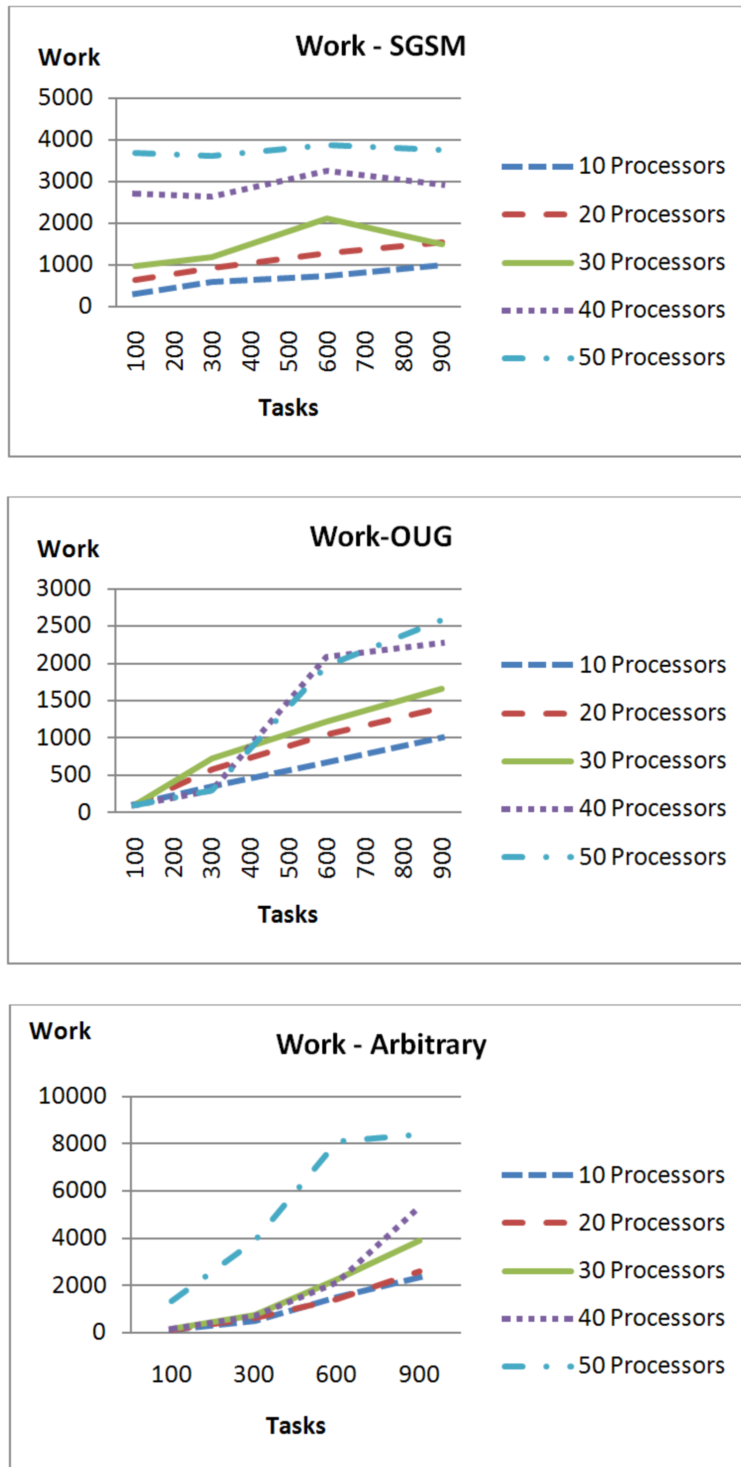


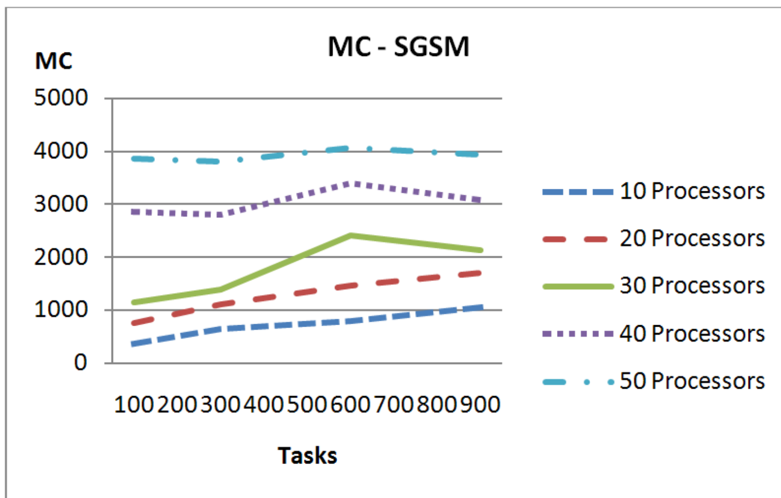
Figure 19: Work - Effect of task number

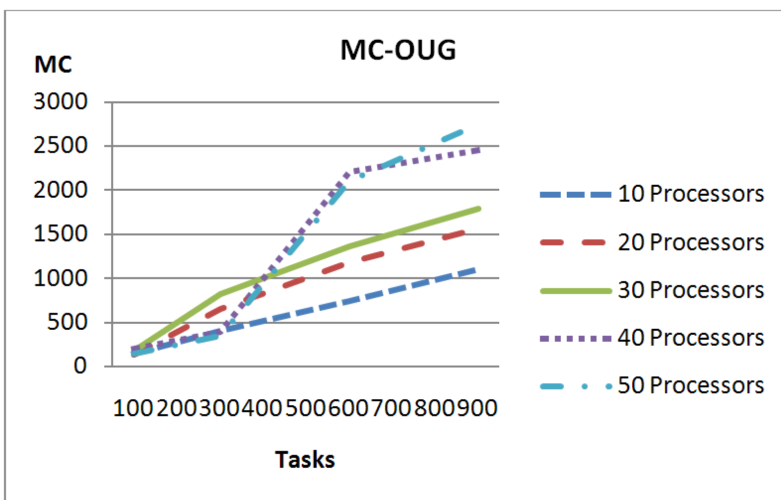
With the pattern SGSM, work is not particularly affected by increasing the number of tasks. All processors start executing in singleton groups and then proceed to slowly merge to one group. The results we see indicate that, while the processors are in separate groups, they choose different tasks to complete (due to the scheduling algorithm). When they merge they

have a higher number of completed tasks and that reduces pending work, and consequently overall work. This indicates the good load balancing rule of the algorithm in such a regrouping pattern; the work is independent of the number of tasks under this regrouping pattern.

In other regrouping patterns, the work is increased when increasing the number of tasks. This is only natural due to the fact that the work load of the system is increased and fragmentations occur (which do not in the case of the SGSM pattern). For the OUG pattern, we notice that for 40 and 50 processors, we get a dramatic increase of work for 600 and 900 tasks due to the regrouping pattern. At high numbers of tasks, the original group fragments and merges more times. Fragmentations cause execution of more redundant tasks and merges cause overhead in communication. For the Arbitrary pattern, we observe that for P=50 the work is much higher (see Scenario 1.1 for further details). From this scenario we realise that we should choose a high number of tasks (600 or 900) to give a chance to the OUG pattern to have effect.

In Figure 20, we can see how message complexity changes when task number is increased. As expected, we observe similar behaviour for message complexity as in work for this scenario. In Figure 21, we can see how execution time is affected when task number is increased. In the SGSM pattern, we observe generally steady plot lines when the number of tasks is increased, as we observed in the case of work for the same pattern. We also observe that for P=20 and P=30 the runtime is higher than the rest (see Scenario 1.1 for further details). For the OUG and Arbitrary patterns, we observe that execution times are increased with increasing numbers of tasks. Execution times are greater for P=50 which reinforce the indication that this number of processors causes a high overhead in communication (see Scenario 1.1 for further details).





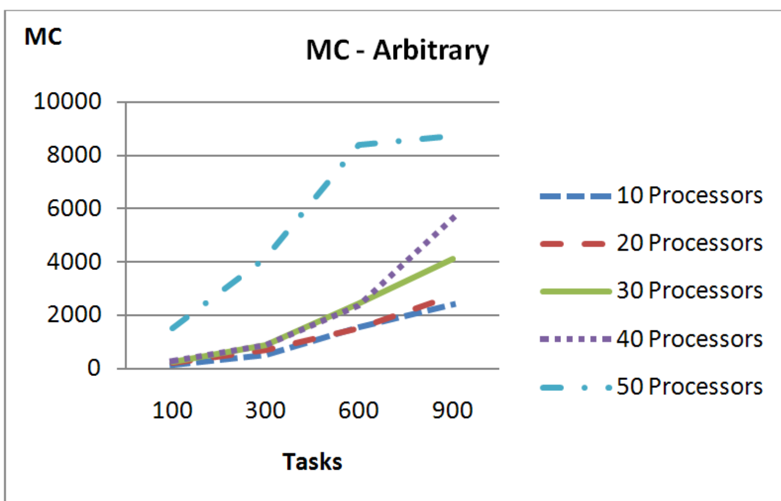


Figure 20: Message Complexity - Effect of task number

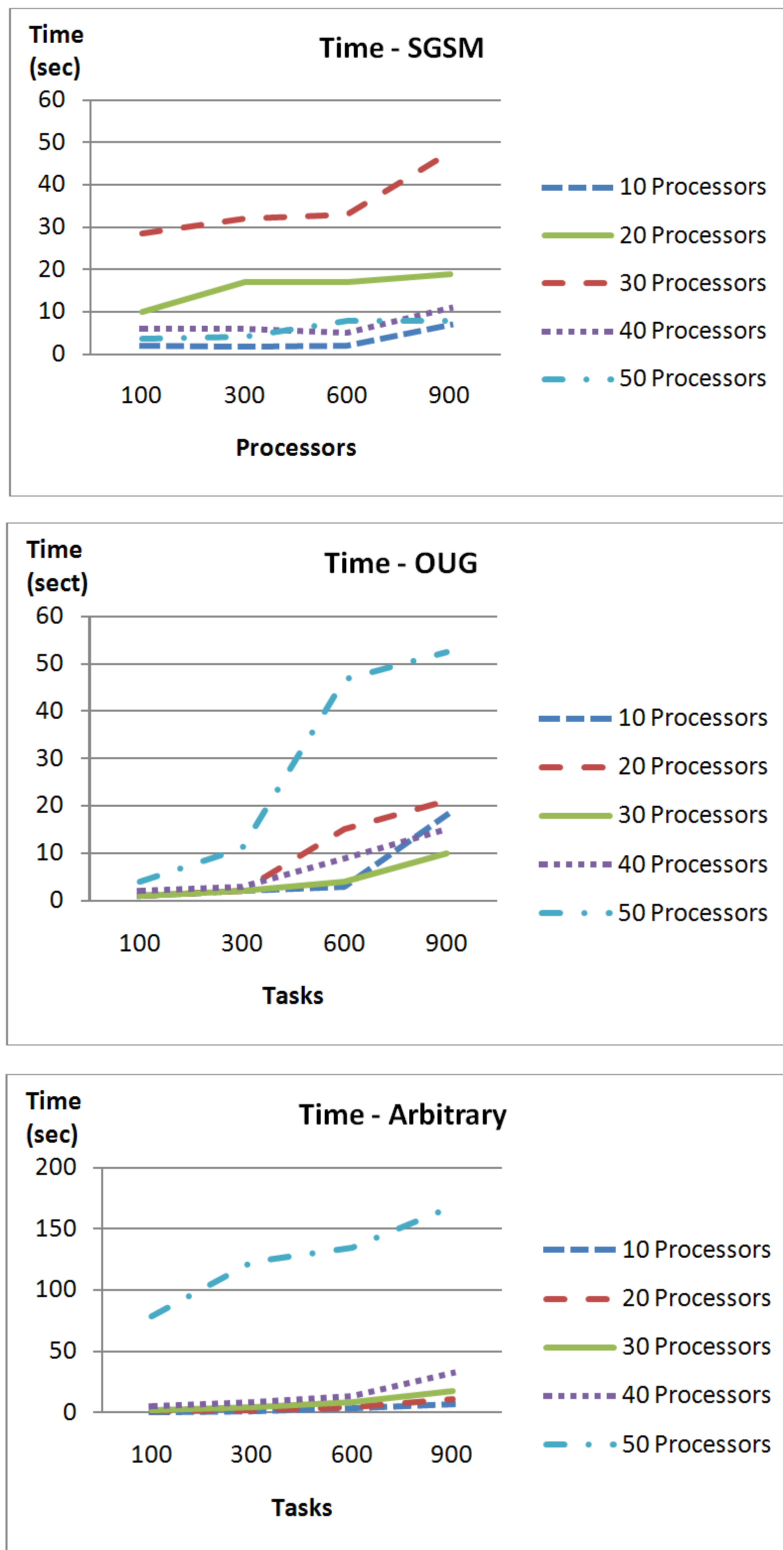


Figure 21: Time - Effect of task number

6.2.3 Summary

For all patterns, increasing the processor number increases the communication overheads, thus work, message and execution times are increased. We would rather choose

P=40 to allow the regrouping patterns to have effect and also avoid high overhead in communication by using 50 processors. Increasing the number of tasks, the work, message and execution time increase due to fragmentations (as one would expect), except in SGSM were the load balancing rule assigns tasks in a way that minimizes redundancy across groups and there are no fragmentations. We would rather choose a high number of tasks (600 or 900) in order to allow the other regrouping patterns (OUG and Arbitrary) to have effect.

Overall, we choose the combination of 40 processors and 900 tasks to continue our experiments, as these values seem to provide the most meaningful and interesting results.

6.3 Experiment 2: Effect of Group_Max Variable

This experiment aims to study how the severity of the networking failures that cause partitions (i.e., how many link failures and recoveries occur concurrently) affect the execution of the OMNI-DO algorithm. The number of link failures and recoveries that can occur concurrently is represented by the variable Group_Max. We run this experiment for specific regrouping patterns (SGSM and OUG) and arbitrary regrouping patterns. For this experiment we run a scenario with the following parameters:

- Task Allocation = LBA2 – for the same reason as Experiment 1.
- Limit Pct = 10 (or Limit_Min = 1, Limit_Max=1000 and Limit_Pct=4 for arbitrary regroupings) – for the same reason as Experiment 1.
- Task Type = 1 (simple) – for the same reason as Experiment 1.
- Tasks = 900 – as decided by Experiment 1.
- Processors = 40 – as decided by Experiment 1.

6.3.1 Scenario 2.1: Concurrent link failures and recoveries

In this scenario we investigate how the number of concurrent link failures and recoveries affects the OMNI-DO algorithm in SGSM, OUG and arbitrary regrouping patterns. The scenario is run for Group_Max values 2, 5, 10, 20, 30 and 40. Bear in mind that this variable affects differently each regrouping pattern as mentioned previously.

For SGMS, when the processors merge to a single group faster (due to more link recoveries), work and message should increase due to the communication overheads caused by merges. On the other hand, this should reduce work and message complexity, since redundant tasks are reduced. Overall, we do not expect the number of concurrent link recoveries to significantly affect the execution of the algorithm since overheads and less redundant task due to merges will balance each other out. For OUG, we do not expect to see much change either since all the processors are mostly in the same group and when they fragment into various groups our algorithm guarantees reduced task execution redundancy. For arbitrary regroupings we expect that increasing the number of concurrent link failures and recoveries will increase work and message complexity. In Figure 22 we can see how work and message complexity changes when we change the Group_Max variable.

In the case of SGSM, we observe that there is a decline in work and message complexity when Group_Max is increased since the processors merge to one group faster (more link recoveries). Work and message complexity remain relatively steady after Group_Max=10. However, when the value of Group_Max is near the number of processors, work and message complexity is increased again. This indicates that when all processors (that are in singleton groups) attempt to join the same group at the same time, the communication cost is greatly increased and has an effect on work and message complexity. Recall, that during the stabilization of a group (due to a merge) tasks continue to be executed and if more processors are part of the group, stabilization takes more time. In the case of OUG, as expected, work and message complexity are steady for most of Group_Max values. This behaviour empirically demonstrates the reduced task execution redundancy of the algorithm.

In the arbitrary regrouping pattern we observe that the fewer groups are allowed to be created the better the performance of the algorithm. When more groups are allowed then more singleton groups are created and more redundant tasks are executed (as seen from the gathered data). However, since Group_Max is less than P then non-singleton groups exist and processors in some singleton groups join other groups at some point. From Group_Max=20 and on, which is half the number of the processors in this experiment, we observe that work

and message complexity are reduced. In such high Group_Max values even more singleton groups (than with small Group_Max values) are created and most of them stay that way (very few processors that are in singleton groups join other groups). This indicates that allowing so many concurrent groups cause the algorithm to perform worse than if each processor were allowed to compute on its own (the case where more processors are in singleton groups). In a realistic situation, it is rare to have a distributed system with so many concurrent link failures and if it exists any algorithm could not perform any better.

In Figure 23, the execution time graphs for the same scenario can be seen. In the SGSM pattern graph (where only merges occur), we observe that execution times are not greatly affected by the number of concurrent link recoveries. For the OUG pattern, time is steady for most of Group_Max values (as in work and message complexity graph). This indicates again, that the algorithm provides reduced redundant task execution. For the Arbitrary pattern, the graph is again similar to the work/message complexity graph.

6.3.2 Summary

Running scenarios for SGSM shows that work, message and execution time are steady for most numbers of concurrent link recoveries. At very small values or values near the number of processors, they are increased. At small values the processors remain disjoint for longer and at high values more processors attempt to join the same group concurrently introducing further communication overheads.

For OUG work, message and execution time are relatively steady. This indicates once more that the algorithm provides reduced task execution redundancy due to its randomized inter-group allocation method.

The arbitrary regrouping graphs show that the algorithm has good performance unless there are many concurrent groups, which we already knew from the competitive analysis of algorithm RS in [10].

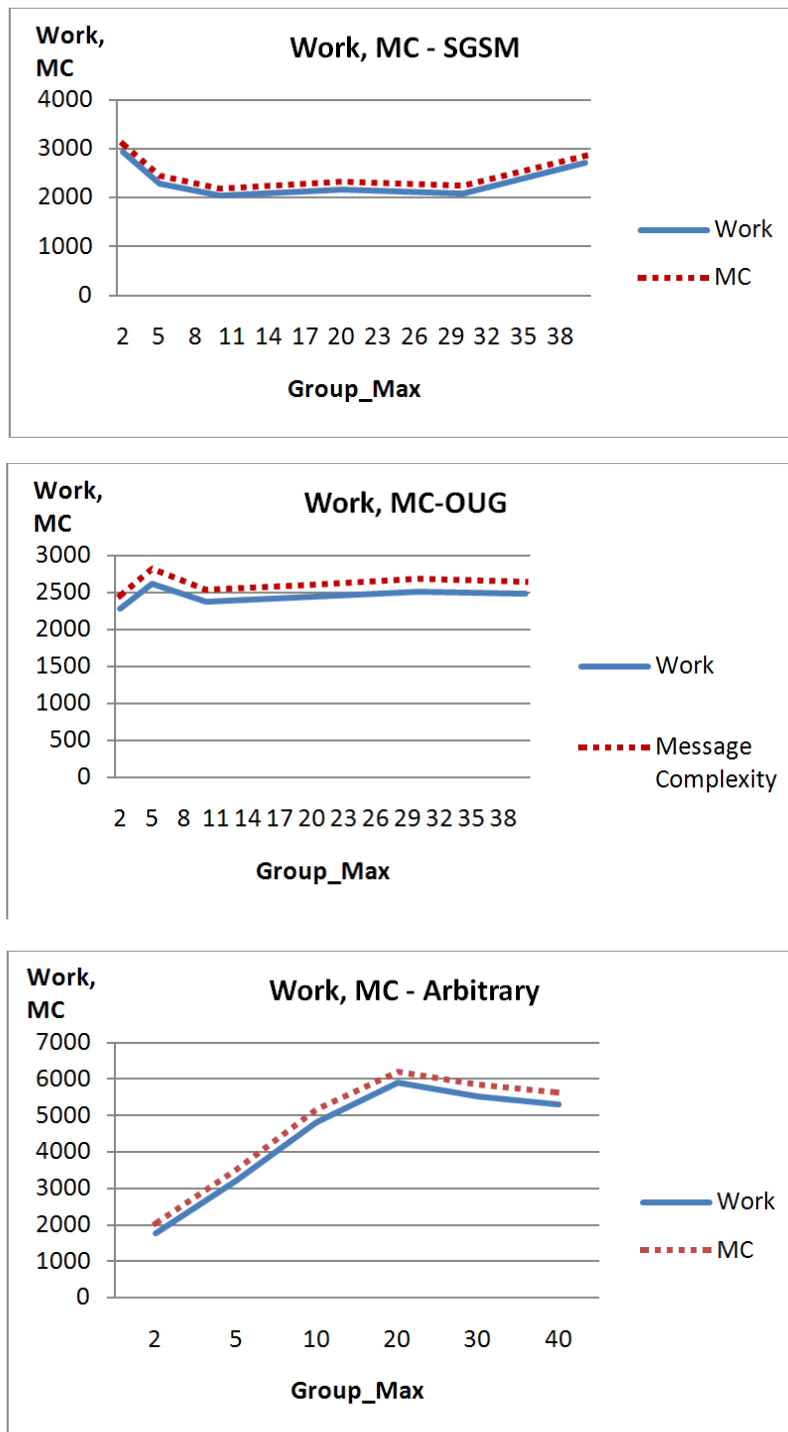


Figure 22: Work, MC - Effect of Group_Max

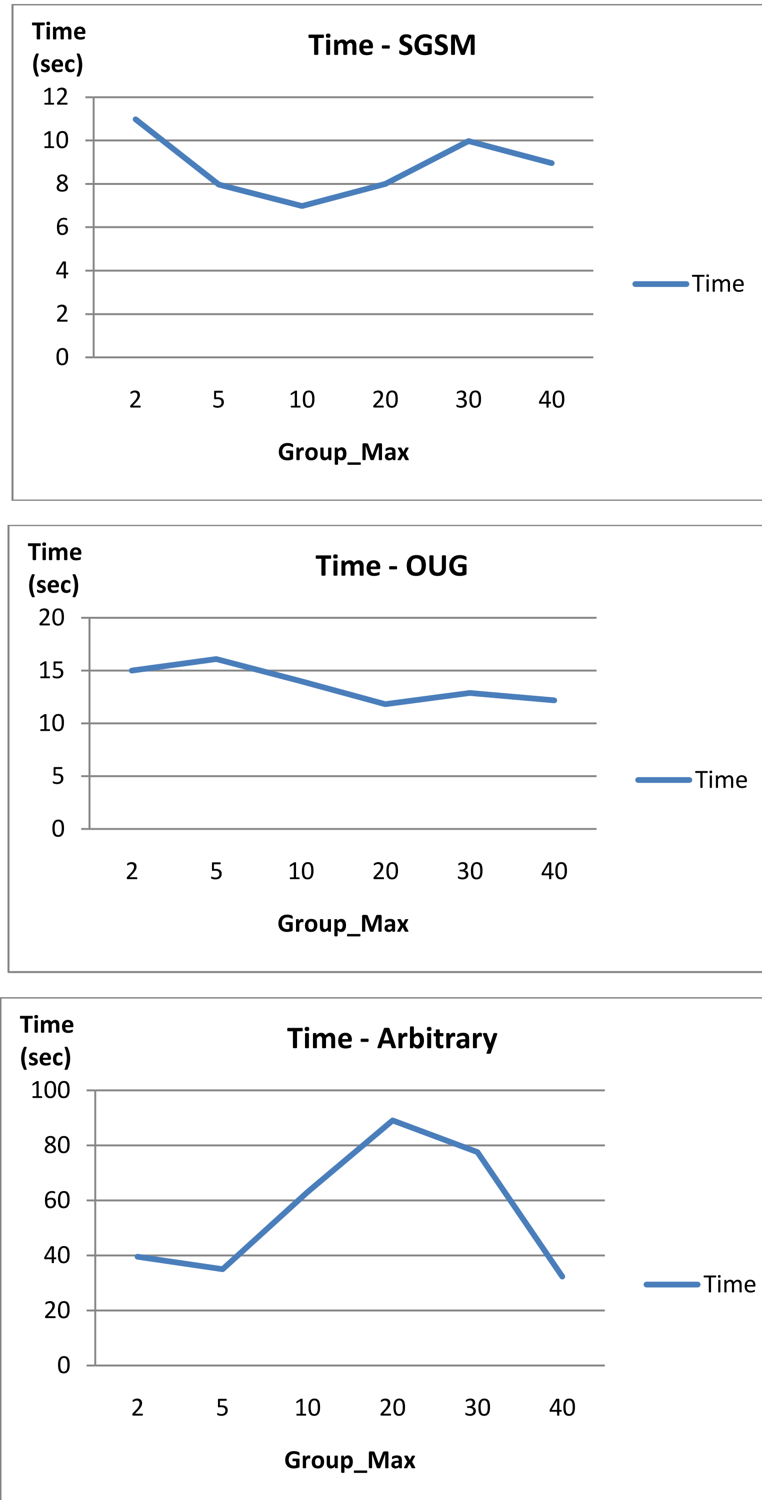


Figure 23: Time - Effect of Group_Max in SGSM

6.4 Experiment 3: Effect of regrouping frequency (Limit_Pct variable)

In this experiment we investigate how the algorithm fairs when the regrouping frequency increases. Specifically, we investigate how changing the value of variable

Limit_Pct (which represents the regrouping frequency) affects the algorithm. We run a scenario with the following settings:

- Task Allocation = LBA2 – for the same reason as Experiment 1.
- Group_Max = 2 – for the same reason as Experiment 1.
- Task Type = 1 (simple) – for the same reason as Experiment 1.
- Tasks = 900 – as decided by Experiment 1.
- Processors = 40 – as decided by Experiment 1.
- Limit_Max = 1000 – for the same reason as Experiment 1.
- Limit_Min = 1 - for the same reason as Experiment 1.

6.4.1 Scenario 3.1: Regrouping frequency

In this scenario we investigate how changing the value of Limit_Pct (how often partitions change) affects the OMNI-DO algorithm in SGSM, OUG and arbitrary regrouping patterns. The scenario is run for Limit_Pct values 5, 10, 15 and 20 for SGSM and OUG. For arbitrary regroupings it is run for 2, 4, 10, 15 and 20 (representing 0.2%, 0.4%, 1%, 1.5% and 2% possibility that each processor changes group in each round). Since this percentage is for each processor, having 40 processors increases the expectation that a regrouping occurs in each round.

We anticipate increased work and message complexity in high values of Limit_Pct for SGSM, since processors stay in separate groups for more time. In the case of OUG, we anticipate reduced work and message complexity in high values of Limit_Pct, since processors stay in the same group for more time and thus share task completion knowledge for more time. With arbitrary regroupings, we expect higher work, message complexity and execution time when this variable is increased, since the frequency of regroupings is increased.

In Figure 24 we see how work and message complexity are affected when Limit_Pct changes.

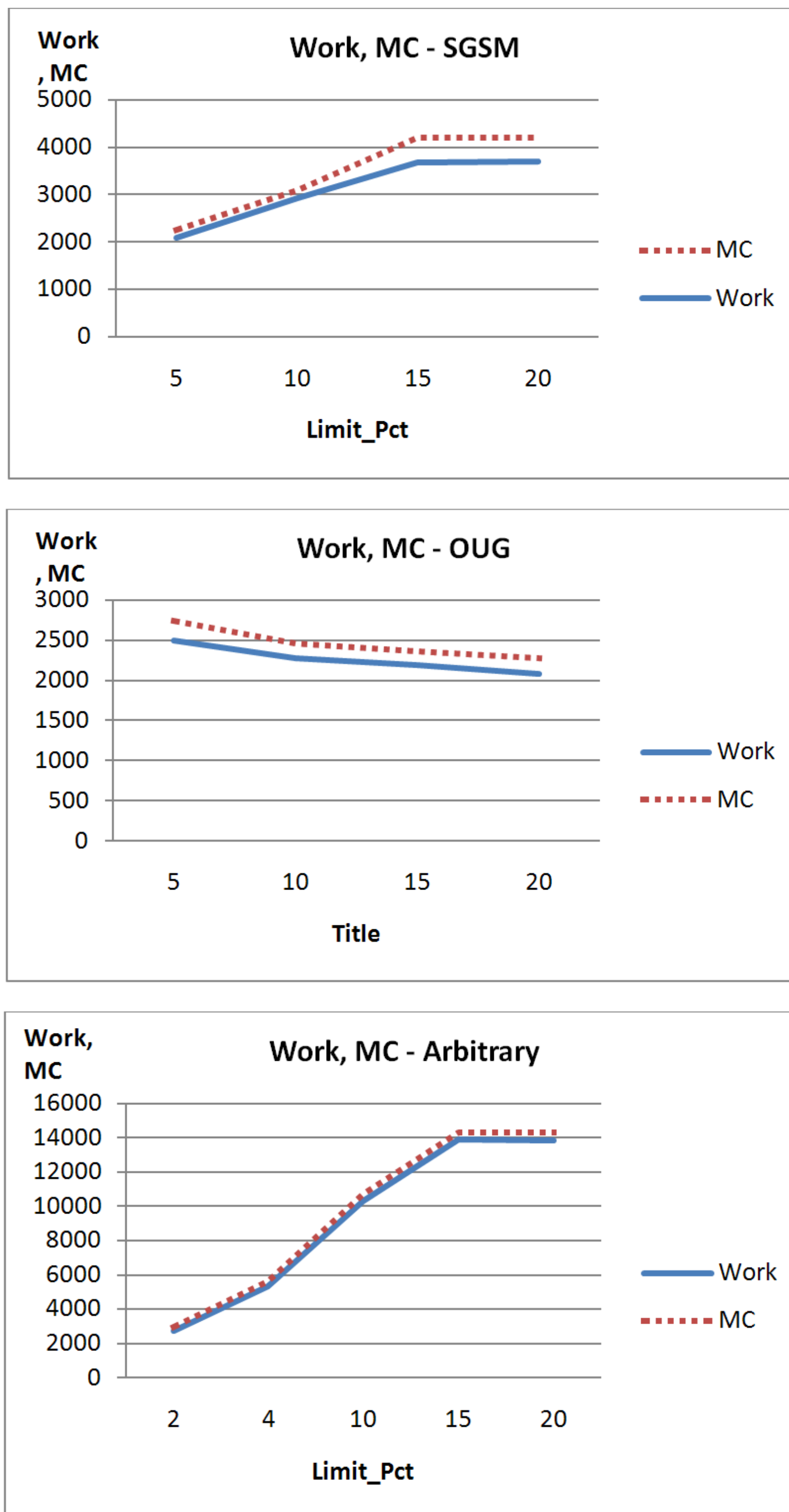
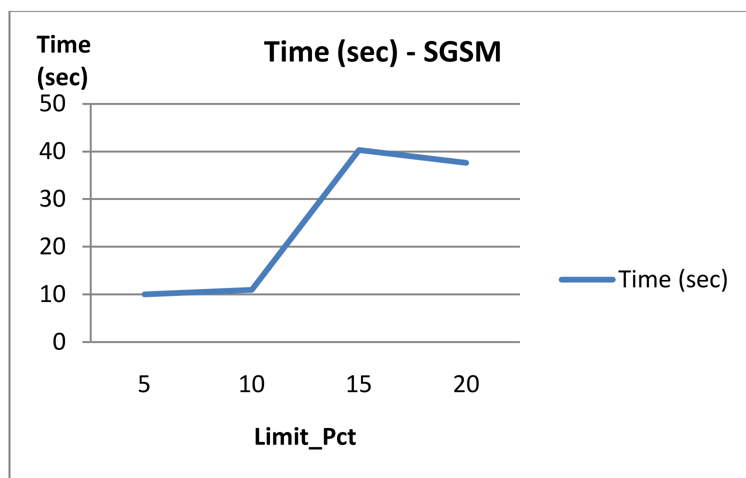
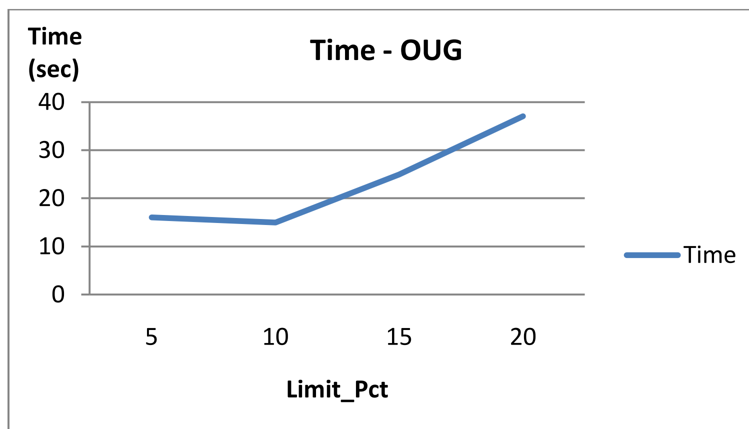


Figure 24: Work, MC - Effect of Limit_Pct

As expected, we see higher work and message complexity when Limit_Pct is increased for SGSM. This verifies our intuition that the longer it takes for processors to merge to one group, more redundant tasks are executed. For OUG we observe that delaying fragmentations (and consequently merges), work and message complexity are reduced

because all the processors are initially in one group. This provides them with an advantage since they complete more tasks without redundancy. In arbitrary regroupings we observe that increasing Limit_Pct dramatically increases work and message complexity. With a high regrouping frequency any algorithm behaves similarly. In Figure 25 we see how execution time is affected when Limit_Pct changes.





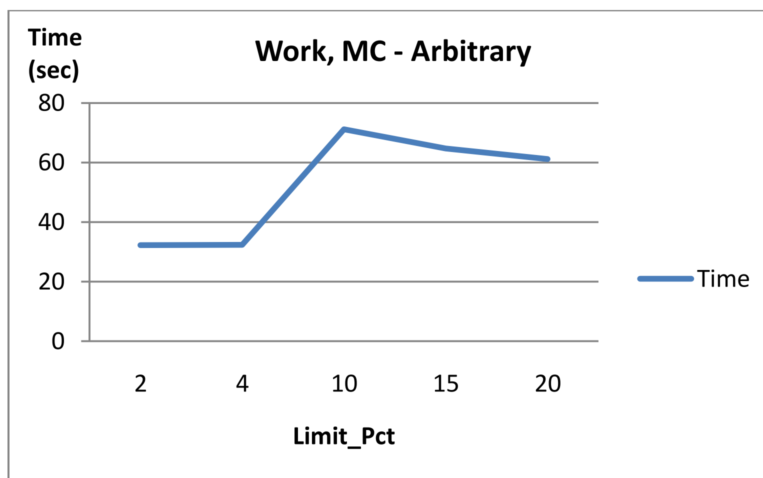


Figure 25: Time - Effect of Limit_Pct

Increasing regrouping frequency generally increases the execution time of the algorithm. The communication overhead due to the coordination needed when more regroupings occur, causes increase of execution time.

6.4.2 Summary

Summarizing the results of this scenario, we confirm our intuition that the longer it takes for processors to merge to one group, more redundant tasks are executed, thus work, message and execution time increase. The opposite happens if a group takes longer to fragment. If groups fragment frequently we observe high work and message complexity but execution times are relatively steady due to reduced communication overheads.

6.5 Experiment 4: Simple/Intensive Tasks

We introduce computationally-intensive tasks in our experiments. We want to investigate if and how the performance of the algorithm is affected when running simple and intensive tasks. Recall that intensive tasks are basically the computation of a number of π digits (see section 5.2 for details). We run a scenario with the following settings:

- Task Allocation = LBA2 – for the same reason as Experiment 1.
- Group_Max = 2 – for the same reason as Experiment 1.
- Limit Pct = 10 (or Limit_Min = 1, Limit_Max=1000 and Limit_Pct=4 for arbitrary regroupings) – for the same reason as Experiment 1
- Tasks = 900 – as decided by Experiment 1.
- Processors = 40 – as decided by Experiment 1.
- π digits = 150 – We keep this small number such that it takes more time to complete each task but we want to show that there is a measurable difference even with a small change in the task computation intensity.

6.5.1 Scenario 4.1: Simple vs intensive tasks

This scenario examines the performance of the algorithm between simple and intensive tasks in SGSM, OUG and arbitrary regrouping patterns.

We do not anticipate much difference between the two type of tasks in regards to work and message complexity but we do expect that it will be lower for intensive tasks. We expect more redundancy when running simple tasks, since they are completed faster and the processors probably manage to execute more of them between regroupings. We chose not to measure the time complexity as no interesting observations can be made; the time required computing the intensive tasks would dominate all other computations. In Figure 26, work and message complexity when executing with simple and intensive tasks can be seen.

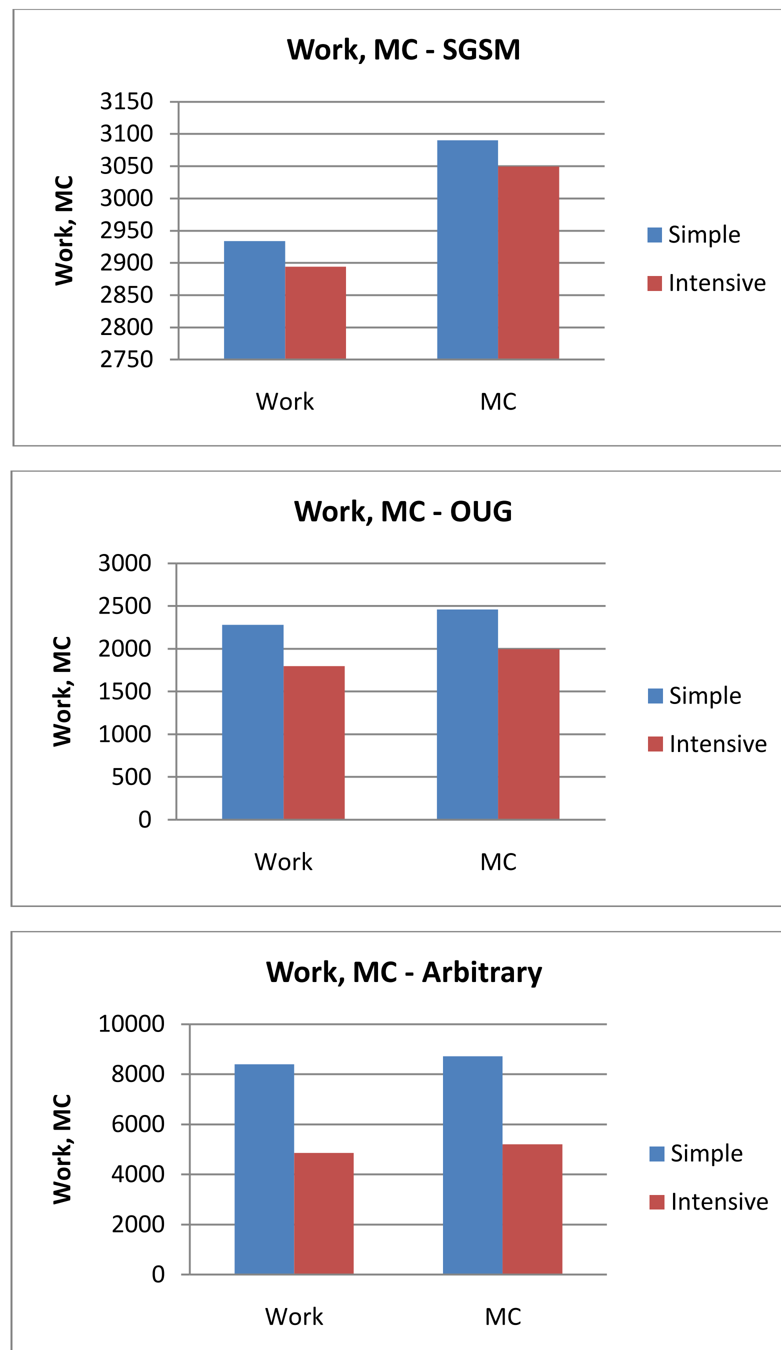


Figure 26: Work, MC - Effect of Task Type in SGSM

The algorithm performs a bit better when running with intensive tasks. In the case of arbitrary regroupings we see an even bigger difference. As mentioned above, more simple tasks (than intensive) can be executed during regroupings. While for the other two patterns the regroupings are predefined, in this case more regroupings occur (as seen from the data we gathered) and cause this difference. Considering intensive tasks are usually what distributed systems are built for, and hence these results give a good indication for the practicality of the algorithm.

6.5.2 Summary

Summarizing our findings for this scenario, the algorithm performs better when executing with intensive tasks rather than simple tasks. Since in realistic settings one expects to deal more with intensive tasks, and hence these results give a good indication for the practicality of the algorithm.

6.6 Experiment 5: Comparison of LBA1 and LBA2

We conduct this experiment to empirically verify that using LBA2 is better than using LBA1. We run a scenario with the following settings:

- Group_Max = 2 or 5 – For SGSM and OUG we use 2. For arbitrary pattern we have chosen to make this 5 in order to increase probability of merges. If no merges occur then LBA2 behaves the same as LBA1.
- Limit_Pct = 10 – for the same reason as Experiment 1.
- Tasks = 900 – as decided by Experiment 1.
- Processors = 40 – as decided by Experiment 1.
- Task Type = 2 (intensive) – in order to compare the two load balancing rules in a more realistic setting.
- Limit_Max = 1000 – for the same reason as Experiment 1.
- Limit_Min = 1 - for the same reason as Experiment 1.

- π digits = 300 – We increase the task intensity in this experiment because we want to get a more clear cut result.

6.6.1 Scenario 5.1: LBA1 vs LBA2

This scenario looks at the performance of the two different task allocations, LBA1 and LBA2, using patterns SGSM, OUG and arbitrary regroupings. The scenario is run only for intensive tasks. We expect to confirm that LBA2 rule is better than LBA1, when there are merges. Despite the overhead introduced by LBA2 in the first round of each new group (to identify the task permutation for the group), we expect time to be less as well (again when there are merges). Since intensive tasks are used, we expect that time saved due to less intensive tasks being executed is greater than the delay due to determination of the task permutation in each group.

In Figure 27 we see the work and message complexity for both rules. We observe that the work and message complexity are less when using rule LBA2 rather than LBA1, as expected. Have in mind that this scenario is run with N=900. In SGSM, LBA2 performed 2900 work (2000 redundant tasks) and LBA1 performed 3800 work (2900 redundant tasks). Therefore LBA2 performed 23.7% less work than LBA1. Similarly, in OUG LBA2 performed 20.5% and in arbitrary regroupings 20% less work than LBA1.

In Figure 28 we see the execution time for both rules. We observe that time is less when using LBA2 rather than LBA1, as expected, since merges occur. The difference in work and message is greater between the two rules when using the SGSM pattern rather than OUG. Regarding execution times however, the difference is greater when using the OUG pattern rather than SGSM. We observe that when using the LBA1 rule, the execution times are similar for both patterns (58 for SGSM and 52 for OUG). This indicates that regroupings in which many processors participate increase execution time (which is what happens with the OUG pattern). When using the LBA2 rule the execution time for the OUG pattern is a lot less than the SGSM pattern. In the OUG pattern, since reduced redundant tasks are executed and all processors often merge into one group, fewer regroupings occur before all tasks are

completed. As observed earlier (with experiment 3) regrouping frequency affects the execution time when using the OUG pattern. Since fewer regroupings occur when using the LBA2 rule (rather than the LBA1 rule) the execution time is naturally a lot less. In the case of SGSM, the number of regroupings is not affected by the load balancing rule. With the arbitrary pattern, execution times are generally higher for both load balancing rules but again LBA2 performs better than LBA1.

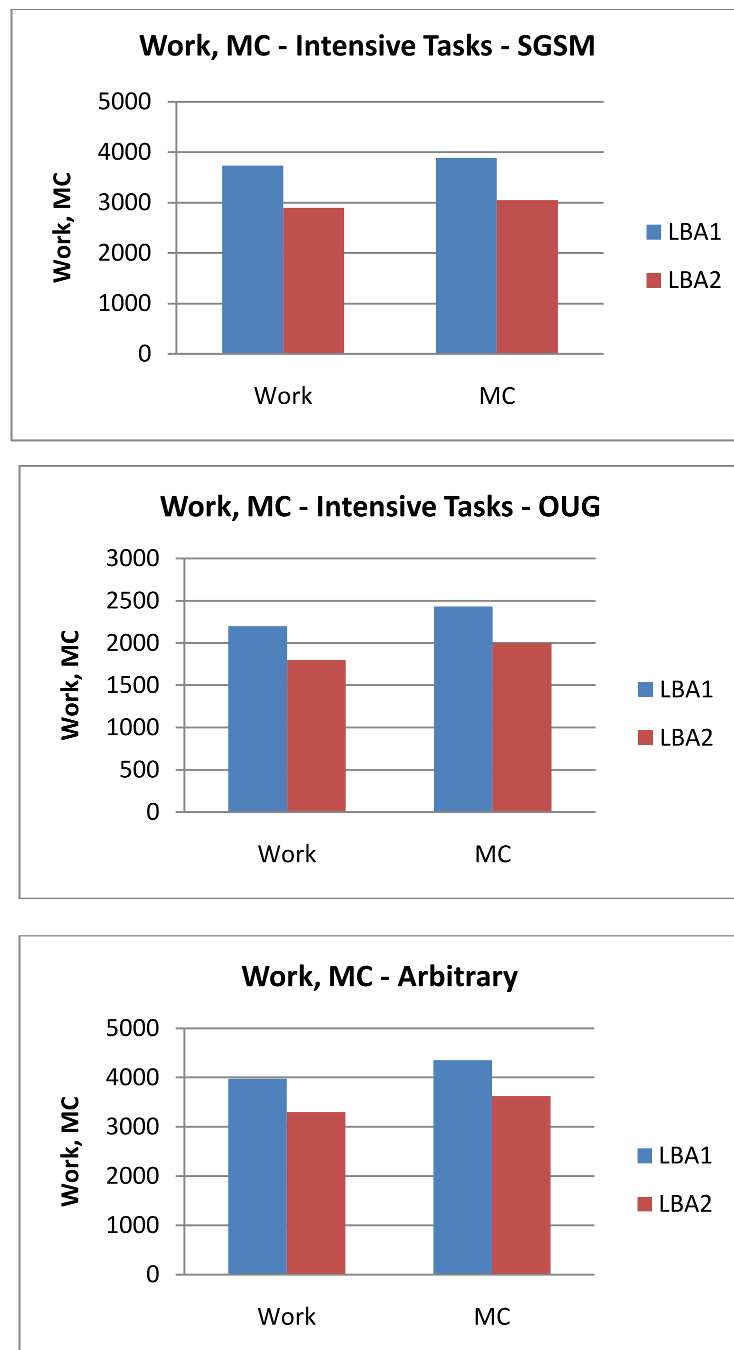


Figure 27: Work, MC - Comparison of LBA1 & LBA2 with Intensive Tasks

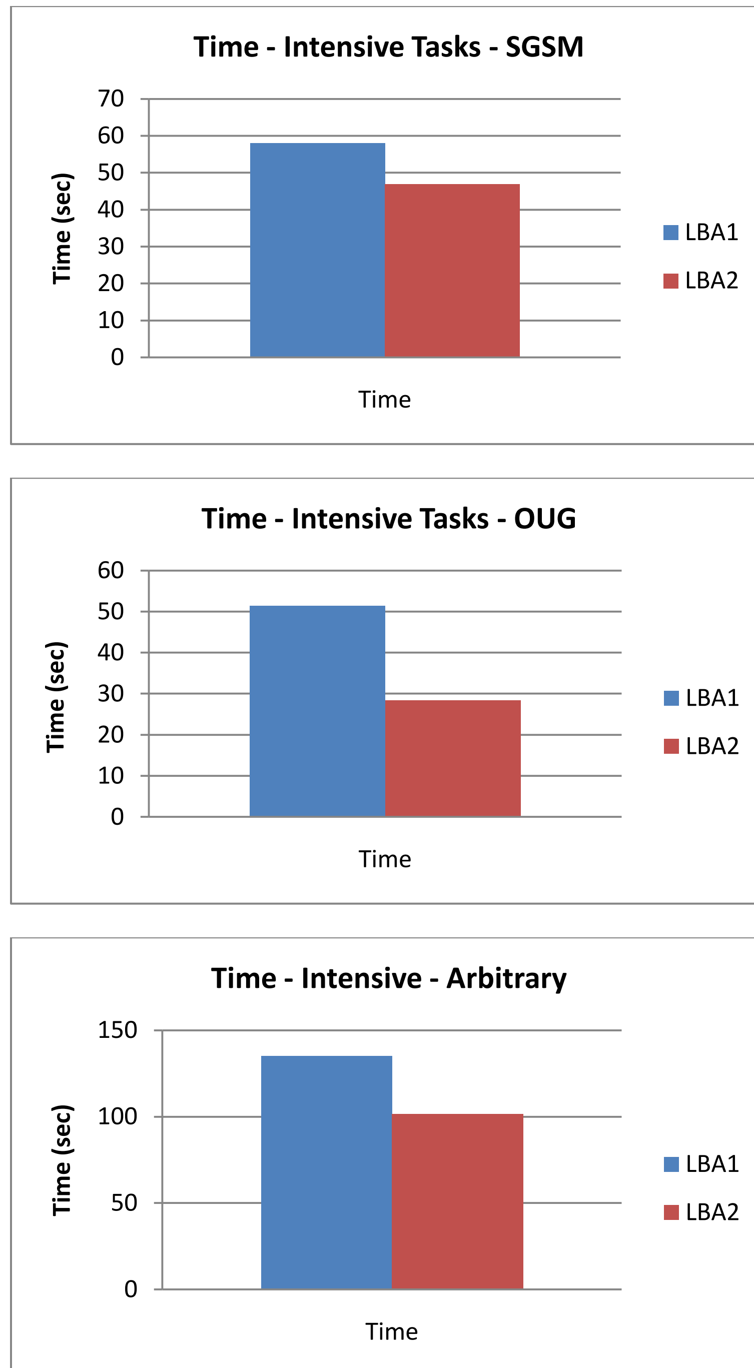


Figure 28: Time - Comparison of LBA1 & LBA2 with Intensive Tasks

6.6.2 Summary

Summarizing our results from this scenario, LBA2 clearly performs better than LBA1. This empirically validates that task scheduling across groups is better than allocating the same tasks in all groups (when there are group merges). The work and message difference is greater for the SGSM pattern rather than OUG, but the opposite is observed for execution time. This is due to the fact that with OUG, fewer regroupings occur when using the LBA2

rule (rather than the LBA1 rule). No change in the number of regroupings is observed for the SGSM pattern.

6.7 Conclusions

In summary of all our experiments we may conclude:

- For all patterns, increasing the processor number increases the communication overheads (e.g., when regroupings occur, the stabilization of a group takes more time during which tasks are executed), thus work, message and execution time are increased. Consequently, we should be careful when picking the number of processors since too many processors in relation to the number of tasks proves inefficient, especially if there are frequently regroupings.
- Increasing the number of tasks causes work, message and execution time to be increased (due to fragmentations), except in SGSM were the load balancing rule assigns tasks in a way that minimizes redundancy across groups (since only merges occur).
- Work, message and execution time are relatively steady for most numbers of concurrent link failures/recoveries, for SGSM and OUG. For SGSM, at small values the processors remain disjoint for longer and at high values more processors attempt to join the same group concurrently introducing further communication overheads. For OUG work, message and execution time are relatively steady due to reduced task execution redundancy provided by the load balancing rule used. For arbitrary regroupings work, message and execution time are increased with increased concurrent groups allowed. Too many concurrent failures make a distributed system inefficient.
- The longer it takes for processors to merge to one group, more redundant tasks are executed, thus work, message and execution time increase. The opposite happens if a group takes longer to fragment. If groups fragment frequently we observe high work and message complexity but execution times are relatively steady due to reduced communication overheads.

- The algorithm performs better when executing with intensive tasks rather than simple tasks in all metrics.
- LBA2 performs better than LBA1, when group merges occur.

As in theoretical analysis, we observe that fragmentations cause execution of more redundant tasks and merges cause overhead in communication. Our experimentation results confirm the theoretical upper bound results. It is demonstrated that task scheduling across groups is indeed better than allocating the same tasks in all groups (when group merges occur). The algorithm has good performance unless many concurrent groups exist or/and regroupings occur frequently, as expected. In realistic situations intensive tasks are used and our experimentation demonstrated that using intensive tasks yields better performance than using simple tasks. Hence, the algorithm seems to have good performance under practical situations.

However, while we expected that merges introduce overhead in communication (regarding the messages sent when regroupings occur) we did not expect that this would cause increase in work as well. As explained, this occurs due to the fact that when a new group is formed it takes some time for the group to be stabilized (e.g., for all indented processors to join the group). During this stabilization phase, processors continue to execute the algorithm and execute tasks (even more in the case of simple tasks), increasing total work. To rectify this, when a regrouping occurs each processor should delay executing the algorithm to give a chance to the new group to take its final form. This should reduce work and message complexity and consequently execution time due to reduction of executed tasks. The reduction of execution time due to less work should overshadow any increase in execution time due to the introduced delay. Additionally, in a singleton group the processor sends the multicast message each time it executes a tasks (as dictated by the algorithm). Since it knows that there are no other processors in the group (from the group communication service) there is no need for this message to be sent.

Chapter 7

Epilogue

Completing a set of tasks using a set of processors in a distributed environment is one of the fundamental problems in distributed computing called DO-ALL. A lot of research is done on the subject and some algorithms are proposed and theoretically analyzed. Group Communication Services form the basis of algorithms proposed for OMNI-DO, a variation of the problem in a partitionable network setting.

In this thesis, one algorithm that solves the OMNI-DO problem is implemented that is a combination of algorithms AX and RS, proposed in previous studies on this subject. The Ensemble System, that provides group communication services, is studied and used in the implementation. The performance measures of interest are work, message complexity and execution time. We have investigated empirically the behaviour of the algorithm in specific and arbitrary regrouping patterns and various parameters such as task and processor numbers, regrouping frequency and quantity of concurrent link failures and recoveries. Additionally, we have conducted experiments for comparison between simple and computationally-intensive tasks, as well as between two load balancing rules.

Overall, the algorithm mostly behaves as expected in the experiments we conducted. When fragmentations occur, the performance is worse due to increased execution of redundant tasks. When merges occur, the number of communicating messages is increased due to regroupings. Additionally, the work is increased due to the stabilization needed when a new group is formed (this is more evident in the cases where many processors try to join the same group at the same time). The introduction of a delay before executing the algorithm, when a new group is formed, should help minimize the communication overheads introduced from merges. The algorithm performs well unless many concurrent groups exist or/and regroupings occur frequently. The load balancing rule (LBA2) implemented in the algorithm

is clearly better than the alternative (LBA1), when merges occur, since task scheduling is done across groups in addition to inner-group scheduling. From our results it can be seen that the algorithm is better for executing intensive tasks rather than simple tasks.

Future work on this subject includes configuring and testing on LAN or WAN networks. In such environments, the algorithm can be tested for robustness with real link and processor failures. Additionally, metrics such as execution time may need to be revisited since in our experiments it was inflated due to the local environment. While Ensemble is one of the popular GCS in the literature, it does not seem to be actively supported by its creators any longer and it is difficult to configure it for modern systems. More recent Group Communication Services with active communities can be used for this purpose (e.g., [27]).

Bibliography

[1] Amir, O., Amir, Y., and Dolev, D. 1993. *A highly available application in the Transis environment*. In Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France.

[2] R. J. Anderson and H. Woll, *Algorithms for the certified write-all problem*, SIAM J. Comput., 26 (1997), pp. 1277-1283.

[3] J. Aspnes, *Competitive analysis of distributed algorithms*, Lecture Notes in Computer Science, Developments from a June 1996 seminar on Online algorithms: the state of the art, Vol. 1442, 1998, pp. 118–146.

[4] Birman, K. and van Renesse, R. 1994. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press.

[5] B. Chlebus, R. De Prisco, and A.A. Shvartsman. *Performing tasks on restartable message-passing processors*. Distributed Computing, 14(1):49–64, 2001.

[6] G. V. Chockler, I. Keidar and R. Vitenberg, *Group Communication Specifications: A Comprehensive Study*, ACM Computing Surveys 33(4), pp. 1-43, December 2001.

[7]D. Dolev and D. Malki, *The Transis Approach to High Availability Cluster Communication*, Communications of the ACM, Vol. 39, no. 4, April 1996, pp. 64-70.

[8] S. Dolev, R. Segala, and A.A. Shvartsman. *Dynamic load balancing with group communication*. In Proceedings of the 6th International Colloquium on Structural Information and Communication Complexity (SIROCCO 1999), pages 111–125, 1999, pp. 111-125.

[9] C. Dwork, J. Halpern, and O. Waarts. *Performing work efficiently in the presence of faults*. SIAM J. Comput., 27 (1998), pp. 1457–1491.

[10] Ch. Georgiou, A. Russel and A. A. Shvartsman. *Work-Competitive Scheduling for Cooperative Computing with Dynamic Groups*, SIAM J. Comput., vol. 34, no. 4, 2005, pp. 848-862.

[11] Ch. Georgiou, A. Russel and A. A. Shvartsman. *The complexity of synchronous iterative Do-All with crashes*, Distributed Computing, 2003.

[12] Ch. Georgiou and A. A. Shvartsman. *Cooperative computing with fragmentable and mergeable groups*, J. Discrete Algorithms, 1 (2003), pp.211-235.

[13] Ch. Georgiou and A. A. Shvartsman, *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*, Springer, 2008.

[14] J. F. Groote, W. H. Hesselink, S. Mauw, and R. Vermeulen, *An algorithm for the asynchronous Write-All problem based on process collision*, Distributed Computing, 14 (2001), pp.75-81.

[15] M. G. Hayden, *The ensemble system*, PhD thesis, Cornell University, January, 1998.

[16] M. Hayden and O. Rodeh, *Ensemble Reference Manual*, February, 2004.

[17] M. Hayden and O. Rodeh, *Ensemble Tutorial*, February, 2004.

[18] Jasonp, *jasonp's Pile of Pi Programs and Peripheral Paraphernalia Page*, 14 Dec 2009, Available at: http://www.boo.net/~jasonp/pipage.html.

[19] P.C. Kanellakis and A.A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, Dordrecht, The Netherlands,1997.

[20] Z. M. Kedem, K. V. Palem, and P. Spirakis, *Efficient robust parallel computations*, in Proceedings of the 22nd ACM Symposium on Theory of Computing (STOC 1990), ACM, New York, 1990, pp. 138-148.

[21] G.G. Malewicz, A. Russell, and A. A. Shvartsman. *Distributed cooperation during the absence of communication*. In Proceedings of the 14th International Symposium on Distributed Computing (DISC 2000), pages 119–133, 2000.

[22] C. Martel and R. Subramonian. *On the complexity of certified Write-All algorithms*, J. Algorithms, 16 (1994), pp. 361-387.

[23] D. Powel, *Group Communication*, Communications of the ACM, Vol. 39, no. 4, April 1996, pp. 50 – 53.

[24] R. De Prisco, A. Mayer, and M. Yung. *Time-optimal message-efficient work performance in the presence of faults.* In Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC 1994), pages 161–172, 1994.

[25] O. Rodeh and D. Dolev, *The Design and Implementation of Lansis/E*, Master thesis, The Hebrew University of Jerusalem, May, 1997.

[26] D. Sleator and R. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202 – 208.

[27] Spread Concepts LLC, *The Spread Toolkit*, 14 Dec 2009, Available at: http://www.spread.org/.

[28] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd and D. Karr, *Building adaptive systems using ensemble*, Software—Practice & Experience, Vol., no. 9, July 1998, pp. 963-979.