12-14-2001

A Distributed Memory Implementation of LOCI

Thomas George

A DISTRIBUTED MEMORY IMPLEMENTATION OF LOCI

By

Thomas George

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computational Engineering
in the College of Engineering

Mississippi State, Mississippi

December 2001

A DISTRIBUTED MEMORY IMPLEMENTATION OF LOCI

By

Thomas George

Approved:

Edward A. Luke Assistant Professor of Computer Science (Major Professor)	Pasquale Cinnella Associate Professor of Aerospace Engineering (Committee Member)
Clarence O. E Burg Assistant Research Professor of Computational Engineering (Committee Member)	Boyd Gatlin Associate Professor of Aerospace Engineering (Graduate Coordinator)
A. Wayne Bennett Dean of the College of Engineering	

Name: Thomas George

Date of Degree: December 14, 2001

Institution: Mississippi State University

Major Field: Computational Engineering

Major Professor: Dr. Edward Allen Luke

Title of Study: A DISTRIBUTED MEMORY IMPLEMENTATION OF LOCI

Pages in Study: 52

Candidate for Degree of Master of Science

Distributed memory systems have gained immense popularity due to their favorable price/performance ratios. This study seeks to reduce the complexities involved in developing parallel applications for distributed memory systems. The Loci system is a coordination framework which was developed to eliminate most of the accidental complexities involved in numerical simulation software development. A distributed memory version of Loci was developed and has been tested and validated using a finite-rate chemically reacting flow solver developed in the sequential Loci framework. The application developed in the original sequential version of Loci was parallelized with minimal changes in its source code. A comparison with the results from the original sequential version guarantees a correct implementation. The performance measurements indicate that an efficient implementation has been achieved.

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to my major professor Dr. Edward Luke for his invaluable advice and assistance during my studies and during my work on this thesis. He has been a good mentor and extremely patient in explaining difficult concepts. His constant encouraging words and ideas have inspired me to exceed my own expectations on this project. I would also like to thank Dr. Pasquale Cinnella and Dr. Clarence Burg for their suggestions and technical advice.

I would also like to thank Dr. Roy Koomullil for providing me with valuable knowledge to make a headstart on this project. If not for his help, this project would have taken a while longer to be completed. In addition I would like to thank Dr. Boyd Gatlin and the Engineering Research Center for providing financial support and facilities to this research effort.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

Identifiers:	
α	A generic variable name
β	A generic mapping name
χ	Property Existential Function
τ	Iteration level name
Logical Operators:	
$\text{ran}(\beta)$	The range of map β
$\text{dom}(\beta)$	The domain of map β
$\wedge$	Logical AND
$\vee$	Logical OR
$\neg$	Logical Negation
$\Rightarrow$	Logical Implication
$\rightarrow$	Mapping (composition) operator
$a \leftarrow b$	Rule operator, meaning b generates a
$\oplus$	A generic operator
$\cup$	Set Union
$\cap$	Set Intersection

CHAPTER I

INTRODUCTION

As the requirements for simulating complex numerical models exceed the physical limits of processor and memory speed, it is becoming more attractive to use multiprocessors to increase computing power. Two kinds of parallel systems have become popular : the tightly coupled shared-memory architecture, and the distributed memory architecture [1]. A tightly coupled multiprocessing system consisting of multiple CPUs and a single global physical memory is more straightforward to program as it is a natural extension of a single CPU system. However, this type of multiprocessor has a serious bottleneck: the main memory is accessed via a common bus - the serialization point. Although this is not a requirement for Non Uniform Memory Access(NUMA) architectures, the shared memory paradigms on NUMA architectures are yet to demonstrate scalable solutions. Distributed memory multiprocessors, however, do not suffer from this drawback. These systems consist of a collection of independent computers connected by a high speed interconnection network. If the network topology is chosen carefully, the system can contain more processors, by a higher order of magnitude, than a tightly coupled system.

In the past, only a limited number of government laboratories and large businesses had access to high performance parallel systems, primarily because of the tremendous cost of these systems. Today, with the rapid growth of microprocessor power, reduction of memory prices, and emergence of high speed networks, parallel systems have become cost effective to build and maintain. The introduction of distributed parallel systems based on groups of networked workstations and personal computers, commonly called clusters of workstations, has made parallel systems accessible to a wide range of users. The largest of these clusters, assembled out of commercial off-the-shelf(COTS) parts, are beginning to compete with offerings from traditional supercomputer vendors(e.g. Beowulf project [2]). Unfortunately, although the hardware costs of putting together a system capable of super-computing performance levels are dropping radically, the software costs required to actually utilize such resources are substantial. The biggest obstacle

to the spread of parallel computing and its benefits in economy and power is the problem of inadequate software. The author of a parallel algorithm for an important computational science problem may find the current software environment obstructing rather than smoothing the path to the use of the cost effective hardware available.

The higher performance platforms have made simulation of increasingly complex scenarios possible. Tedious message passing paradigms are currently the principal method of programming these low cost computing platforms. Despite these barriers, complex simulation software has been developed and successfully deployed to solve real engineering problems using state of the art technology [3]. Significant research has been conducted in developing codes that have portability across serial and parallel architectures and in developing reusable multi-disciplinary components to reduce the time taken in development. Brooks [4] identified two sources of software complexity: essential and accidental. Under this classification, essential complexity is the complexity required to solve a problem, while accidental complexity results due to the implementation requirements. This accidental complexity might cause software "bugs" which increases the validation costs of the parallel implementation.

The present study is an extension of the formal specification approach proposed by Luke [5]. Loci was proposed to eliminate much of the accidental complexity encountered in numerical simulation software development by changing the way software is specified. The Loci specification is fundamentally a concurrent specification, which enables one to develop parallel software automatically. The current implementation of the Loci framework is targeted at shared memory architectures. The Loci system was initially proposed as a coordination framework for CFD applications. The design was motivated by the realization that a significant portion of the complexity and bugs associated with developing large scale multi-disciplinary computational field simulations are caused by incorrect looping structures, improper calling sequences, or incorrect data transfers. The Loci framework addresses these problems by automatically generating the control and data movement operations of an application from component specifications, while guaranteeing a level of consistency between components. For developing applications for distributed memory architectures, the distribution of data among the processors increases coordination difficulties. The automatic verification of consistencies between components and the fine grain of description of the computations makes Loci an ideal candidate for developing applications which can be parallelized automatically. An application program developer need not

worry about any of the nuances involved in parallelizing an application which is already developed in the Loci framework. Previous to this work, Loci provided parallelization of an application on shared memory architectures only.

The development of this thesis begins with an overview (in Chapter II) of the current tools that aid the user in parallel programming. A comparison of these various tools is made with the Loci framework. Chapter III describes the sequential version of the coordination framework Loci. The details of how the coordination system is specified and also the details on translating this specification framework to machine executable form are discussed. Chapter IV gives a comprehensive description on the development issues and the methodology adopted in developing the distributed memory version of the Loci framework. An attempt is made at describing the methodology using the relational approach similar to the one used in the development of the Bernoulli compiler [6]. The results obtained after testing the distributed memory version of Loci using a finite-rate chemistry model are reported in Chapter V. Finally, Chapter VI provides a concluding summary of the work and identifies some key areas for improving the performance and scalability of the current implementation.

CHAPTER II
RELATED WORK

The rapid advancements in the simulation of natural phenomena and complex physical processes have caused the demand for super-computing resources to rise sharply in the past decade. In the past ten years, supercomputer performance has steadily increased more than a hundred fold. This has allowed computational scientists to simulate increasingly more complex models of fluid flow and compute the flow over more complicated geometries. Although parallel machines provide tremendous potential for high performance, their programming can be a tedious task. The famous Parallel Platform Paradox says it all. It states that *the average time required to implement a moderate-sized application on a parallel computer architecture is equivalent to the half-life of the latest parallel supercomputer.* Although this can be disposed off as too pessimistic a view, the reality is that the time spent in code maintenance is substantial. The software development process for parallel processing includes design of a parallel algorithm, partitioning of the data and control, communication, synchronization, scheduling, mapping and identification and interpretation of the various performance measures. While an efficient implementation of some of these tasks can only be done manually, a number of tedious and error prone chores such as scheduling, mapping and communication can be automated [7]. Several research efforts have demonstrated the usefulness of program development tools for parallel processing. This chapter discusses some of the current technology that is presently being used to address these issues.

2.1 Parallel Compilers

2.1.1 FORTRAN D Compiler

Fortran D is an enhanced version of Fortran that introduces data decomposition specifications. If a program is written in a data parallel programming style with reasonable data decompositions, it can be implemented efficiently on a variety of parallel architectures. The compiler technology

developed is for Multiple Instruction Multiple Data(MIMD) distributed memory machines. The compiler aims at automating the task of deriving node programs based on the data decomposition in a way that reduces the communication and load imbalance. The current prototype of the compiler performs message vectorization, collective communication and fine grained pipelining. The code generation strategy is based on the concept of *data dependence* [8].

2.1.2 Bernoulli Sparse Compiler

The Bernoulli sparse compiler was developed to demonstrate the effectiveness of the *relational* [9] approach to sparse matrix code compilation. In this approach, the user writes programs as if the matrices are dense, and then provides a specification and formats of the matrices which are actually sparse. The compiler generates the parallel sparse Single Program Multiple Data (SPMD) code for the given sequential dense matrix program provided, descriptions of sparse matrix formats, and distribution formats of data and computations are given. The approach is based on viewing parallel $DOANY$ loop execution as relational query evaluation, and sparse matrices as a distributed relation. For execution on distributed memory systems, the loop nests are viewed as distributed queries and the process of generating SPMD node programs is viewed as the translation of distributed queries into equivalent local queries and communication statements. The properties of the distributed relation are exploited in order to gain high performance.

2.2 Parallel Libraries

One approach to reducing the complexity of parallel applications is by developing libraries that implement the most complex components of common application primitives in parallel. The development of parallel libraries encapsulate and hide the complexities from the user. Most of these libraries aid the application program developer in obtaining parallelism for the application at a coarser level. This section discusses two libraries which have demonstrated encouraging results.

2.2.1 CHAOS/PARTI

The CHAOS/PARTI [10] runtime support library is a set of software primitives that are designed to handle irregular problems on distributed memory architectures. The primitives

are designed to ease the implementation of computational problems on parallel architecture machines by relieving users of the low-level machine specific issues. The original sequential code is mostly unaltered, with the exception of the introduction of various calls to the CHAOS/PARTI primitives, which are embedded in the code at appropriate locations. The CHAOS/PARTI library provides primitives that couple partitioners to application programs, remap data, and partition work to processors, thus optimizing interprocessor communications. The communication patterns are captured at runtime, and appropriate send and receive messages are automatically generated. In addition to being used for hand-parallelization codes, the CHAOS/PARTI library has also been integrated with the Fortran 90D compiler being developed at Syracuse University.

2.2.2 POOMA

The POOMA [11] library is a collection of templated C++ classes for writing parallel PDE solvers using finite-difference and particle methods. POOMA programs are written at a very high level, using data-parallel array expression in the style of High Performance Fortran or serially using iterators on each CPU. In many ways, this library provides many of the features that were previously only available through specialized compilers. Since POOMA uses C++ templates, it offers a generic data-parallel interface that can work with built-in as well as complex data types. POOMA II is an enhanced version which can handle unstructured and adaptively refined meshes. An application developer using the POOMA framework can write parallel PDE solvers without having to worry about the low-level details of layout, data transfer and synchronization.

2.3 Domain Specific Languages

The primary function of programming languages and tools has always been to make the programmer more productive. A domain specific language(DSL) can be viewed as a programming language dedicated to a particular domain or problem. The development of domain-specific languages has reduced the complexity of building scientific applications. These languages provide special constructs that facilitate more economical expression of numerical algorithms. In most cases, the additional semantic content in the domain-specific language can be used to provide optimizations that would be difficult to perform in a more general case.

2.3.1 Strand/PCN

Strand [12] is the first commercially available concurrent logic programming language. It is mainly a single assignment logic-programming-based language that is designed for the implementation of parallel applications. A Strand program consists of a set of procedures defined by rules. Communication between rules is provided by single assignment variables. Constructs are provided in the language to accomplish the mapping of processes to processors. The mapping constructs do not affect the program correctness, rather they affect the program performance.

Program Composition Notation(PCN) [13] is a superset of Strand. The focus of PCN approach to parallel programming is the development of programs by the *parallel composition* of simpler components, such that the resulting programs preserve the properties of the components that they compose. Due to the high-level nature of the PCN language, a highly sophisticated compiler is required to achieve efficient execution on sequential and parallel computers. The PCN compiler implements both the PCN language and the constructs introduced to support the reuse of parallel code.

2.3.2 Formal Specification Approach

Birken [14] [15] proposed an approach to reduce the complexity of parallel computations for numerical PDE problems. In this approach, all the application-dependent knowledge is supplied by the developer of the sequential program via a formal specification, using first-order predicate calculus with quantors. The component specifications given by the application developer are linked on their interface level by linking the semantics of their relations via some additional rules. Using these specifications and an automatic theorem prover, the space of all programs is heuristically searched to find a program that satisfies the specification. However, the process of generating consistent graphs by the automatic deduction system is not efficient to be used at run time. Hence, the formalization techniques are applied to automatically generate efficient source-code based on the popular techniques in Artificial Intelligence.

2.4 Summary

Significant work has been done to relieve the programmer of a parallel application from machine-dependent and architecture-specific tasks. Even after demonstrating encouraging

results, the fact still remains that these compilers/libraries/languages are seldom used by everyone. All the technologies covered in this chapter are specific to numerical simulation applications. Of these, only the domain specific languages seriously addresses the problem of accidental software complexity. The Loci system provides a guarantee concerning the logical consistency of the derived simulation [16] while providing an efficient implementation. This consistency guarantee gives it an edge over other implementations of a similar nature. A distributed memory implementation of Loci is aimed at enabling the current Loci framework to solve larger problems, and also to develop multi-disciplinary simulations in a unified framework.

CHAPTER III

LOCI: THE SEQUENTIAL VERSION

The Loci system is an application framework that seeks to reduce the complexity of assembling large-scale finite-difference or finite-element applications. It could also be applied to many algorithms that are described with respect to a connectivity network or a graph. In the Loci system, computational graphs are represented by collections of entities and collections of maps or connectivity lists. Before we describe the distributed memory implementation it is necessary to describe some of the terms and notations being used in this framework. This chapter gives a description of the various data structures used, the notations used for representing the rules, the different types of rules available and also the method by which Loci guarantees the consistency of components while coordinating them.

3.1 Definitions

Definition 3.1 A store *defines a bijective mapping from entities to values. Thus for a variable identified as* α *a store is defined by the set* $\alpha = \{(i, v_i)|i \in dom(\alpha)\}$. *A store is basically a construct for providing entity labels to values.*

Definition 3.2 A map *defines relationships between entities. It is represented by a set of ordered pairs such that the map* β *is represented by the set* $\beta = \{(i, j)|i \in dom(\beta), j \in ran(\beta)\}$. *The inverse of map* β *is given by* $\beta^{-1} = \{(j, i)|(i, j) \in \beta\}$.

Definition 3.3 A constraint *defines an identity mapping of entities. Thus constraint* β *is defined by* $\beta = \{(i, i)|i \in \mathcal{C}\}$ *where* $\mathcal{C}$ *is the set of entities in the constraint.*

Definition 3.4 A constraint qualifier *is of the form* $CONSTRAINT(clist)$ *where clist consists of a comma-separated list of variable accessors. The constraint set is formed from the intersection of the domains of these variable accessors.*

Definition 3.5 A parameter *defines a singleton, or mapping from a set of entities to a single value. Thus, for a variable identified as α, a parameter mapping is identified by the set $\alpha = \{(i, v)|i \in dom(\alpha)\}$.*

Definition 3.6 A Variable Identifier *consists of a variable name, α, iteration identifier, τ, and offset, $\theta \leq 1$ The variable identifier is represented by the notation $\alpha\{\tau + \theta\}$ for relative offsets and $\alpha\{\tau = \theta\}$ for absolute offsets.*

Definition 3.7 The mapping operator, *denoted as $\rightarrow$, is defined by the set $\beta \rightarrow \alpha = \{(i, \alpha_j)|(i, j) \in \beta, (j, \alpha_j) \in \alpha\}$. Mapping operators can be chained such as $\beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \alpha$.*

3.2 The Loci Data Model

The most fundamental concept in the Loci system is the concept of an entity. Entities are conceptually places where values can be stored. Values are bound to entities via the `store` construct which provides an injective mapping from entities to values. The `parameter` construct provides a singleton interface to a value where a set of entities is mapped to a single value. Relationships between entities are provided by the `map` construct. The `constraint` construct, used to constrain the computations to some subset of entities, provides an identity mapping over a given subset of entities. These constructs are illustrated in figure 3.1 .

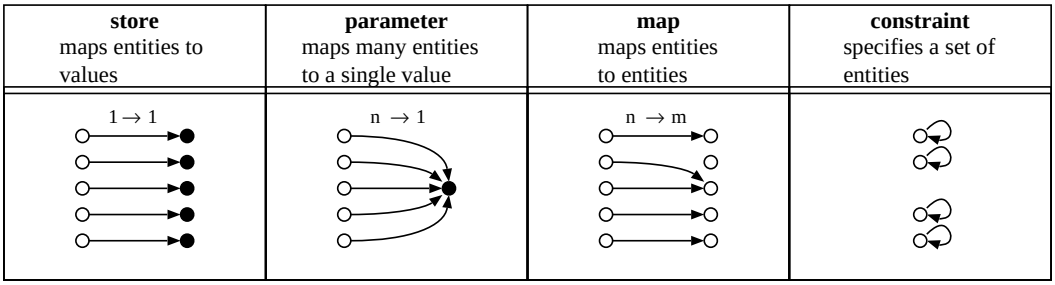

Figure 3.1: Four Basic Database Constructs

The database is the fundamental starting point for logic programming systems. The four basic constructs shown in figure 3.1 are used to formulate a database of facts that describes the problem. Thus the database becomes a center of communication for programs derived from the specifications. Each of the facts in the database is given an identifier that consists of a name,

an iteration label and an iteration offset. The fact database provides an association of variable identifiers to facts about entities. In general, these facts consist of associations of entities with either values or other entities.

3.3 Rule Semantics

In addition to a database of facts that includes the problem specification, a database of rules describes transformations that can be used to introduce new facts into the database. These rules correspond to fundamental computations involved in solution algorithms such as rules for evaluating areas of faces, or for solving equations of state. These rules are specified using text strings called `rule signatures` .

Definition 3.8 A rule signature *is defined by a head and body written as "$Head \leftarrow Body$". The head consists of a variable accessor, while the body consists of a list of variable accessors and rule qualifiers. Variable accessors are represented as $\beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \beta_N \rightarrow \alpha$ where $N \geq 0$, β_i are variable identifiers of maps, and α is a variable identifier for the accessed store.*

Definition 3.9 A rule signature is compatible *when all of the variable identifiers represented in the head and the body have compatible iteration identifiers. A compatible rule signature has a head time, represented by τ_h, which is the least upper bound of all iteration identifiers in the head, and a body time, represented by τ_b, which is the least upper bound of all iteration identifiers listed in the body.*

For example, the rule signature $p \leftarrow rho, T, R$ represents that a value for pressure(p) is provided when values for density(rho), temperature(T) and gas constant(R) are present. A rule signature may also contain the mapping operator "$\rightarrow$" to represent the use of indirection in a computation. For example, the rule signature of a computation that generates areas of faces utilizing positions of related nodes may be given as $area \leftarrow face2nodes \rightarrow position$, where "$face2nodes$" is a mapping that connects faces to their defining nodes. It states that for all entities that satisfy all the properties given in the body, a computation can be performed to produce a value given in the head. There are several classes of computations that are encountered in a typical unstructured grid computation. This section will identify the semantics of these computation rule classes.

3.3.1 Rule Constraints

The context of a rule is defined as the intersection of domains of all variable accessors and constraints in the rule specification. Any rule that is specified can be constrained to compute values for some subset of entities. For example, if one needs to specify the value of nodes where the Dirichlet boundary condition is applied, this might be specified by the rule signature: $u \leftarrow CONSTRAINT(Dirichlet)$. In addition to constraining the rule applications, constraints also provide assertion semantics: a constraint implies that a rule must provide values for every entity in the constraint. In many cases this can be used to automatically detect inconsistencies caused by incomplete specifications. In addition, Loci identifies over-specification by requiring that each entity may have only one given value. Thus, if a boundary condition is applied to an interior node of the domain, there is a conflict between the boundary conditions and the stencil specifications and thus an error would result.

3.3.2 Point-wise Rule Semantics

The most common rule in finite difference or finite element applications is the point-wise rule. The point-wise rules represents an entity by entity computation of values that are placed in the stores listed in its head. The body of the rule specification describes how information is brought to the site of computations. The semantics of the point-wise rule application requires that an output variable can define only one value per entity. If an ambiguous specification produces two rules that compute values for the same entity, it is flagged as an error during scheduling. Point-wise rules include recursive specifications where the same variable name appears both in the head and the body of the rule. Thus, recursion in point-wise rules is bound to the number of entities in the database as long as the single value per entity rule is not violated.

3.3.3 Reduction Rule Semantics

A typical reduction operation consists of a `unit` rule, an `apply` rule and a `join` operation listed in table 3.1. The process of generating the reduction begins with the unit rule. The unit rule initializes a reduction variable to the identity element, the apply rule "accumulates" results of a function application to a set of values, and the join operation "accumulates" the partial results. Typically the unit rule has a constraint qualifier that limits the entities to be included in

the computation. Once a unit rule has been applied to a set of entities, all apply rules are applied to the unit rule initialized entities. Recursion is not allowed in reduction rule specifications for either the unit or apply rules.

A reduction operation can be defined by any function that can be decomposed according to the Bird-Meertens formalism [17]. In this abstraction, a reduction is described by a function composed of three components: a function that is applied to a set of values `f`, an associative and commutative operator $\oplus$ that is defined on the type returned by the above mentioned function, and an identity element of operator $\oplus$, `e`. Thus a reduction, r, over values, $\{v_i | i \in [1, N]\}$, using function f and operator $\oplus$ is defined as

$$r = f(v_1) \oplus f(v_2) \oplus \cdots \oplus f(v_i) \oplus \cdots \oplus f(v_N). \tag{3.1}$$

When the reduction is evaluated using a left or right precedence rule, then a sequential evaluation is derived; however, the associative property of $\oplus$ allows for different parallel evaluation orders. For example, the set of values can be partitioned into subsets that can be evaluated concurrently as in

$$r = \{e \oplus f(v_1) \oplus f(v_2) \cdots \oplus f(v_p)\} \oplus \{e \oplus f(v_{p+1}) \oplus \cdots \oplus f(v_N)\}, \tag{3.2}$$

where the identity element is used to initialize each subset.

Table 3.1: Basic Reduction Steps

Rule Type	Function	Rule Signature
Unit Rule	$r_i^0 = e$	$r \leftarrow CONSTRAINT(v), UNIT(e)$
Apply Rule	$r_i^{j+1} = r_i^j \oplus f(v_i)$	$r \leftarrow r, v, APPLY(\oplus)$
Join Op	$r_i^{m+n} = r_i^m \oplus r_i^n$	Derived (No signature)

3.3.4 Iteration Rule Semantics

Iteration is defined by using three types of rule specifications: `build` rules that construct the iteration, `advance` rules that advance the iteration, and `collapse` rules that terminate the iteration. The rule specifications collectively follow an analogy to the inductive proof, in which build rules are analogous to an inductive base, while advance rules are analogous to an inductive

hypothesis. For example, the description of an iteration, where a variable named `q` is iterated to a converged solution, may be specified by the following three rules. A build rule of the form `q{n=0}<-ic`, while an advance rule might look like, `q{n+1}<-q{n},dq{n}`, and finally the iteration of `q` might be terminated by the collapse rule `solution<-q{n},CONDITION(converged{n})`.

The iteration in the above example proceeds by initializing the first iteration, `q{n=0}`, using the build rule. Next, termination of iteration is then checked by computing `converged`. If the test succeeds then the collapse rule terminates the iteration, otherwise, the iteration advances in time by the repeated application of the advance rule. The advance rule computes values for the variable `q` for the next iteration level `{n+1}` given the current iteration values at time level `{n}`.

3.3.5 Stationary Rules and Time Promotion

To support iteration, variables that exist in lower levels of the iteration hierarchy are automatically promoted up the iteration hierarchy. Thus a variable that is computed in iteration `{n}` is communicated to iteration `{n,it}` automatically. In addition, the rules that are completely specified at the stationary level will be promoted to any level of the hierarchy. This allows for the specification of iteration independent relations.

3.4 Execution Schedule Generation

The purpose of defining a specification language, for describing numerical solution methods, is to facilitate economical and correct implementations of numerical simulation programs. The process of converting the specification language into a numerical simulation program requires scheduling of rule subroutines. An execution schedule for rule subroutines is developed through a three step process. The first step constructs a variable dependency graph. This dependency graph indicates the connection between the variables that are implied by the rule database. The graph is then processed to remove the recursive components and a Directed Acyclic Graph(DAG) is created. The second step, identified as the existential analysis phase, computes the attributes that are defined for each entity. Finally, a pruning stage computes the entities that are required for the computation. The results of the pruning stage are then used to generate an execution schedule.

3.4.1 The Dependency Graph

The dependency graph is a directed graph that describes the flow of data through rule specifications. In this graph both rules and variables(attribute names) are vertices while data dependencies are indicated by graph edges. A dependency graph is illustrated in figure 3.2. The edges connect variables to rules and rules to variables, whereas rule to rule or variable to variable connection does not exist. The dependency graph is generated using rule signatures. For each variable in the body of a rule there is an edge from that variable vertex to the rule vertex and for each variable in the head of a rule there is an edge from that rule vertex to the head variable. For rules stored in the database, the dependency graph is created as described above. In addition to rules in the database there are some "derived " rules which also need to be placed in the dependency graph before it can be used to generate the execution schedule. These derived rules include the promotion of stationary rules and the generation of PROMOTE and GENERALIZE rules. Once these rules have been added to the graph, the dependency graph represents a data flow graph of all possible programs that may be derived from the rule database.

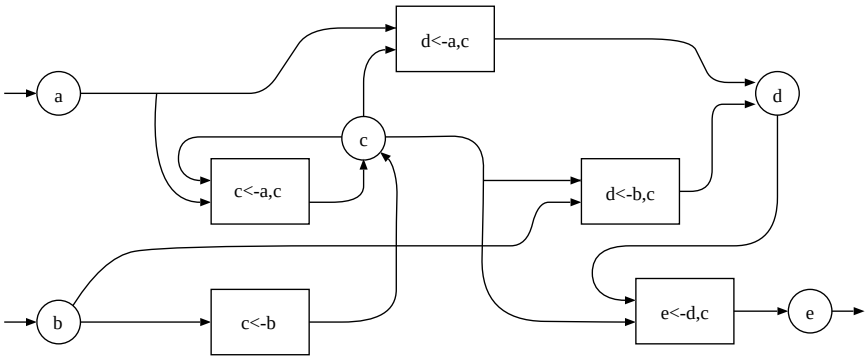

Figure 3.2: An Example Data Dependency Graph

3.4.2 Graph Processing

The first step in generating an execution schedule is determining the order in which the rules must be satisfied. When the dependency graph is a Directed Acyclic Graph(DAG), this execution order can be determined using a topological sort [18]. Unfortunately the graph is not guaranteed to be a DAG, since rules may include recursive specifications. In the process of analyzing the dependency graph, it is sometimes useful to remove a collection of rules from the graph so that

different scheduling policies can be applied to them. In order to convert the dependency graph into a DAG, it is necessary to remove the recursive loops so that they can be treated separately. This is done by noting that recursion produces strongly connected components in the dependency graph. The strongly connected components of a graph can be identified efficiently using common graph algorithms. A straightforward partitioning of the graph is done by, removing the local variables of the partition along with the rules from the graph, and inserting a new node(known as a pseudo rule) that represents the external dependencies of the removed nodes. Unfortunately, a straightforward partitioning based on the rules in a strongly connected component set does not remove the recursive loop from the graph. In order to remove this loop we need to alias all variables of the strongly connected component to input and output versions, and then move all the edges that connect to the component variables to their input aliases, and all the edges that leave the component variables to their output aliases [16]. Now the component variables will be local to the partition and the pseudo rule, that represents the component, will not be recursive.

If there are iterations involved, a simplification of the dependency graph can be accomplished by partitioning it based on iteration levels.This is done by sorting the rules based on the maximum of the head and body times(τ_h and τ_b). If the partitioning proceeds from the leaf nodes of the iteration hierarchy to stationary time, then the partitioning will form a hierarchy of graphs that mimics the iteration hierarchy. Once this partitioning is performed each level of iteration is described by an independent dependency graph. Thus, computations that are performed by an iteration will appear as a pseudo-rule in the dependency graphs that uses these results, hiding many of the particular details involved with analysis of iterations.

At this juncture, the dependency graph has been generated. It has been sorted such that it forms a hierarchy of DAGs. This hierarchy of DAGs is composed of three principal types of pseudo-rules.

1) Pseudo-rules representing recursion over entities.

2) Pseudo-rules representing the creation of iteration.

3) Pseudo-rules that define the iteration loop itself.

Also, since the analysis done so far depends only on the rule database, these computations can be performed just once, while assembling the rule database, rather than at every simulation run.

3.5 Existential Deduction

The second phase in the generation of the schedule is the existential deduction phase which determines "what" attributes can be assigned to "which" entities. For example, the rule $p \leftarrow rho, R, T$, specifies that, entities that have attributes rho, R and T, will also have the attribute p. The existential deduction begins with the given facts and follows the DAG order computing the entities associated with each attribute until the goal is reached. The existential deduction for point-wise rules involves the entity by entity application of modus ponens. The point-wise rule specifies attributes at the intersection of the arguments in its body. For singleton rules the existential deduction proceeds in a similar fashion as the point-wise rule. Since apply rules are invoked only for the entities that have been initialized to the unit value, apply rules cannot create attributes. Therefore, apply rules can be neglected and only the unit rules need to be considered for the existential analysis phase. During the existential deduction, recursive loops are iteratively evaluated until all possible attributes are generated.

3.6 Pruning and Schedule Generation

The result of the existential deduction is a concurrent schedule that obtains the requested goal. The schedule obtained in this fashion will compute all possible attributes that can be computed from the attributes specified in the fact database. Hence, as an attempt to optimize, we do a pruning phase to compute just the values that are needed to provide the requested goal. The pruned schedule is obtained by transposing the head and the bodies of rule specifications. Once the set of transposed rules is generated, the pruning is performed by calculating an existential deduction for the transposed rule database given the existential information derived for the requested variables.

The scheduling process naturally produces a concurrent schedule. Only partitioning of entities to processors is needed to generate a schedule for parallel processors in a shared memory architecture.

3.7 Summary

This chapter, thus presents an overview of the Loci framework. A description of the different kinds of data structures and rules, along with a description of the different steps in the schedule generation process, is provided.

CHAPTER IV

LOCI: DISTRIBUTED MEMORY VERSION

This chapter gives a comprehensive description of the implementation details of the distributed memory version. The guidelines followed throughout the development of the distributed memory version of Loci, are as follows.

1. A generalized code is to be developed which can be applied to any graph based problems.
2. Applications already implemented in the Loci framework should automatically parallelize for a distributed memory system incurring minimal changes in its source code.
3. The distributed memory version should work without any glitches with the current thread implementation so that the utilization of the available computational resources can be maximized.
4. The application should be able to solve very large problems with a fair measure of speedup.

The various sections described in this chapter are in the chronological order of the development.

4.1 Fact Database Decomposition

The first step to developing a distributed memory version of Loci is the decomposition of the fact database among the processors. A fact database is set up on all the processors using the basic constructs, namely `store`, `parameter`, `map` and `constraint` available in the Loci data model. The class hierarchy of the data structures, along with the different methods available to perform operations on them, are shown in figure 4.1. StoreRep is the abstract base class which defines the interface of the derived classes. MapRepI, MultiMapRepI, MapVecRepI ConstraintRepI, StoreRepI, ParamRepI, StoreVecRepI and MultiStoreRepI are concrete object types. All the map containers as well as the constraintRepI store values of integer type while the ParamRepI, the

StoreRepI, the StoreVecRepI and the MultiStoreRepI are class templates. The data type of the values contained in the different stores are determined at compile time. StoreVec and MultiStore are dynamic containers as their size is set at run time. There are a number of methods associated with these data structures for the sequential implementation. While developing the distributed memory version, more methods were added to the existing ones to facilitate the reordering of the fact database, storing of information for communication in the fact database and grouping of data into a contiguous block for communication purposes.

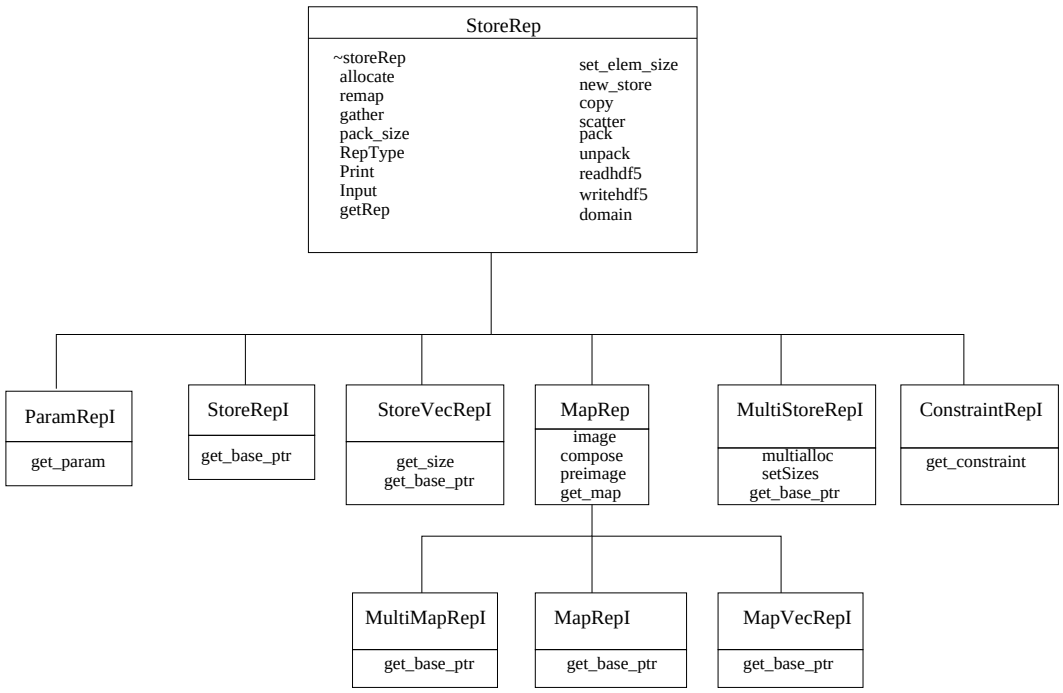

Figure 4.1: Class hierarchy in the Loci system

Just as in most logic programming system paradigms, the database is the fundamental starting point for the Loci system. Hence, for the distributed memory implementation, our natural choice of starting point is providing a distributed fact database. As a first implementation step we create a copy of the fact database on all the processors. We then partition the fact database into as many parts as there are processors. The various steps involved in the development of the distributed memory version is described in the succeeding sections.

4.1.1 Generation of a unified connectivity graph

We start with a serial fact database on all the processors which contains all the facts and the rules according to the problem specification. For example, a set of facts for an unstructured solver might contain information about the nodes, faces and cells in a grid and the relationships between them. To obtain a partitioning of the fact database we need to first create a connectivity graph.

There are a number of ways in which this connectivity graph can be created. The mesh could be converted into a nodal graph(i.e each node of the mesh becomes a vertex of the graph) or a dual graph(i.e each element becomes a vertex of the graph). If we are to follow the traditional ways of domain decomposition, then we need to have a different implementation for each kind of problem. One drawback with the above mentioned approach is that the application needs to have a separate routine for creating the graph. If we are to design the implementation with the motive that the changes made to the application should be a minimum, then this approach cannot be taken.

The approach we followed is slightly different in that all the processors create a serial fact database. In the process of schedule generation, we form a dependency graph from the fact database. This graph could be utilized to construct a connectivity graph for domain decomposition. As a first step, we extract all the entities associated with the stores and maps in the fact database. In this approach, an assumption that we are making is with respect to the temporal aspect of the dependencies in the graph. When we create a directed acyclic graph for generating a schedule, we get an order in which the rules must be executed. When we loop over all the rules in the fact database and create a graph by performing a union on all the maps, we are assuming that all the rules are executed at the same time. By doing so we hope to minimize the overall volume of communication. But this approach could lead to a case of computation imbalance as the different parts of the graph are computed at different stages. Ideally we must have a separate partition for every stage of the dependency graph so that the computations are well balanced. But in our case the entities are divided such that at a particular stage there might be more computations being done at a particular processor.

Since we take all the maps in the database for forming the graph there might be an over-specification of the connectivity information. Right now there is no way to isolate the necessary

connectivity information as it is problem specific. In order to have a general method to create the graph we need to take into consideration all the relations in the database. This approach guarantees the decoupling of the distributed memory version from problem specific domain decompositions. However, the user may provide a specific decomposition strategy if the generic approach provides insufficient performance.

4.1.2 Graph Partitioning

METIS [19] was used for partitioning the graph. METIS was chosen because it was known to produce high quality partitions extremely fast. An adjacency list representation of the graph was created and a Compressed Storage format(CSR) of it was given as input to the METIS_PartGraphKway routine. The METIS_PartGraphKway routine produces a partition such that the total communication volume, the amount of data that needs to be communicated between processors and the number of messages that needs to be sent or received is a minimum. This routine returns an array which lists the processor numbers associated with each entity in the graph. We then group these entities such that each processor has a list of the entities that it owns locally. The maps in the fact database might require some additional entities to be present in a particular processor, in addition to the entities that are locally owned. These entities are also added to the list of entities accessed by a processor.

This approach is not scalable with respect to memory usage due to the high density of the graph which implies that there exists too many edges due to over specification of connectivity information. This leads to a large number of edges being cut while partitioning. When METIS was run as a separate program the memory usage for the entire run was found to be much less. The large number of edges being cut can be attributed to the fact that we are neglecting the temporal aspect of dependencies and also to the fact that we are taking into consideration all the maps available in the fact database. The approach presently taken is not a long term solution to the domain decomposition problem. From our observations we have found out that, in order to have an application which is scalable with respect to memory, it is better to start the application from a distributed fact database.

In short, we have the flexibility to do a generalized partition of the fact database which is expensive with respect to memory usage or we can give it any decomposition which is optimal to the problem under consideration.

4.1.3 Local Numbering and Clone Region Identification

At this stage, each processor needs to store only the entities that are owned by it and the entities that it accesses using the maps in the fact database. Since there might be maps whose range may contain entities that are not owned by a processor, we might need values for entities that we do not own. Hence, we need to determine what entities each processor accesses. The entities which are needed by a processor, but are not owned by it, are termed as "clone entities".

Since maps gives the relationship between entities, our initial step to identifying the clone region is to loop over the maps and find out the entities that are not owned by a processor locally, but are needed by it to compute the rules in the rule database. To do this, first, we find out all the variables associated with the mappings in the head, body and the constraints of the rules. We then take the image of all those maps for the entities that are owned locally by a particular processor. If any of the entities obtained after applying the maps does not belong to the local entities owned by a processor, it is identified as its clone entity and the processor it belongs to is also noted. Since we had created the fact database and partitioned the graph concurrently on all processors, we can construct the data-structures for the clone regions locally. If we were to do the partitioning on a single processor, we would have had to communicate the necessary information to the other processors.

Once the clone region entities are identified, we assign a local numbering to the entities. Local numbering has the advantage that the entities are labeled in contiguous segments. Since the union and intersection operations on entity sets are of the order of the number of intervals [18], this improves the efficiency of scheduling in Loci. As the number of intervals is typically small and independent of problem size, most of these intersection and union operations can be performed in constant time. A local numbering can be assigned to the entities in a variety of ways. One approach is to give a local numbering to the entities based on the increasing order of global numbering. This approach has a disadvantage that the clone entities get numbered in between the locally owned entities. This causes the entities which are locally owned to be non contiguous and therefore the algorithms which take advantage of the contiguity of the entities are no longer optimal.

In the current implementation, entities are grouped into categories. Since most of the computations happen locally, we want to keep the locally owned entities of like kind as contiguous

as possible. First we give a numbering to the locally owned entities of one kind. Then we number the clone entities of the same kind in the increasing order of processor numbers(which own those entities). Then we move on to the next interval in the category and assign the numbering as the previous case. Finally, we end up labeling the entities in contiguous segments such that entities of the like kind are grouped together.

4.1.4 Additional Data Structures for the Fact database

In the process of identifying the clone region, we come to know of certain information which is needed for communication purposes. We need to store this information in the fact database. For this reason a C++ structure `distribute_info` was added to the fact database. The `distribute_info` data structure stores the following information in the fact database.

1. A vector of structures which contains the processor ID from which the clone entities are to be received and also an entity set containing those entities. The entities are stored in their local numbering.

2. A vector of structures which contains the processor ID to which the entities owned locally are to be sent so as to fill its clone region and also an entity set containing those entities(in local numbering).

3. A map which gives the global to local numbering of all the entities owned locally and also for those entities in the clone region.

In addition to this data structure, we store the local entities as an additional fact in the fact database. A method `isDistributed()` was added to the fact database which returned true if the fact database was decomposed. A map giving the local to global mapping of entities was also added to the fact database.

4.1.5 Re-ordering of the fact database

This is the final step in the fact database decomposition. Before discarding the unwanted information from the fact database some operations need to be done. Once all the necessary information is added to the fact database, depending on the global to local mapping stored in the `distribute_info` data structure, the fact database is reordered.

For reordering the fact database a method `remap` was written which takes in the mapping from global to the local numbering and changes the domain of the store or map to the local numbering. This enables us to keep the entities numbered consecutively. An example of a `remap` operation for a `multiStore` is shown in figure 4.2. The `remap` method for the stores makes use of another method called `scatter` which assigns the values for the entities pointed to by the map passed to it.

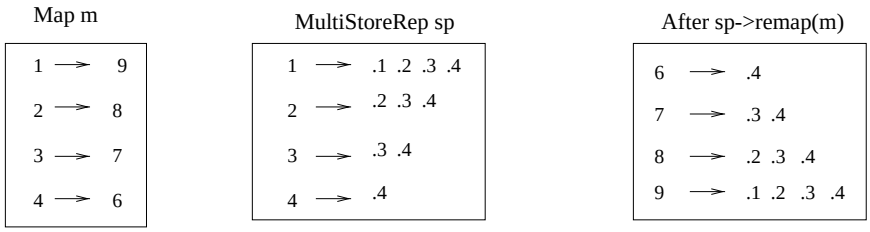

Figure 4.2: Remap method applied to a mutiStore

Maps need to be treated a bit differently. Since maps gives the relations between entities in the fact database, once the entities are re-numbered, the domain and range of the maps should also change to the local numbering. The `compose` method was written to collapse two maps into a single map. This is possible only if the range of one of the maps has elements that are in the domain of the other map. An example of the `compose` operation is shown in figure 4.3. The `remap` operation for the maps make use of the `compose` method and the `scatter` method. First the `scatter` method is called to make the domain of the map in local numbering. Then the `compose` method is called so as to collapse the two maps so that the final map contains both the domain and range in local numbering. The `remap` operation for a `multiMap` is shown in figure 4.4.

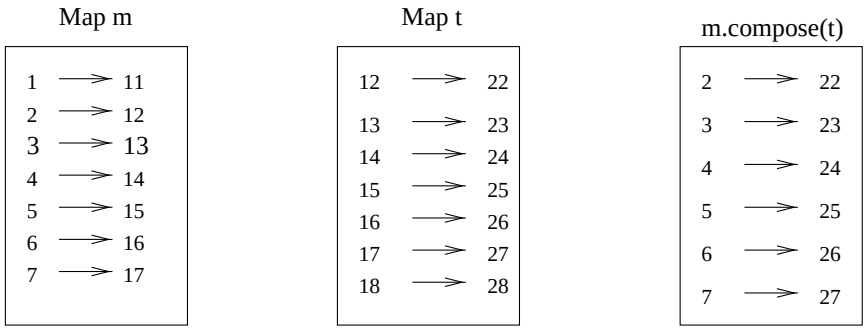

Figure 4.3: Compose method applied to a map

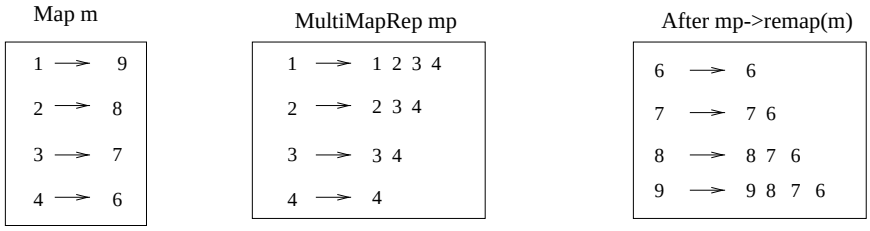

Figure 4.4: Remap method applied to a mutiMap

The reorder routine calls the `remap` method for each of the stores and maps in the fact database. Now the fact database contains only those entities that are in the clone region and those that are owned locally by that processor. All other information regarding the entities stored in other processors is discarded.

4.2 Dependency Graph Analysis

The dependency graph analysis proceeds as in the serial case (Chapter III). The recursive dependencies are removed and the dependency graph is converted to a DAG. Since the entities are distributed among processors and there are relations between entities that are not owned locally by a processor we need to communicate the necessary information. A naive method is to communicate the information as you come across rules in the sorted dependency graph. A smarter way is to group these communications so that communication can be shared and the start up cost of repeatedly sending small messages can be minimized. This can be done by taking advantage of the fact that the communications associated with a variable produced by a rule can be delayed till it is "consumed" by another rule. Therefore we insert synchronization points in the DAG as shown in 4.5. We group the communication for all the rules that produced the variables f, g and h. These synchronization points are found out by using a form of breadth first search. It is not strictly a breadth first search in the sense that we consider only the variables at a particular depth of the DAG such that all rules computing those variables have been applied. The depth at which these variables transition from production to consumption represents a barrier for synchronization where the communication induced by use of maps is resolved.

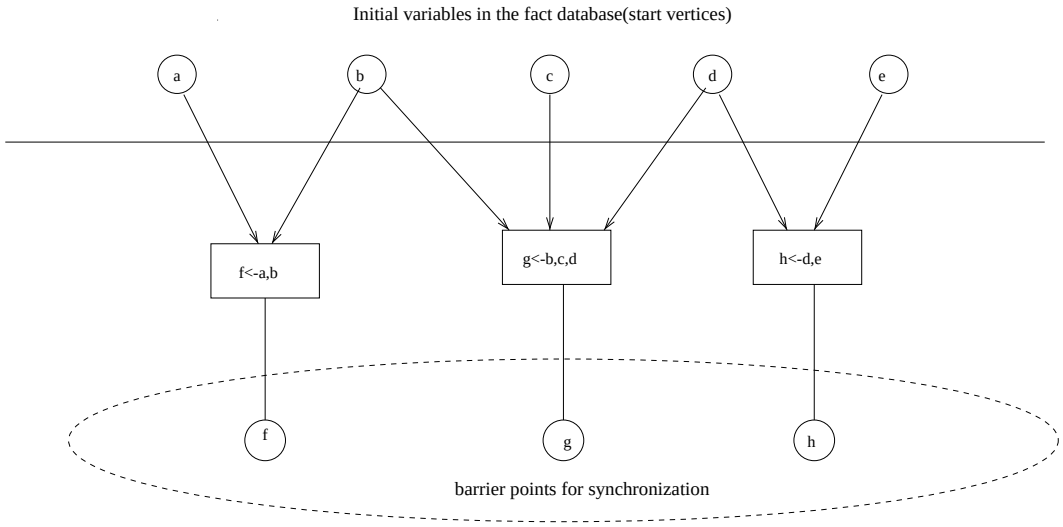

Figure 4.5: A sorted DAG with synchronization points

4.3 Existential Analysis

In the distributed memory version existential analysis takes place in two stages. In the first step, existential deduction proceeds locally as in the serial case. The context of the rules are restricted to the local entities. The existential information derived in this first step will contain entities that are in the clone region. This information needs to be communicated to the processor that owns them so that it can schedule computations for that value. As an optimization step, this communication is done at the barrier synchronization points. Since the clone regions are already identified depending on the existential information of the rules, the processors communicate with each other so that the processors correctly identifies what values need to be computed. As long as there are no maps in the head of a rule the existential deduction is straightforward. But in the case where there are maps in the head of a rule we might end up computing values for some of the entities in the clone region. In this case it is necessary that we send those values to the processor that owns them. At the end of the second step we store the information about the entities in the clone region that are produced by the application of the rule with mapping in its head.

4.4 Pruning and Schedule Generation

The pruning phase proceeds in a similar fashion as the existential deduction phase. There are two stages for the pruning phase also. In the first stage, pruning happens locally as the serial case for the transposed rule database. After the pruning phase we will include only those calculations required to provide the requested variables. In this stage, we start from the requested goal and pass on the requests to the variables that are needed for computing the goal. These variables in turn might generate the need for some other variables. During this phase the context for the rules are restricted to the entities that are owned locally, but the requests that are passed on, are stored in the fact database. In the second stage, we do the communication of entities using the information about the requests stored in the fact database.

Although we have the existential information, once we undergo a pruning phase, we can reduce the computations to a subset of the existential information obtained. The requests stored in the fact database are communicated in a similar way as the existential information was treated. If a processor has a request for a variable, then this variable will need values for entities in its clone region if there are maps in the body of the rule. Similarly for the case with maps in the head of a rule, communication is performed as in the existential analysis phase. During this phase of communication the following information is stored. The data structure used is a list of C++ structures. Each structure stores the following information.

1. Variable for which the values are communicated.
2. The processor with which communication occurs.
3. The *sequence* of entities to be received. We use a *sequence* in this case because we need to receive the entities in the order in which they are sent.
4. The set of entities to be send.

The dependency graph analysis, existential deduction and the pruning phase depends only on the rule database and hence these analyses need to be done only once. Therefore the pruning phase becomes a natural choice for storing the information needed for communication during the execution of the schedule.

4.5 Parallelization Approach

The fundamental data type for the Loci system is the `entitySet`, a value class that describes arbitrary sets of entities. These entity sets are used for existential deduction phase of scheduling and for control and allocation functions. The rules in the rule database are executed by applying it over sets of entities. The developer of the rule provides an inlined member function that performs the specific computations required by a single entity. So if we manipulate the entities over which computations take place we can parallelize the execution of rules.

The Loci system supports different types of rules such as point-wise, singleton, reduction, iterative and recursive rules. Rules are typically represented using text strings called rule signatures. Rule signatures contains the head of the rule, body of the rule and the mapping operator "$\rightarrow$" which represents the use of indirection in a computation. In a distributed fact database, the entities have been distributed such that each processor has its local entities and a set of clone entities. Before we can execute a rule we need to find out the context over which a rule may be applied. The context of the rule is defined as the intersection of the domains of all variable accessors and constraints in the rule specification. The constraint qualifier guarantees that the rule will provide a context for all the entities in the constraint. The context determines the loop bound of the rule. To avoid ambiguity we will rename the context after each stage of schedule generation as follows.

1. $Context\_exists$ is the context obtained after the existential analysis phase.

2. $Context\_schedule$ is the context obtained after the pruning phase.

4.5.1 Point-wise Rule

Point-wise rule is the most common rule in finite difference or finite element applications. For example in the rule $c \leftarrow a, b$, the value for c can be computed over an entity set, provided there exists values for a and b for those entities. The point-wise rules are executed for the stores listed in its head. Before a rule can be computed we need to find out the context over which the rule can be applied. This is done by taking the intersection of domains of all variable accessors and constraints in the rule specification. The final computations take place as follows. $constraint$ consists of a set of entities that satisfy a given constraint equation. $value(a)$ gives the set of entities for which there exists a value for the variable A.

$context\_exists = value(a) \wedge value(b) \wedge constraint$

$FORALL(i \in context\_schedule)$

DO

$c[i] = a[i] + b[i]$

$ENDALL$

The above pseudo-code can be represented using relational algebraic notation [20] as follows.

$C(i, c) : -B(i, b) \bowtie A(i, a) \bowtie constraint$

If there are maps specified in the body of the rule then the range of these maps are used in deriving the context.

$context\_exists = value(a) \wedge value(b) \wedge ran(m) \wedge constraint$

$FORALL(i \in context\_schedule)$

DO

$c[i] = a[i] + b[m[i]]$

$ENDALL$

The above pseudo-code is equivalent to the following expression.

$C(i, c) : -M(i, j) \bowtie B(j, b) \bowtie A(i, a) \bowtie constraint$

In addition to maps we have mapVec and multiMap which contain more than one entity mapped to, for each entity in its domain. An example rule ($c \leftarrow m \rightarrow b$) with the application of a multiMap is shown in figure 4.6. As the map m has entities 0 and 10 in its range for entity 1 in its domain, it is not included in the context(there exists no value for the variable b for the entity 0) . The final context computed is the interval [3,4,5].

For the rules with mapping in the body of the rule, the context computed might need entities which are not owned locally. Initially when we start computation, we assume that we know the values for those entities in the clone region. Since we have a serial fact database on all processors, the necessary information required to start the computations can be obtained without explicit communication. This information is stored when we do the reordering of the fact database. We then apply the rule over the entities in the context. Once the rule is computed we might have a different set of values for the entities owned locally. These entities might be the clone entities for some other processor. So we need to send these values to the neighboring processors(actually to the processors that requested them).

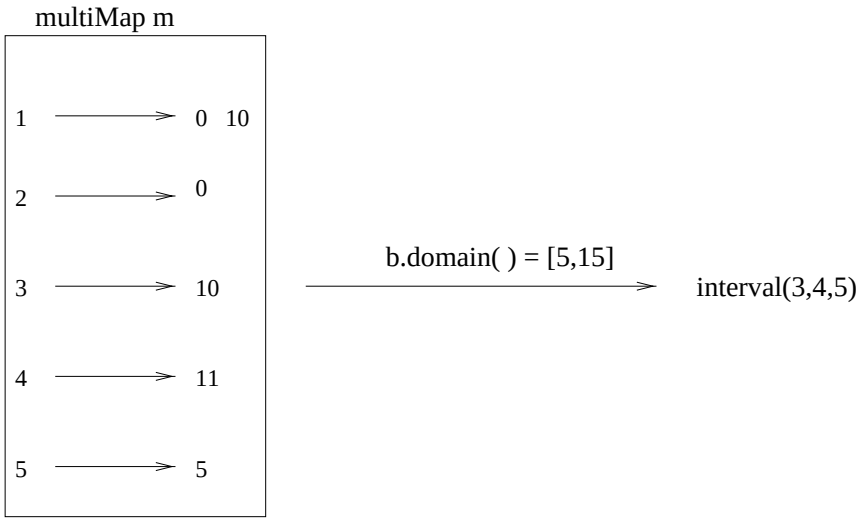

Figure 4.6: Effect of maps in the body of a point-wise rule

Additional complication arises when there are maps in the head of a rule. For the case $m \rightarrow a \leftarrow b, c$ the rule is computed as follows.

$context\_exists = dom(m) \wedge value(b) \wedge value(c)$

$FORALL(i \in context\_schedule)$

DO

$a[m[i]] = c[i] + b[i]$

$ENDALL$

The final results are generated for entities which may or may not be in the *context_exists* of the rule. This might cause a processor to compute values for entities that are not locally owned by the processor. These values have to be send to the processor that actually owns them. If we perform the necessary communication after the application of the rule, we can be assured that our initial assumption, that the local and clone entities will have the right information, will hold true for each iteration.

In all the above examples, we showed the computations being performed over the entities in the *context_schedule*. This *context_schedule* is determined after the pruning phase of schedule generation. The only difference in the sequential and parallel version is that we need to perform communication at each synchronization point. In a serial or a shared memory implementation all the entities are present in a common memory location and hence the entities can be accessed without any communication. By dividing the *context_schedule* over which the rule is being

computed, we can parallelize the execution of the schedule. For the distributed memory version, we adopt the "owner compute" rule as the scheduling policy. Owner compute means that a particular loop iteration is executed by the processor owning the entity for which the head variable is computed. In other words, each processor loops over the *context_schedule* which is restricted to the locally owned entities.

4.5.2 Singleton rule

Singleton rules do not contain any store variables in the rule head or the rule body. In this case the rule computes a single value for the parameter in the head of the rule. The parameter variable is present on all the processors. Hence the value can be computed independently on all the processors and there is no need for any communication in this case.

4.5.3 Reduction Rule

When a reduction rule is used to compute a variable, the value obtained is the result of a set of rule applications. The first step is the application of the unit rule. No value for a reduction can be generated without first binding a unit value to the target entities. Then the apply rules act on the unit rule initialized entities. Once the apply rules complete, the final value of the entity is uniquely determined based on the assumption of the associative and commutative properties of the operator in the apply rules.

Reduction rules can be divided into two types depending on the type of variable in the head of the rule. When the head variable is a parameter then the reduction is called parameter reduction and if the head variable is a store then the reduction is termed store reduction. For the case with parameter reduction only the head variable is restricted to parameter type while the body may contain stores, maps and parameters. So each processor might compute a different value for the parameter. These different values need to be combined using the `join` operation so that all the processors end up with a unique value for the parameter. This `join` operation on all the partial results is performed using `MPI_Allreduce`.

Store reductions are done in a way similar to point-wise rules. Each processor computes the values locally. If there are maps in the head of a rule, the store reductions end up accumulating partial sums for the clone entities. These partial results have to be send to the processor that owns them. The processor that actually owns them will have a partial result for those entities

computed using its local context. The processor that receives the two partial results accumulates them using a `join` operation. The final results of join are send to all the processors that made requests for those entities.

4.6 Communication Routines and Data Structures

Communication routines were written which would perform the two types of communication needed for the execution of rules. The `fill_entitySet` and `send_entitySet` was written to perform the communication of entities during the existential deduction and the pruning phase. These are the two main types of communication happening in the entire schedule generation and the schedule execution phase. Most of the other communications are either a variation of these two or done using the information collected using these two types of communication.

1. The `fill_entitySet` routine makes use of the information stored in the `distribute_info` to find out the processor IDs from which a particular processor has to receive the clone entities. An entity set is passed to the routine. This entity set is used to determine the entities that are to be send by this particular processor to fill the clone regions of the neighbors. As an optimization, another routine was written which takes in a vector of entity sets. Since a vector of entity sets is passed to the routine, the processor loops over all the entity sets and groups the entity sets such that only a single message needs to be send to a particular processor.

2. The `send_entitySet` routine performs the communication to handle the case of maps in the head of a rule. If there are maps in the head of a rule and it produces values for entities in the clone region, then those values must be communicated to the processor that owns them. The entity set passed to the routine is intersected with the clone entities and the resulting entities are communicated. As in the case of `fill_entitySet` another routine which processes a vector of entity sets was also written.

3. The `send_requests` routine performs the same operation as the `send_entitySet` but for the fact that it stores the necessary information needed for communication during schedule execution.

4. The `sort_comm` routine makes sure that the entities are received in the same order in which they are packed in a particular processor. This routine returns the final list of structures which are used for communication during schedule execution.

Once we have a contiguous numbering of the local entities and the clone regions, we can utilize this contiguous information to group the data for communication more efficiently. Each of the store types are provided with a set of methods for determining the size of the data to be packed, for packing the data into a buffer and also for unpacking the packed buffer into a store. For communicating entities we use an integer buffer as, entities are integer identifiers. For communicating stores and parameters we need to have some way of minimizing the number of messages send. This can be done if we can group all the different types of data communicated so that only a single message needs to be send between two processors. This grouping of data was done using the `MPI_Pack` and `MPI_Unpack` functions [21]. Since we pack the values into a single buffer in the form of bytes(`MPI_BYTE` data type), we can group the communication of different variables having values of different data types.

4.7 Communication Protocols

For communication purposes the most straight forward approach is to use a point to point communication using plain `MPI_Send` and `MPI_Recv` functions. But these are blocking calls and therefore `MPI_Send` doesn't complete until the buffer is empty(available for reuse) and `MPI_Recv` doesn't complete until the buffer is full(available for use) [22]. To minimize the latency in communication we use the `MPI_Irecv`, `MPI_Send` and `MPI_Waitall` communication structure. During the fact decomposition phase and the execution schedule generation phase, we store the information needed for communication. So we know the processors with whom we have to communicate and also message details are available at the time of performing the communication. Therefore our sole aim is to perform the communication in the most efficient way as possible.

When we use `MPI_Irecv` to initiate a non blocking communication we can overlap computation with communication. We first post the receive operation followed by a set of operations to set up the send buffer and then perform the send operation. We then do a `MPI_Waitall` on all the receive operations. Since we post the receive operations first, as soon as a corresponding send operation is posted by another processor, the send-receive communication operation gets completed between

that pair of processors. In all the communication routines, memory is allocated during run time for the buffers used during communication. To optimize the memory allocation, memory is allocated as a big chunk and wherever possible the buffers are reused.

In our initial approach for performing the communication, we determine the size of the message to be received from the information stored in the fact database. While this approach will work for all the static containers like stores and parameters, it could create problems for dynamic containers like the storeVec and the multiStore since the size may not be known for the receiving processor. A naive method to solve this problem was to send the size of the message to the receiving processor such that it allocates the buffer to receive the message. This approach has the disadvantage that it incurs an additional start up cost to send this extra message. To minimize this extra start up cost we store the message size from the previous iteration. The approach followed can be summarized into the following steps.

1. As a first step we need to allocate a buffer for receiving the messages. We have an integer array to store the maximum message sizes that has been sent to the neighboring processors. Initially this integer is set to store the size of an integer. The size of the buffer to be allocated is determined after comparing with the maximum message sizes previously received.

2. If the size returned by using the `pack_size` method is equal to the size of an integer or if it is greater than the maximum size of the message send to that particular processor, then the size of the message is sent instead of the message. If we end up sending the sizes alone, we create another list to send the message a second time.

3. On the receiving front, we first receive all the message from the list of neighboring processors. If the size of any of the messages received is equal to the size of an integer, then we have to receive a second message from the same processor. The only problem with this approach is when the store has values of integer type and the size of the message sent is equal to the size of an integer. In that case we might end up receiving the same message twice. But since this might be a rare occurrence, it can be ignored.

In effect we have a protocol which can handle message sizes that vary at run-time. As long as the message sizes remain consistent, we can be assured that the cost for memory re-allocation and sending the messages will be a minimum.

4.8 Summary

A detailed description of the implementation is given in this chapter. The description of the development process is given in a chronological order. A generalized partitioning scheme which can be applied to any serial fact database is introduced. The schedule generation process is the same as the sequential version, except for the identification of barrier synchronization points in the DAG, where the communication of entities take place. In addition to the description of the methodology adopted to handle the parallelization of the different types of rules, a description is provided for the additional data-structures and sub-routines needed for the distributed memory implementation.

CHAPTER V

RESULTS

This chapter presents some performance measurements of the distributed memory version of Loci obtained from the simulations run using the finite-rate chemistry solver *Chem*, which was developed using the sequential Loci framework. The *Chem* code has already been thoroughly tested and validated on both sequential and thread based shared-memory implementation of Loci. A comparison with the results from the original sequential version gives an assurance that a consistent and correct implementation has been achieved. The current distributed memory version has been tested on various platforms, including the SUN Ultra and SGI Challenge compute servers and the Intel PC's(running on linux) using the Kuck & Associates compiler, the GNU g++ compiler, and the CC compiler provided by both SUN and SGI. The performance measurements were done on the 400 MHz SUN UltraSPARCII Cluster (32 GB of RAM, 64 processors on 16 nodes).

5.1 Issues

Although the aim of this thesis is to completely automate the parallelization of an application developed in the Loci framework, some issues are yet to be resolved. The following section discusses some of them, and the methods taken to resolve them. The most important question is: " How much of the support for the distributed memory version needs to be in the framework and how much of it should come from the application developer?". This is a difficult question as the best answer may depend on the application being developed and the size of the problems being solved.

5.1.1 Input/Output(I/O)

There are a number of ways in which the I/O part of an application can be designed to deal with distributed memory architectures.

1. A single processor creates a serial fact database, partitions it and then broadcasts the information to the other processors.

2. All processors simultaneously create the serial fact database, partition it and reorder the fact database such that it contains only the locally owned entities and the clone entities (Chapter IV).

Either approach is a viable option as long as there is no constraint on the memory available for the simulation. When we run larger problems, the memory consumed can be large, especially if we use the generalized partitioning strategy described in chapter IV.One of the goals in developing a distributed memory implementation was to solve very large problems. During test runs using moderately large grids, it was found that coupling the partitioning using METIS is not a scalable option with respect to memory. To run large problems, an option was added to read in a file containing the partition of entities, if the user chose to do so.

Another issue which had to be resolved was outputting the final solution. The results obtained after running the simulation on a distributed memory architecture will have the final solution scattered among processors. In order to check whether the final solution matches with that of the sequential version, the solution has to be collected on to a single processor and written out to a file. This approach is not an efficient one for very large problems, as it involves the communication of huge messages between processors and the serialization of file writing.

5.1.2 Load Balancing

The current distributed memory version of Loci has no provision for providing load balancing support. Currently it relies on the initial partitioning of entities among processors for performance improvements. A preliminary study was done on the load balancing aspect. The benchmark cases were run using two kinds of partitioning: the general partitioning strategy which we expected to be unbalanced (Chapter IV) and a more balanced partitioning (which could be provided by the application developer). METIS was used to create both partitions.

A more "balanced" partitioning was obtained by the following strategy (this method of partitioning had to be coupled with the *Chem* code as it is specific to the kind of problems being solved). First, a partitioning is obtained for the cells in the grid. This partitioning produced fewer edge-cuts(approximately ten times less) than the unified connectivity graph described in Chapter IV. Then we loop over the maps in the fact database to get the connectivity information about the nodes and faces. This partitioning of cells, faces, and nodes is not unique, as there might be duplication of faces and nodes on different partitions. In order to make the final partitioning unique, we compare the total number of faces or nodes of each partition. For example, if two processors share a particular face, the ownership of the face is given to the processor which has lesser number of faces. A similar treatment is done on the nodes also.

From the preliminary results of speed-up measurements, it appears that there is a slight imbalance in the load for the generic partitioning provided by Loci. A generalization cannot be made in this regard till more extensive studies are conducted.

5.2 Validation of the Implementation

The development of the distributed memory version of Loci evolved in a number of stages. After obtaining a partitioned fact database, the first step was the comparison of the existential deduction information with that obtained from the serial case. The next step was the validation of the pruning phase. The final context over which the rules are applied was collected on to a single processor and compared with that of the sequential case. Once the results of the existential analysis and the pruning phase matched that of the sequential version, the next step was the validation of the steps taken to parallelize the different types of rules. During the development process, several test cases were written: for example, test cases with, rules having no maps, rules having maps in its body, rules having maps in its head and rules having maps in its head as well as body. For reduction rules, separate test cases were written for parameter reduction and store reduction. In addition, test cases which incorporated a combination of the different rule types were also considered. In all these test cases, the final results were collected on to a single processor and the results were compared with that of the sequential case.

The communication routines were tested extensively by randomly distributing the entities amongst the processors and also by running the test cases on a varied number of processors.

After testing all the rules individually during the development phase, the distributed memory implementation was tested using a more complicated application: the *Chem* code. Several bugs were eliminated during the testing of the distributed version of Loci using the *Chem* code. In addition to the *Chem* code, the distributed memory version was also tested on a finite element application and the results obtained matched that of the sequential version. Currently, the distributed memory version is being used for Large Eddy Simulations and also for simulations involving pollutants in cities. The grids used for these simulations are larger than the ones used for the test cases during the development.

Since the development process took place in a step by step procedure and as each step was thoroughly tested before proceeding to the next step, we can be somewhat assured of the robustness of the current implementation. The distributed memory version has also been tested extensively by the users of the *Chem* code and as of now there are no outstanding issues. The sample case shown below for validation purposes is just one of the numerous examples used for testing.

5.2.1 A Sample Case

The simulation of flow of high temperature air in a supersonic nozzle was used to validate the *Chem* code during its development [16]. The same test case is considered for testing the validity of the distributed memory implementation developed. The geometry of the nozzle is given by the relationship:

$$A = \frac{\pi}{16}\left[1 + \sin\left(\frac{\pi x}{2L}\right)\right]^2, \tag{5.1}$$

where L is the nozzle length. For this example, L is two meters. A 100 by 40 point axi-symmetric grid is used for this simulation, as illustrated in figure 5.1. The supersonic inlet boundary conditions are given by a pressure of $p = 1.945 \times 10^5\ Pa$, a temperature of $T = 9001\ K$, and a density of $\rho = 0.0385433\ Kg/m^3$. The mixture is set to the equilibrium composition given by the mass fractions 1.33836×10^{-5}, 0.233854, 0.0515657, 0.713442, and 0.00112502 for $O2$, O, $N2$, N, and NO, respectively. The velocity of the inlet flow is 4000 m/sec.

The Mach number and the temperature contours obtained for the sequential case are shown in figures 5.2 and 5.4 respectively. Figures 5.3 and 5.5 shows similar contours obtained for the

distributed memory version, and they are undistinguishable. The output files used for generating the contour plots were compared with that of the sequential version by using the $diff$ command and there were no difference in the solution files (the solution files were outputted in single precision).

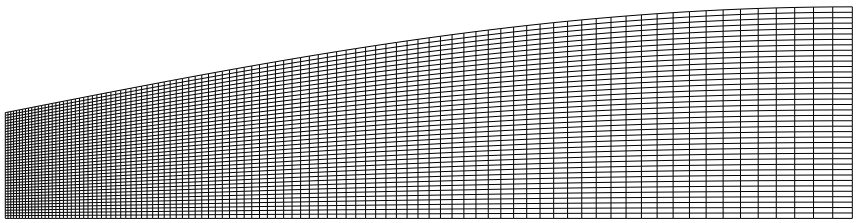

Figure 5.1: Supersonic Nozzle Grid

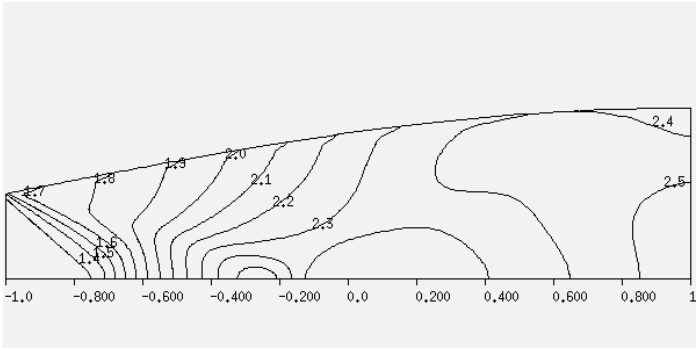

Figure 5.2: Mach Number Contours (Supersonic Nozzle, 5 Species Air for the Sequential Case)

5.3 Performance Measurements

To test the ability of the sequential version of the $Chem$ code to solve more realistic problems, simulations were performed for the case of a jet emanating from a convergent-divergent nozzle and impinging on a perpendicular plate(Case-I) and an inclined plate(Case II)[23]. These test cases are used for the performance measurements of the distributed memory implementation. The convergent-divergent nozzle is shown in figure 5.6. A three-block structured grid is used for this case. The grid is clustered near the impingement plate. There are approximately 0.25 million cells in the grid used for this simulation. The air jet is channeled from a high-pressure reservoir

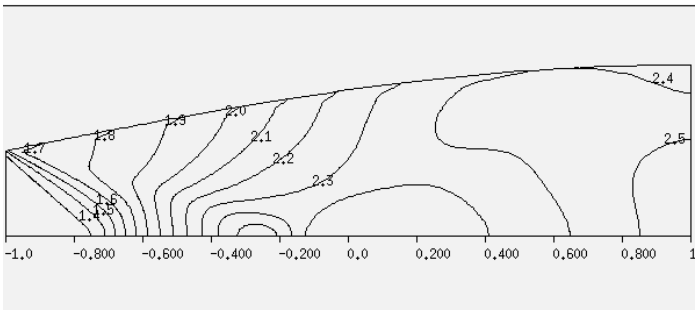

Figure 5.3: Mach Number Contours (Supersonic Nozzle, 5 Species Air for the Distributed Memory Implementation)

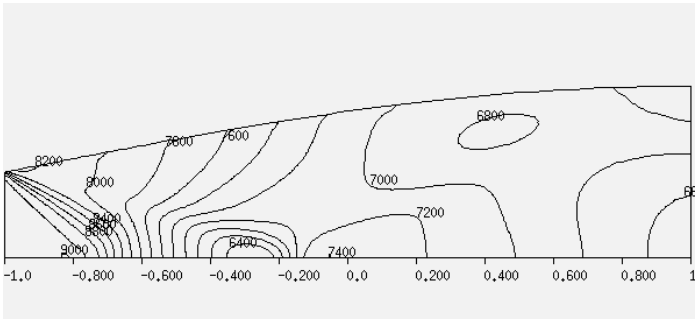

Figure 5.4: Temperature Contours (Supersonic Nozzle, 5 Species Air for the Sequential Case)

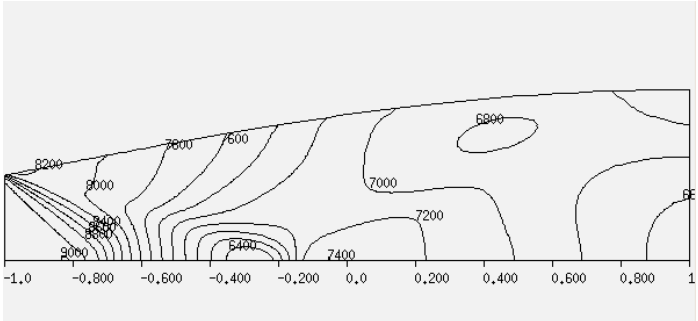

Figure 5.5: Temperature Contours (Supersonic Nozzle, 5 Species Air for the Distributed Memory Implementation)

into the atmosphere and is impinged on the flat plate. Two types of partitioning strategy are used for running the simulation on a distributed memory platform: the generic partitioning strategy (Chapter IV), and the partitioning strategy provided by the application developer.

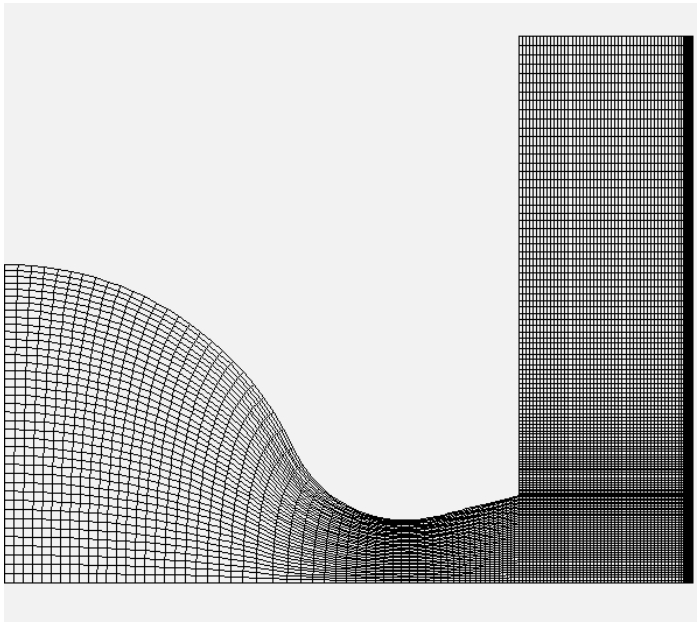

Figure 5.6: 3-block structured grid used for the 2D perpendicular plate impingement case

5.3.1 Jet Impingement on a Perpendicular Plate(Case-I)

The distance between the nozzle and the perpendicular plate is set to $2D$, where D is the nozzle exit diameter. An explicit Runge_Kutta time integration scheme is used along with Van Leer [24] flux splitting scheme, as both schemes are computationally less intensive. Moreover, the Baldwin-Lomax turbulence model is used, as it tend to use more communication. The idea is to reduce the computations involved and increase the communications so that we can get a better knowledge of the overhead incurred in the communication modules. The simulation was run for one hundred iterations, with a CFL number low enough to ensure stability.

A crucial way to quantify the performance of a parallel implementation is by using the notion of speedup. Ideally one might expect the speedup to be directly proportional to the number of

processors employed. However, in practice, the overhead incurred by communication weakens the potential for a perfect speedup. Figure 5.7 shows the time taken for creating the execution schedule(which involves the graph processing time and also the existential and pruning stages of the schedule generation) while figure 5.8 shows the speedup obtained for execution time for the two types of data distributions.

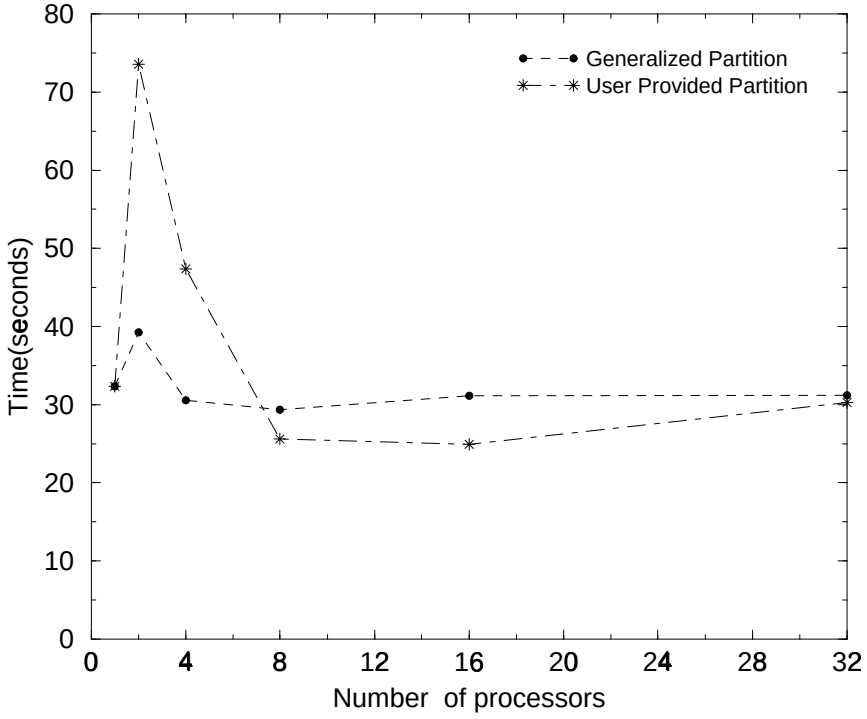

Figure 5.7: Time taken for creating the execution schedule (Case-I)

The efficiency plot shows the percentage of utilization of the processor set. A consequence of Amdahl's law [25] is that the efficiency drops with an increasing number of processors if problem size is held constant. The observed results which are shown in figure 5.9 conform to Amdahl's law.

From the speedup and efficiency curves, it can be noted that the generalized partitioning strategy leads to a reduced utilization of the processors compared with the more "balanced" partition provided by the application developer. This supports our initial hypothesis that the generalized partitioning might have a load imbalance(Chapter IV). The time taken for creating the execution schedule is comparable to that taken by the sequential version except for the case with the run on two processors. This could be attributed to the initial partition imbalance. One

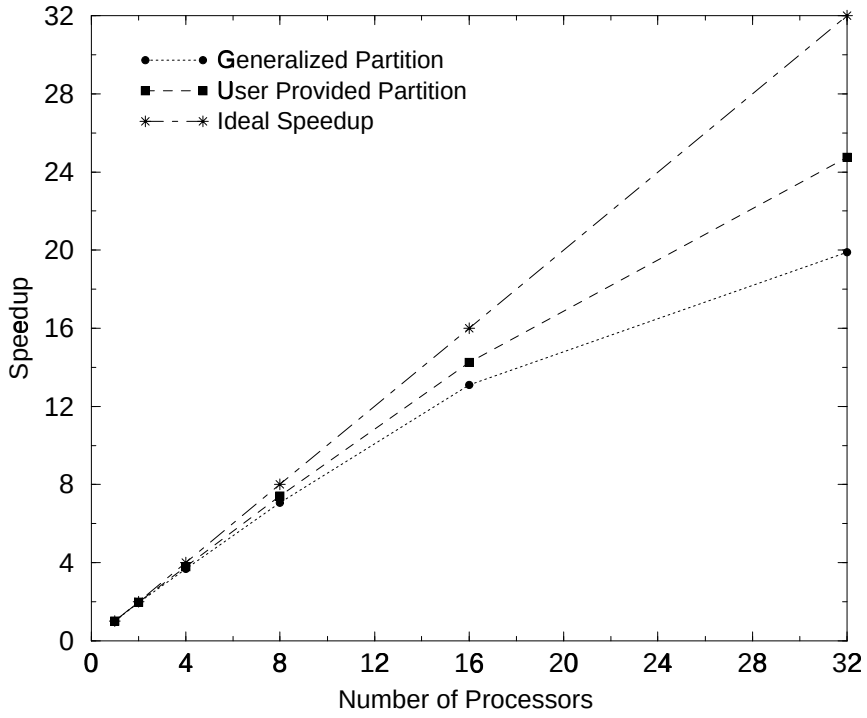

Figure 5.8: Speedup obtained in execution of the schedule (Case-I)

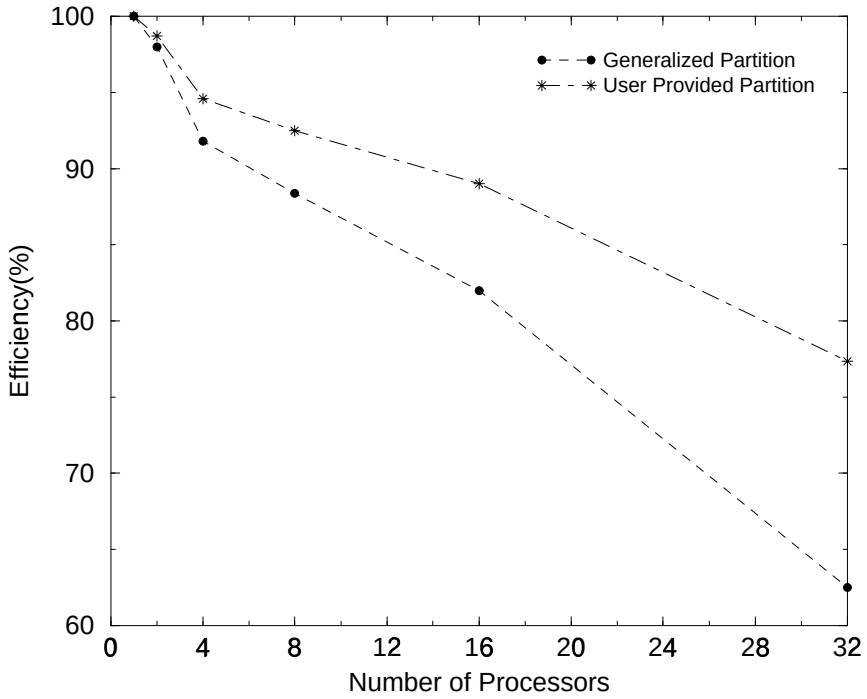

Figure 5.9: Plot showing the efficiency of processor utilization (Case-I)

cannot expect to get a fair measure of speedup on this timing as the operations involved are barely computationally intensive.

5.4 Jet Impingement on an Inclined Plate (Case-II)

The distance between the nozzle exit and the center of the plate is chosen to be $5D_N$ where D_N is the nozzle exit diameter. The angle that the plate makes with the nozzle axis is taken to be 35 degrees. An implicit Euler time integration scheme is used, with one Newton iteration per time step. In addition, six Gauss-Seidel iterations are performed. This test case was chosen because it is computationally intensive, and hence, it enables us to get a better idea of the efficacy of our implementation.

The Gauss-Seidel algorithm used for the parallelization is a $relaxed\ Gauss-Seidel$ method. The convergence rate is usually affected by the frequency of updates for the values residing in the clone region. In the case of fewer updates, there is a slower propagation of new computational information(hence a reduced convergence rate), but there is a gain in the performance as the communication cost incurred is smaller. The relaxed Gauss-Seidel algorithm makes use of fewer updates of the clone entities to increase the performance.

The plots of the time taken for creating the execution schedule and the speedup obtained for the execution times are shown in figures 5.10 and figures 5.11 respectively. The corresponding efficiency plot is shown in figure 5.12. For these studies, the number of Gauss-Seidel iterations was kept constant as we are evaluating the communication and not the algorithm efficiency. The efficiency plot for this simulation shows a better utilization of the processors than Case-I, as expected.

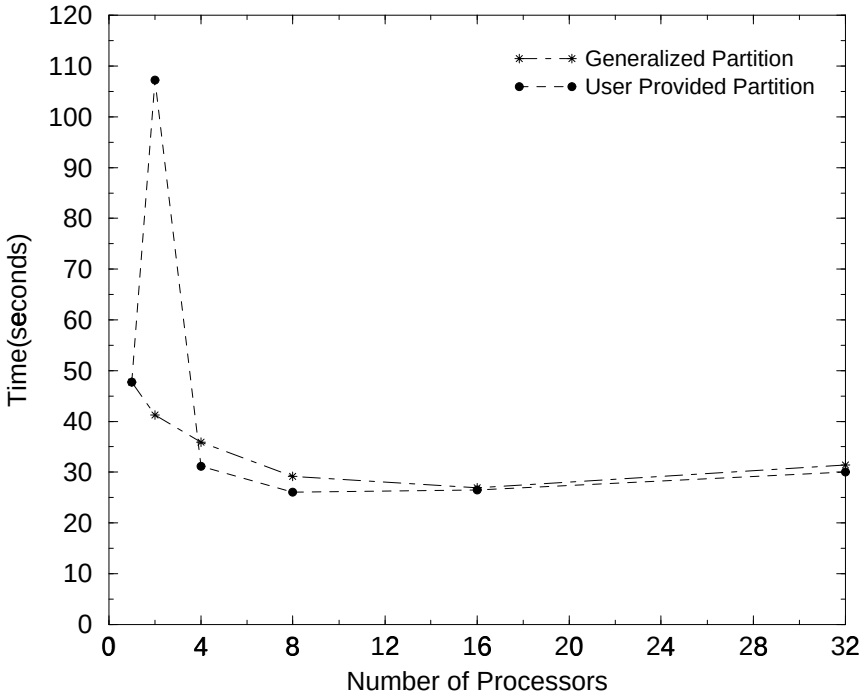

Figure 5.10: Time taken for creating the execution schedule (Case-II)

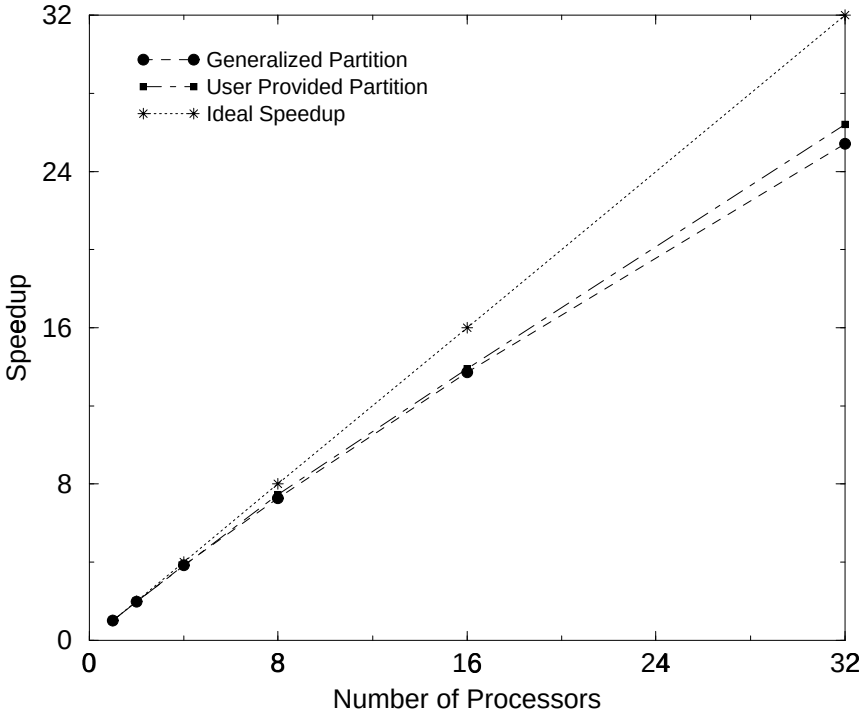

Figure 5.11: Speedup obtained in the execution of the schedule (Case-II)

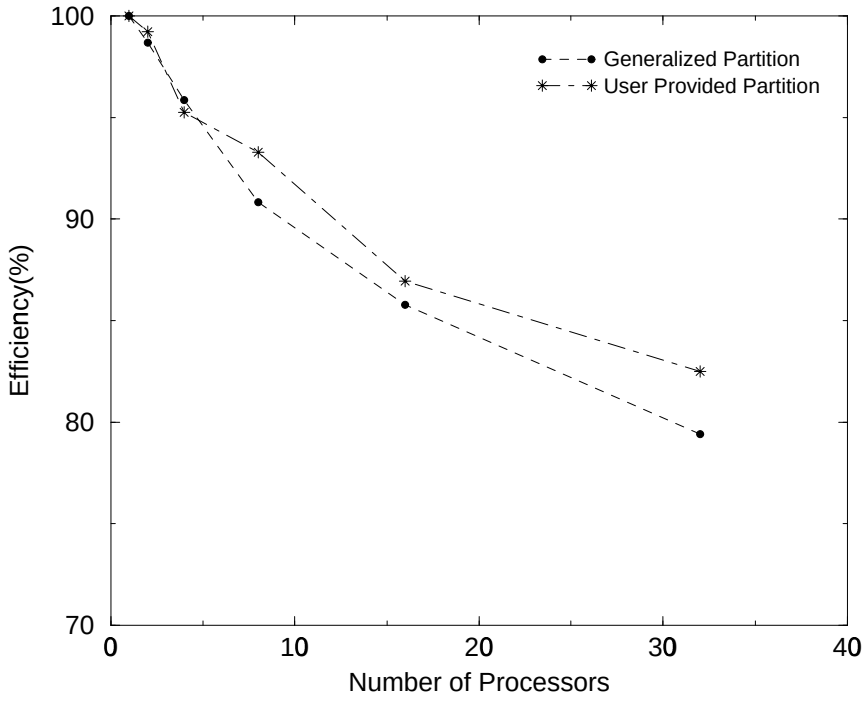

Figure 5.12: Plot showing the efficiency of processor utilization (Case-II)

CHAPTER VI
CONCLUSION

The present study demonstrates a method to develop a distributed memory implementation for a concurrent logic programming based coordination framework. The current distributed memory version is tested using the finite-rate chemically reacting flow solver *Chem* developed in the Loci framework. A comparison between the results obtained for the serial and parallel execution of the code ensures that a correct implementation has been achieved.

The main goal of the implementation was to have an automatic parallelization tool for the developers of CFD application. The present implementation has achieved the goal for moderately large problems. Since the Loci framework is designed to be used for any graph based applications, a generalized partitioning strategy has been evolved in order to achieve completely automatic parallelization.

6.1 Future Work

While testing the validity of the distributed memory version using the Chem code, we encountered certain issues (discussed in Chapter V). These issues have to be resolved in order to have a scalable and optimized implementation. Certain compromises have been made in order to have a completely automatic parallel application - the most crucial one being, the decision on the starting point of parallelization. Presently, the current implementation is not the most optimal implementation possible. It is open to future work in the following avenues:

1. The overhead incurred by the communication modules could be minimized, by reducing the dynamic memory allocation of the buffers and also by reducing the number of redundant messages if there are any.

2. The HDF5 support in the Loci framework could be utilized to provide support for parallel I/O for the applications so that very large problems could be solved. This approach might

require essential modifications in the I/O part of applications, which are already developed in this framework.

3. Enable load balancing capabilities within the Loci framework to maximize the performance obtained by the distributed memory implementation.

4. The current generalized partitioning routine could be modified to use a different partitioning tool other than METIS, or a different partitioning strategy could be developed to partition the unified connectivity graph in order to reduce the memory usage.

5. Efficient algorithms and methods from the relational database literature could be explored for developing a more efficient implementation, as this work has been just an innovative attempt made towards developing a distributed memory version of the Loci framework.

REFERENCES

[1] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," 1991.

[2] T. Sterling, D. Becker, D. Savarse, J. Dorband, U. Ranawak, and C. Packer, "Beowulf: A parallel workstation for scientific computation," in *Proceedings of the 1995 International Conference on Parallel Processing*, vol. 1, pp. 11–14, ICPP, August 1995.

[3] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, "The design and implementation of a parallel unstructured euler solver using software primitives," *Journal of the American Institute of Aeronautics and Astronautics*, vol. 32, pp. 489–496, March 1994.

[4] F. P. Brooks, "No silver bullet," in *Proceedings of the IFIP Tenth World Computing Conference* (H. J. Kugler, ed.), pp. 1069–1076, Elsevier Science, 1986.

[5] E. A. Luke, "Loci: A deductive framework for graph-based algorithms," in *Proceedings of the 3rd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'99)*, Lecture Notes in Computer Science, ISCOPE, Springer-Verlag, 1999.

[6] V. Kotlyar, K. Pingali, and P. Stodghill, "A relational approach to sparse matrix compilation," in *Proceedings of the EuroPar '97 Conference*, 1997.

[7] I. Ahmad, Y. Kwok, M. Wu, and W. Shu, "Automatic parallelization and scheduling of programs on multiprocessors using casch," in *Proceedings of the 1997 International Conference on Parallel Processing*, pp. 288–291, ICPP, 1997.

[8] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling fortran d for mimd distributed memory machines," in *Communications of the ACM, Vol. 35, No. 8*, August 1992.

[9] V. Kotlyar, K. Pingali, and P. Stodghill, "Compiling sparse code for sparse matrix application," in *Proceedings of the ACM/IEEE SC'97 Conference*, ACM, 1997.

[10] R. Das, M. Uysal, J. Saltz, and Y. S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 462–479, September 1994.

[11] J. V. W. Reynders, J. C. Cummings, P. J. Hinker, M. Tholburn, M. Srikant, S. Banerjee, S. Karmesin, S. Atlas, K. Keahey, and W. F. Humphrey, *POOMA: A Framework for Scientific Computing Applications on Parallel Architectures*. 1996.

[12] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1990.

[13] I. Foster, R. Olson, and S. Tuecke, "Productive parallel programming: The PCN approach," *Scientific Programming*, vol. 1, pp. 51–66, 1992.

[14] K. Birken, "Towards automatic parallelization of numerical software based on formal specification concepts," in *Concepts of Numerical Software* (W. Hackbusch and G. Wittum, eds.), Notes on Numerical Fluid Mechanics, Vieweg Verlag, 1998.

[15] K. Birken, "Semi-automatic parallelization of dynamic, graph-based applications.," in *Parallel Computing: Fundamentals, Applications and New Directions.* (E. D'Hollander, G. Joubert, F. Peters, and U. Trottenberg, eds.), Elsevier Science, 1998.

[16] E.Luke, *A Rule Based Specification System for Computational Fluid Dynamics.* PhD thesis.

[17] R. S. Bird and L. Meertens, "Two exercises found in a book on algorithmics," in *Program Specification and Transformation* (L. G. L. T. Meertens, ed.), pp. pp 451–457, North-Holland, 1987.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* MIT Press, 1999.

[19] G. Karypis and V. Kumar, *METIS: A Software package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices , Version 4.0.* University of Minnesota, 1998.

[20] J. D. Ullman, *Principles of Database and Knowledge-Base Systems Volume 1: Classical Database Systems.* Computer Science Press, 1999.

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

[22] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface second edition.* The MIT Press, 1999.

[23] J. Wu, L. Tang, X. Tong, E. Luke, and P. Cinnella, "Numerical simulations of jet impingement and jet separation," tech. rep., Mississippi State University, 2000. MSSU-ASE Report No. 00-2.

[24] B. V. Leer, "Towards the ultimate conservative difference scheme. II. Monotinicity and conservation combined in a second order scheme.," *Journal of Computational Physics*, vol. 32, pp. 101–136, 1979.

[25] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities.," in *AFIPS Conference Proceedings*, vol. 30, pp. 483–485, April 1967.