UPPSALA
UNIVERSITET

IT 12 050
Examensarbete 30 hp
Oktober 2012

Adapting a Radial Basis Functions Framework for Large-Scale Computing

Afshin Zafari

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Adapting a Radial Basis Functions Framework for Large-Scale Computing

Afshin Zafari

This work is aimed at extending a parallel computing framework for radial basis functions methods for solving partial differential equations. Existing framework uses Task Based parallelization method in shared memory architectures to run tasks concurrently on multi-core machines using POSIX Threads. In this method, an algorithm is viewed as a set of tasks each of which performs a specific part of that algorithm while reading some data and producing others. All the dependencies between tasks are translated into data dependencies which makes the tasks decoupled. This work uses the same method but for distributed memory systems using message passing scheme of inter-process conversations. These frameworks cooperates with each other for distributing and running the tasks among nodes and/or cores in a hybrid way of multi-threading and message passing parallel programming paradigms. All the communication between processes (nodes) are performed asynchronously (non-blocking) to be overlapped with computations and the execution flow of the framework is implemented using state machine software construct.

Handledare: Elisabeth Larsson
Ämnesgranskare: Jarmo Rantakokko
Examinator: Jarmo Rantakokko
IT 12 050
Sponsor: Martin Tillenius

Tryckt av: Reprocentralen ITC

Contents

Chapter 1

Background

Introduction

In this project, we are aiming to complete an existing software for solving radial basis function (RBF) approximation problems. Initial objective of this existing software is reaching to a set of library or program modules which give several functionalities and components to be used for solving an RBF problem. The main user of such a software are the domain experts who know the mathematical model of the problem and the crucial parameters regarding accuracy and stability of the methods for solving them. On the other hand, the software has to be efficient enough to make the best use of the computational capability of the underlying resources for high performance computing (HPC). The software will hide the techniques and methods used for best utilization of the computational resources from the main users by encapsulating the required functionalities inside some prepared and tested program modules. This way, the main user is no longer worried about the ways that the solution to the problem may be implemented, and only needs to focus on the definition of a given problem by specifying various parameters that are related to different questions to be answered and verified.

The efficient resource utilization implies that the modules are responsible for optimized memory management and CPU usage. This means that the modules have to be aware of the resources that are available to them and depending on the problem parameters decide which methods fit better. The main important issue in considering the resources, as in any HPC program, is how the program can be executed in parallel when multiple CPUs are available. These CPUs may be hosted on a single computer or distributed over a network of computers, or both. When the CPUs are hosted in a single computer they all have a shared access to the main memory of the host and this is called a shared-memory system. On the contrary, when the CPUs are distributed over multiple computers in a network, their memory access is restricted accordingly, since the CPUs hosted on different computers have no direct access to each other's local memory. This model , that is called a distributed-memory system, uses message-passing style of programming to let the programs running on different nodes of the

network communicate with each other by transmitting and receiving messages. Due to these distinct ways of programming for shared- and distributed- memory systems, efficient resource utilization translates into a hybrid method of using both styles to gain the best performance of each.

1.1 Introduction to radial basis functions method

The radial basis functions methods rely on the basic idea of approximating a multivariate function with a linear combination of radial basis functions. For approximating a function $u(\underline{x})$ the linear combination

$$s(\underline{x}) = \sum_{j=1}^{N} \lambda_j \phi(\varepsilon \|\underline{x} - \underline{x}_j\|) \equiv \sum_{j=1}^{N} \lambda_j \phi_j(\underline{x}) \tag{1.1}$$

where $\phi(r)$ is a radial basis function, $\underline{x}$ is a point in $\mathbb{R}^d$, $\underline{x}_j$ are the center points for RBFs, ϵ is the shape parameter and λ_j are the coefficients to determine. To find the λ_js from these equations, it is sufficient to solve the following system of linear equations

$$\mathbf{A}\boldsymbol{\lambda} = \mathbf{u}, \tag{1.2}$$

where $\mathbf{A} = (a_{ij}) = \phi_j(\underline{x}_i)$ for $i, j = 1, ..., N$, $\boldsymbol{\lambda} = (\lambda_j)$ and $\mathbf{u} = \big(u(\underline{x}_j)\big)$. The most important advantage of this method is that it works directly with scattered nodes for the approximation which means that it is mesh-free and flexible with respect to the geometry of the computational domain. This also means that this method is suitable for high-dimensional applications (like option pricing , e.g. [22] ,[19]) since it uses only a single geometrical property (distance) of the points. These are the main attributes (mesh-free and suitability for high-dimensional problems) of this method that makes RBF methods different from other methods like finite difference methods (FDM) and finite element methods (FEM) for solving partial differential equations.

1.2 Comparing RBF with finite difference and finite element methods

The mesh-free attribute of RBF methods makes them more efficient in processing data and there is no need to access to the points that lie in a stencil determined by the finite difference method or come from a mesh constructed by finite element method. This makes some computations (like matrix assembly) data-parallel while ate the same time produces a dense matrix of coefficients to be solved.

In addition, the RBF methods can handle scattered data points and are dependent on their structures, the data can be held in memory in an efficient way targeting better performance and resource utilizations. In FDM and FEM methods, the resulting matrix of coefficients are sparse and banded and thus many of the matrix elements would be zero for which computations would

be skipped. Keeping the whole matrix in memory wastes the memory resource too much for holding non useful elements and storing only the non-zero elements (as done in sparse storage formats) shuffles the originally well-defined and predetermined access patterns to elements of the matrix since the adjacent elements are no more accessed by easily incrementing or decrementing indexes. Thus RBF methods neither waste memory for holding non useful elements nor miss the possibility of efficient structuring of the elements for better performance.

1.3 Parallel RBF methods

Dealing with larger number of dimensions requires both handling much more computer memory for computations and numerical solutions and efficient simulations within an acceptable time. Limited resources of time and computer memory for a single computer requires using multiple computers in parallel efficiently enough to handle large problems in terms of response time and resource consumption. The major computations in RBF methods consist of preparing data points, computing the distance and $\phi(\cdot)$ functions and finally solve the system of linear equations of (1.2). The main focus of this work is on assembling the matrix of coefficients in (1.2). To the best of our knowledge, all the efforts and research for parallel execution of RBF methods are disparate and limited to specific algorithms or problems owned by researchers in this area (e.g. [5] ,[28],[23],[14],[20]) and hence there is no single framework for RBF methods which can run in multiple hybrid and/or heterogeneous processors environments. This work is a part of a project whose main objective is to provide such a framework for the end users while at the same time make the technical difficulties of high performance computing transparent.

1.4 Introduction to shared memory software framework

The existing software for solving the RBF problem designed and implemented is to be both user friendly and efficiently parallelized for multi-core architectures. This part of the software performs well in various experiments that can be found in [25] together with the details of the design. To parallelize execution of any algorithm, the solution method is decomposed into tasks that can be performed on different pieces of data. The data dependency of the tasks determines when they can be executed without losing any data integrity of the program. The tasks are scheduled to the available cores according to their dependency to any specific data. When a task runs and updates any data, it will notify all other tasks that their required data is ready and they will start or continue their execution on that newly available data. For the complete discussion on how this scheduling works, see [27].

1.5 Mixing the old and the new framework

As it is shown in Figure 1.1, the framework consists of several modules that work together to fulfill the user requests. There are different modules for `Geometry` and `PDE Expression` which can be used for specifying the geometry and `PDE expression` respectively. The `Problem` module lets user to combine different geometries with single `PDE Expression` to define different instances of a single PDE problem. The `Approximation` module provides functions for selecting the types of the $\phi(.)$ functions and their corresponding parameters like shape parameter ϵ. The `Operation` module contains an extensive set of functions that can be invoked for different operations for the computations required for the problem solution. Most of these functions are basic linear algebra operations that are used in many solution algorithms, like matrix-matrix multiplication. Tailoring of all these components , i.e. `Problem` and `Approximation` and `Operations`, is the role of the `Work Flow` module. Using the `Work Flow` module, user can organize sets of operations to be performed on some problems using some approximations. Therefore, user has enough flexibility to define problems and try to solve them in several different ways as well as several different approximations.

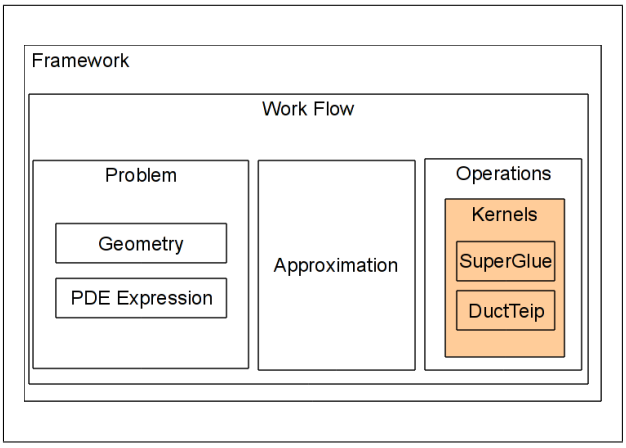


Figure 1.1: RBF Framework Modules

The actual computations that are accessible by `Operation` module are performed inside the `Kernels` module and are implemented in a generic way to be suitable for being invoked easily in different situations for different needs. The `Kernels` module uses two function libraries for executing the requested operations concurrently on all the processors that are available to the program. Since the available processors may be locally hosted by a single machine or be distributed among multiple machines accessible through a network , or both, two different sets of functions are needed to handle different kinds of requirements respectively. The **SuperGlue Library**, shown as `SuperGlue` module in Figure 1.1, implemented the required functionalities for executing the operations on multiple processors of a single machine and Distributed **SuperGlue Library** (shown as `DuctTeip` [1] module in Figure 1.1) is the main part of this project that is new to the `Framework` and implements the procedures which are required for running programs concurrently on multiple machines. More technically, the **SuperGlue Library** covers the functions for parallelization of *Shared Memory* architectures while the `DuctTeip` covers the functionalities for *Distributed Memory* architecture. In other words, this is hybrid method of parallelization in which both the shared and distributed memory architectures are used at the same time for getting the most out of the processing power available to the program.

All these features of the `Framework` would have been useless if they were provided to the

[1] **D**istributed **u**ser annotated **c**oncurren**t** **T**asks **e**xecuted **i**n **p**arallel

end user by a flat and long set of functions. In addition to the modularity, `Framework` is organized in a hierarchical way such that details of the lower layers are hidden to the upper ones. Therefore, the details of the operations and approximations, for example, are not seen by the end user of the `Framework`. More importantly, the details of the parallelization considerations and implementations are not seen by end user. In other words, the user does not have to involve with writing any special piece of code that is for parallel execution of the program.

In the next sections of this chapter, we will investigate existing methods and implementations of parallel execution of programs in hybrid models. While the most famous method for distributed memory architectures is the MPI, there are some methods which tried to extend it or even introduce a new one in such a way that be more appropriate for shared memory architectures as well. Some of these methods are targeted for Linear Algebra computations while others are more general and the following sections are categorized accordingly. This investigation aims for finding useful measures for comparing the requirements of our project to what features the methods deliver to their users. By comparing these features we will reach to a decision on an appropriate methods which suits our project needs as much as possible.

1.6 Partitioned global address space(PGAS) models

1.6.1 Shared memory(SHMEM)

The SHMEM is a library of functions provided as an application programming interface(API) and contains functionalities related to exchanging data between processors in a distributed environment. It is very similar to the message passing interface (MPI) style of programming and the programs can also be combined with or cooperate with MPI functions. Like MPI, SHMEM is in single program multiple data (SPMD) style and all processes start at the same time and run the same program. Like other programming facilities in the PGAS family, SHMEM provides remote memory access (RMA) between processors via individual function calls that allow one processor to read, write or even reference any data that resides on another remote processor. Collective and synchronization operations are also provided as they are necessary for any distributed memory architecture. There are also individual memory allocation functions that have to be used for preparing a specific amount of memory shared among processors. This API is also equipped with atomic memory operations on a remote or local data object. To let the data of any processor be accessible by any other remote process, they have to be 'symmetric' in the way that they are arrays or variables that have the same size, type and relative address on all the processors. The communication functions in SHMEM are not thread safe and when they are used in a multithreaded environment, it is the responsibility of the programmer to ensure that no such function is invoked by multiple threads simultaneously. The cache management for coherency issues in SHMEM, is intentionally left to the specific hardware on which the API is aimed to be run.

1.6.2 Global address space programming interface (GPI)

The PGAS programming model is based on existence of a (logical) global memory that is accessible for multiple processors and every process (physically) owns a specific part of it. According to this partitioning of the global data, any processor can determine which parts of the global data resides locally or belongs to other processors. The Global address space programming interface (GPI) provides this model to the application for direct and full access to a remote data location. This functionality is provided by communication and synchronization primitives. The communication overhead is minimized by overlapping the computation and communications through asynchronous communications [15]. To achieve different kinds of synchronizations, the programmer can use different *queues* of GPI for grouping the communication requests and manage them independently. If the network layer is based on Infiniband (or 10GE), GPI also provides the *passive* communication functionality in which the blocking receive operation takes no CPU time and is woken up directly by the network layer. In passive communications, the receiver does not require to know the exact sender of data since this can be identified by the arguments after the connection is established. This feature, that cannot be found in other solutions like MPI [15] is useful particularly for situations where parts of the global data have to be updated by processors. The only limitation of the passive receive operation is that it is not thread-safe, while all other functions in GPI are thread-safe either through MCTP, OpenMP or PThreads schemes [11]. Relying on the multicast feature of Infiniband networks, GPI can achieve faster barriers in collective operations among large numbers of nodes. By providing global atomic counters and operations such as fetch-add and fetch-compare-swap for them, GPI makes the load balancing and complex synchronizations more straight forward to implement. Combining all these features together with the fact that data exchange is based on one sided asynchronous communication, the GPI shows better performance than pure MPI programs in some experiments [12].

1.6.3 Global Arrays(GA)

The Global Arrays (GA) toolkit provides distributed array data structures (called "global arrays") which can be seen by programmers as shared memory programming. GA complements the message passing programming model and allows the user to combine shared-memory and message-passing styles of programming in the same program. The main operations for global shared memory are put, get, scatter and gather and can only be executed for global Arrays and not on any local memory location. These are one-sided operations that means that regardless of the remote processors who own the data these operation will complete. There is no need to use methods like polling some status variables nor calling any other GA functions to understand the completion of the operations on the remote side. The global data in GA are specified simply by providing the global map of the data portions and their locations. With this map, the shared

global data can be accessed using indexes rather than their addresses. Using GA would be beneficial in situations in which there is a need to control data locality explicitly, one-sided access to global data and also high-level operations on distributed arrays. Although, the GA use for algorithms like Cholesky factorization in which the synchronizations should be performed by point-to-point message passing is not so helpful [17].

1.7 Parallel Linear Algebra Libraries

1.7.1 Scalable LAPACK(ScaLAPACK)

ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing computers and is a continuation of the LAPACK project. Both libraries contain routines for solving systems of linear equations, least squares problems, and eigenvalue problems. The goals of both projects are efficiency, scalability, reliability, portability, flexibility, and ease of use. ScaLAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers. Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are distributed-memory versions of the Level 1, Level 2, and Level 3 BLAS, called the Parallel BLAS or PBLAS, and a set of basic linear algebra communication subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. The BLACS is a message-passing library designed for linear algebra. The computational model consists of a one- or two-dimensional process grid, where each process stores pieces of the matrices and vectors. The BLACS include *synchronous* send/receive routines to communicate a matrix or submatrix from one process to another. Since several ScaLAPACK algorithms require broadcasts or reductions among different subsets of processes, the BLACS permit a process to be a member of several overlapping or disjoint process grids, each one labeled by a context [1, 2]. The synchronous communication between processes implies that there will be occasions that some processors wait (block) for others to complete message transfers. This blocking message transfer between processors may be avoided using other types of communications that do not require parties of communications wait for completion of transfer in other parties. This is a crucial concern for overlapping computations and communications and make the processors as independent as possible from each other.

1.7.2 PLASMA

The parallel linear algebra software for multi-core architectures (PLASMA) is a software library designed to be efficient on homogeneous multicore processors and multi-socket systems of multicore processors. PLASMA can solve dense systems of linear equations and linear least

squares problems and associated computations such as matrix factorizations. PLASMA has been designed to supersede LAPACK (and eventually ScaLAPACK), principally by restructuring the software to achieve much greater efficiency, where possible, on modern computers based on multicore processors. Currently, PLASMA neither serve as a complete replacement of LAPACK due to limited functionality , nor replace ScaLAPACK as software for distributed memory computers, since it only supports shared-memory machines. LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subroutines (BLAS). Highly efficient machine-specific implementations of the BLAS are available for most modern processors, including multi-threaded implementations. The parallel algorithms in PLASMA are built using a small set of sequential routines as building blocks. These routines are referred to as core BLAS. The core BLAS routines are built in a somewhat suboptimal fashion, by using the standard BLAS routines as building blocks. For that reason, just like LAPACK, PLASMA requires a highly optimized implementation of the BLAS in order to deliver good performance. To achieve high performance on multicore architectures, PLASMA relies on tile algorithms, which provide fine granularity parallelism. The standard linear algebra algorithms can then be represented as Directed Acyclic Graphs (DAG) where nodes represent tasks and edges represent dependencies among them. The programming model enforces asynchronous, out of order scheduling of operations. In LAPACK, parallelism is obtained through the use of multithreaded BLAS while in PLASMA,it is not hidden inside the BLAS but is explicitly brought to the fore to yield much better performance. ScaLAPACK and PLASMA interfaces allow the user to provide data distributed on the cores. However, by better cache utilization,for example for matrix factorization operations, the PLASMA library gain much better performance. In the PLASMA shared-memory multicore environment, since the caches are not flushed, these libraries have the advantage to start the factorization with part of the data distributed on the caches [4].

There is also a new distributed implementation of three linear algebra kernels (QR,LU and LLT) based on the PLASMA and called Distributed PLASMA (DPLASMA). This implementation uses a generic distributed Direct Acyclic Graph (DAG) engine for high performance computing. Other than overlapping the computations and communications, DPLASMA takes advantage of task prioritizing and management of tasks on distributed architecture. In DPLASMA, a sequential algorithm is translated into some fine interrelated tasks that are distributed and executed as soon as their dependency are resolved. The DPLASMA engine schedules the created tasks in a distributed environment dynamically and gains good scalability [6]. To better fit the architecture of multicore platforms, the PLASMA library uses tile algorithms to achieve a finer task granularity together with the dynamic task scheduling [13].

Parallel programming based on the idea of representing the computation as a task graph and dynamic data-driven execution of tasks shows clear advantages for multicore processors and

multisocket shared-memory systems of such processors. One of the most interesting questions is the applicability of the model to large-scale distributed-memory systems [18].

The PLASMA in distributed environment uses a *task-based* library to replace existing linear algebra subroutines , like PBLAS, which encapsulates the dynamic scheduling of the fine grained tasks and handling the dependencies by DAGs. It targets scalability of the programs and proposes solutions that manage the dependencies without any direct cooperation of the processors in a distributed manner. PLASMA performance strongly depends on tunable execution parameters trading off utilization of different system resources. The outer block size trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size trades off memory load with extra-flops. PLASMA is currently scheduled statically with a trade off between load balancing and data reuse. Although this library suffers from non-optimized cache hit rate for large block sizes, at least with respect to BLAS, using the method for other classes of problems would be beneficial [24].

1.8 Other task based parallelization libraries

1.8.1 SMPSs framework

SMPSs, stands for **s**ymmetric **m**ulti**p**rocessor **s**uper**s**calar, is a programming environment provided by Barcelona Supercmputing Center (BSC) and is based on a source to source compiler and a run time library. In this environment, programmer writes program as in sequential execution case but specifies functions that can be run in parallel by using `#pragma task` constructs before their definitions and determine what input and/or output parameters they require. Then at run time, the library will extract data dependencies between tasks and generate corresponding graph and keep it in memory to be used throughout the program execution for scheduling purposes;[8]. There are also other constructs for marking non-task codes to be synchronized on some data that their dependencies cannot be handled at run time, e.g. `#pragma wait on (data)`; [9].

To write programs for distributed memory architecture, BSC provides `Nanos++` runtime that can be used for compiling the same SMPSs programs for cluster targets which include multiple nodes of computations. The service for supporting the clusters is still not public [7](at the time of writing this report).

1.8.2 StarPU

StarPU [10] is another software tool which helps programmer to write task based parallel programs. The programs written by StarPU can be targeted for either CPU or GPU processors without any need to extra effort for adapting the program with different types of the targets. Programmers can use StarPU either in #pragma constructs mixed with C source codes or by

using API functions for determining the tasks and their required input and/or output data and the actual functions for executing the tasks. Dependencies between tasks based on their required data can be manually set by programmer or left them for the StarPU to detect and use them as efficiently as possible. Submitting tasks to StarPU for execution is done asynchronouly and the task completeness is notified by using a call back function provided at submittion time. All the required transfering of data among tasks are handled automatically at run time and there is no need for programmer intervention. Once the tasks and data are introduced to library, the tasks are scheduled algorithmically based on their durations which have to be estimated in advance. It is also possible to use history of tasks executions or using their real durations at run time as well as using different predefined scheduling algorithms or new extended ones.

For distributed memory architectures, StarPU provides some functions equivalent to the MPI library except that they can use data that defined in StarPU instead of general buffers in MPI. Communication is two sided between nodes which means that both the sender and receiver of a given data must know this read/write relationship between each other and issue corresponding send and receive commands to another. Alternatively, the whole data can be partitioned and assigned to different nodes (owner of data) and then the whole task graph will be loaded by (distributed to) every node to let the library decide about the actual data transfers.

1.8.3 Other Task Based Frameworks

There are also some other Task Based Parallel programming frameworks that worth to mention here although they have no support of parallel programming for distributed memory architectures. Threading Building Blocks (TBB)[16] from Intel provides a rich set of classes and templates for C++ programmers for parallel execution of the program. By using the "task based programming" subset of these classes and templates, programmers explicitly define tasks and create their corresponding dependency graph using parent-child relationships between task objects. Task scheduler in TBB traverses the graph (or more precisely, the tree) of tasks while it can grow or shrink at run time. Careful use of scheduler efficiency factors is required for programmer to control the parallel execution of the tasks and gain a good performance.

In the OpenMP 3.1, new pragma directives are introduced for tasks that let programmers to define, create and control parallel execution or termination of tasks (blocks of C/C++/Fortran code). However, there is no explicit data dependency and hence no task graph structures in this programming model.[21]

In the Cilk extension to C language (either its original version [26] or its latest one for C/C++ language by Intel[3]) there are spawn and sync constructs that can be used to fork and join execution of specific lines of codes or functions. The forked pieces of program are the tasks and the scheduler uses "work stealing" for load balancing among threads but there is no explicit construct for specifying the data dependencies between tasks.

1.9 Selecting the Method for Project

After the compact description of features of different methods in previous sections, we can now decide on which features are more suitable to our project needs. Obviously, we need our program be able to run in hybrid model while avoiding missing good features or accepting extra efforts for development or even maintenance of the program. More explicitly, we need the program to:

- Run in hybrid model of shared and distributed memory architectures.
- Overlap Computations and Communications.(*asynchronous* communications)
- Make parties of communications are as independent as possible.(*passive* communications)
- Be thread-safe.(without *programmer responsibility*)
- Be suitable for Basic Linear Algebra computations.(e.g. *Cholesky* factorization)
- Be as free as possible in its own data structures (not necessarily *symmetric*).
- Have a good cache utilization in multi-core processors(not suffering from *low cache hit rate*).

It can easily be observed that none of the methods described in previous sections possesses all these features together. In other words, some methods tried to achieve some of the objectives itemized above while sacrificing other ones. However, there are some successes in these methods which can be extracted as guidelines for constructing a new method which combines all the benefits while discarding all drawbacks. These successes can be consolidated into *task-based* parallelizations in distributed memory environments which are built on asynchronous and passive communications provided by MPI library. The same concepts and algorithms for task-based parallelization in multi-core machines (which is implemented in **SuperGlue Library** in existing `Framework`) can be extended to distributed memory architectures similarly. The performance issues of shared memory architectures, like thread-safety and cache locality, will be handled by **SuperGlue Library** inside every node of networked machines; and the performance of communications between nodes is managed by distributed task library (`DuctTeip`). The next chapter of this report explains the design and implementation of this new method and its corresponding functionalities.

Chapter 2

Design

Introduction

One of the main reasons for making programs run in parallel is gaining better performance and getting the most out of the computational capacity of the underlying system. However, it does not happen automatically. Running a program in parallel introduces some overhead for example in the way that different processors are communicating. There are several ways of communication between processors which let programmers better manage and decide about the manner the communication should happen. Using these different ways, the programmer can control the program to either halt or continue execution during the communication process. This control may halt the program until the transmission and receiving are completed, or may let the program to continue and check the completeness of the communication at later points in the program. The decision about either way of controlling the program execution has a great effect on performance of parallel execution of the program. When the processors send and receive messages in a *blocking* mode, for example, both parties of the communication *waits* for another to complete the communication. The duration that the processors wait for each other might be seen as wasting time, if the waiting is not necessary. Therefore, the expert programmers try to avoid such waiting times as much as possible when they are deciding about the parallel execution of the program.

In this chapter, we explain the ways that we used for communications to avoid unnecessary waiting times among different processors in a parallel execution environment. This discussion is particularly applicable for distributed memory systems where the processors may reside on different machines in a cluster of processors networked together. Hence, the communication duration may be considerable due to message travels among different processors. Although there is no networked machines in shared memory systems, the communication overhead still applies when the messages or data are to be transferred between the local memories (caches) of the processors. However, the ways to overcome the communication overhead in these two environments are different and we will only focus on ones used for the Distributed Memory

architectures.

2.1 Previous Work

In this section, we describe how programs can be run in parallel using *tasks*. The *tasks* are pieces of software programs that can be run individually to process their input data and generate the output data. The input data for any *task* may be the output data of some other *tasks*. A *task* is executable whenever its input data are ready and also there is a free processor to run it. Therefore, using the availability of input data to *tasks*, we can determine which *task* can be run at which time. On the other hand, we need also to detect when a processor is free to execute a piece of code. This is performed by the *manager* whose responsibility is preparing queues of tasks for all available processors and whenever their input data are ready runs them by moving them to worker queues. to detect which processor is free and which *tasks* are ready to run. The *manager* keeps a list of *tasks* and a list of available processors and assigns ready *tasks* to free processors. The *tasks* whose input data are not ready remain in the list as waiting and those *tasks* which are finished will be removed from the list. To implement it practically, some other information are required such as: the states of the *tasks* in terms of *is running* , *is waiting* or *finished*; state of processors (*free* or *busy*) and data definitions and their states(e.g. *ready* or *not ready*). Using this way information, task based parallelization assigns as many *tasks* as possible to all the available processors in a machine.

In this way of parallelization there is no need for any explicit synchronization between different pieces of program because the *tasks* can be run independent of each other. Explicit synchronizations (like *wait* or *barrier*) makes the light-loaded processors go idle until their heavy-loaded siblings reach to the synchronization points. The objective of the task based parallelization is minimizing these idle times of processors caused by synchronization.

2.1.1 Task Based Parallelization

The basic objective of task based parallelization is to avoid explicit synchronizations among parallel parts of the program. Other methods of writing parallel programs provide some programming constructs (such as functions, data structures in MPI, and compiler directives in OpenMP) that can be used together with sequential programs which may be existing implementations of some algorithms. These methods have been designed for minimizing the efforts required for converting existing sequential programs to new ones that can exploit the parallel execution of the programs in modern systems. In these methods, there are cases in which it is required to synchronize all the concurrent parts of the program to continue at a specific point at the same time due to some dependencies between them. This kind of synchronization (either explicitly imposed by programmer or implicitly by compiler) introduces idle times for the processors that reach to the synchronization point sooner than others and have to wait for them.

Task based parallelization looks to the algorithm in terms of decomposable tasks with their own input and output data. Tasks are the computational parts of the algorithm that process some data and modify them or produce new data. In this view, when all input data of a task are ready it can be executed and produce its output data or change the input one. Therefore, the execution conditions of tasks can be regarded as availability of their required data. In Figure 2.1, three tasks are shown that read some data as their input and change them or produce new data as their output. This figure shows that some data may be common both in input and output of a single task, tasks may read from or write to multiple data and a single data may be shared between inputs and outputs of multiple tasks. As mentioned above, the tasks can be executed whenever all of their input data are available. **SuperGlue Library** uses this fact and runs as many ready tasks as possible concurrently. Therefore the question of how to run tasks in parallel is changed to the question of how the data availability can be recognized. The difficulty of the question stems from multiplicity and share attributes of data. To better understand how **SuperGlue Library** handles these dependencies the same tasks and data shown in Figure 2.1, are redrawn in top of the Figure 2.2 but all instances of the same data are close together this time.

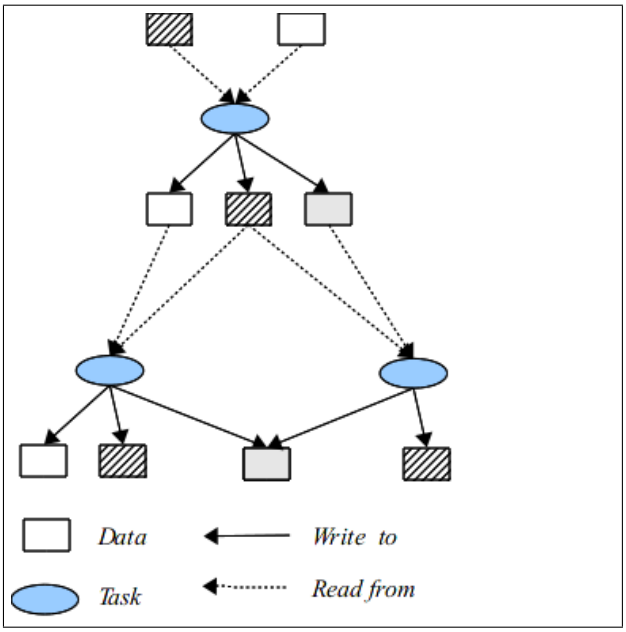


Figure 2.1: Task and Data Dependencies

As can be seen in this figure, there might be different types of accesses (*read* or *write*) to a single data during the execution of the algorithm. It is obvious that any *read*-access to a single data that happens (chronologically) after any *write* to that data should wait until the *write* finishes and then start *reading* from that data. To be able to implement these chronological dependencies, **SuperGlue Library** considers a *version* for every data which counts the number of reads and modifications that happened for that data. Any data is given a *version*, say v, and all the read-accesses to a data that are before a write-access to it are considered for that version v. All the consequent read-accesses after that, and up to the next write-access will be considered for a new version which is greater than the previous one, e.g. $v+1$, see middle part of the Figure 2.2.

By using these versions, the input data of tasks are considered together with versions and hence when any task is finished, the versions of its output data should be upgraded accordingly.

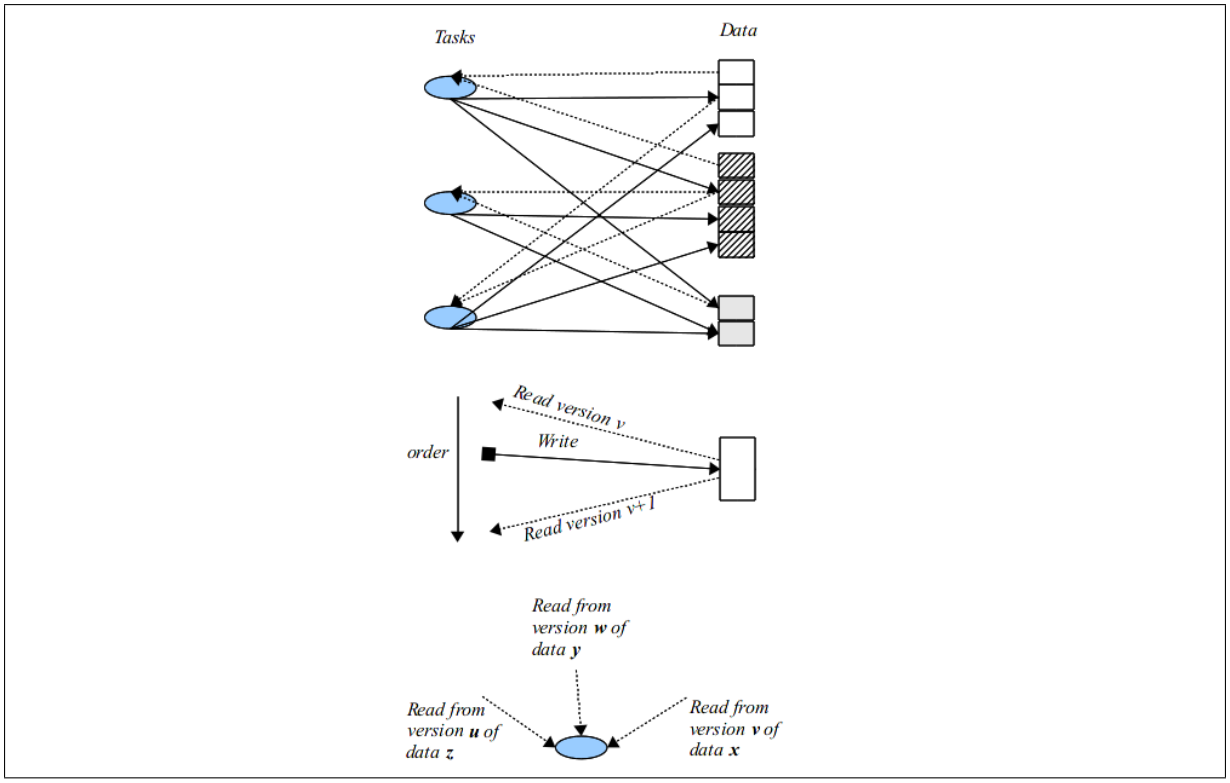


Figure 2.2: **Top**: Task and Data Access. **Middle**: Data are labeled different *Versions* after *Write* operations and all *Read* accesses after that considers a higher *Version* of the **Data**. **Bottom**:Input **Data** for every **Task** is appended by a *Version* of the **Data** as well. To check the ability of a **Task** to execute it is sufficient to check the availability of the *Versions* of its input **Data**.

Therefore any task whose input versioned-data are available can be executed, see the bottom part of the Figure 2.2, and all such tasks can be run in parallel independently. So to let the **SuperGlue Library** execute tasks in parallel, it is sufficient to give it the list of tasks with their data access annotations and the *versions* would be internally generated and managed by the **SuperGlue Library** itself. In addition to *read* and *write* access annotations, there is another one, *add*, which is used in cases when data is used for both reading and writing. These cases happen, for example, in reduction operations in parallel execution of programs where multiple parallel tasks perform some aggregate functions (such as sum, minimum or maximum) on their output data. The *add* annotation for any data access of tasks means that the result of the tasks operations will be added to the content of the data at the end of the task execution.

Figure 2.3 shows how this takes place inside the library as an example. In that figure, different versions of some data become available during different time slots (t_0 to t_4) when some tasks get finished. As new versions of data become available, other tasks get ready and can be executed. The vertical sections of that figure show the available versions of data and

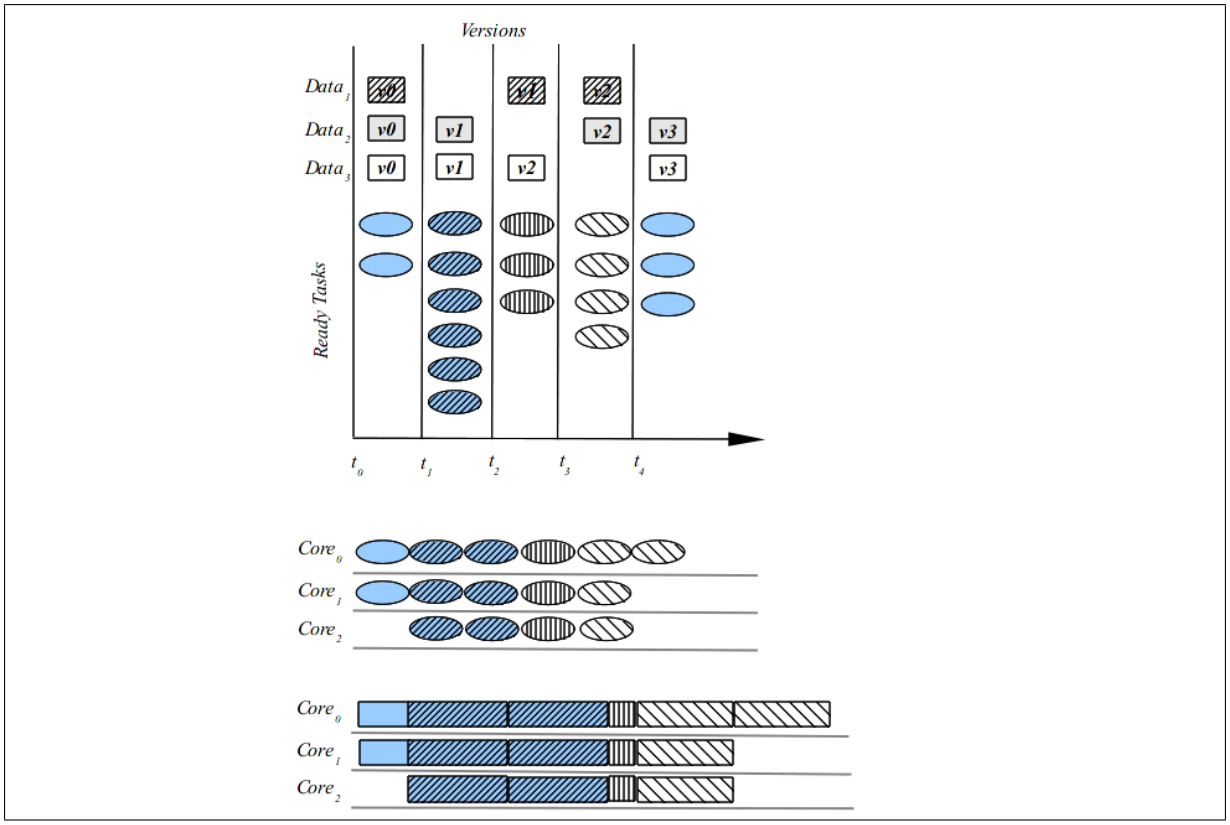


Figure 2.3: **Top**: At different times, new versions of some **Data** get available and new **Tasks** get ready to execute. **Middle**: Queue of **Tasks** for every core. **Bottom**: Actual duration of **Tasks** execution.

tasks that are ready to run using these data. Thus, in any time there might be some tasks that can be executed concurrently. The **SuperGlue Library** enqueues distributes all such tasks to the task queue for every available free processor of the host machine. For example, in the bottom part of the figure, it is shown that how these tasks can be assigned to distributed over three cores of a machine in the order which tasks get ready to execute.

Task Dependencies Formulation

After explaining how the **SuperGlue Library** works, the steps for using our task based parallelization method to run a program in parallel can be defined as follow:

- Algorithm is decomposed into tasks.
  The algorithm for the program (e.g. algorithms for vector or matrix algebra or algorithms for image processing) is translated into a series of inter-dependent *tasks* which denotes the order of the *tasks* executions.
- Input/Output data for the tasks are identified.
  To distinguish between different pieces of data that are moving between input and output

of *tasks*, data is labeled uniquely making it traceable when *tasks* dependencies to data are examined.

- Accesses to data are categorized as *read*, *write* and *add*.
  The flows of data for every *task* are determined by *read* access to data for in-flow and *write/add* accesses for out-flow.

- Dependencies to data are handled by versions of the input/output data.
  At this time that *tasks*, data *labels* and *versions* and *flow of data* are modeled, the *tasks* can be compiled into lists in which all tasks and the data flow of the algorithm are specified.

- Tasks are enqueued for execution.
  The *tasks* and data lists are delivered to **SuperGlue Library** to be enqueued in list of tasks for available processors. check which *task* is ready to run and assigns the *task* to a free processor. There exists a *task* queue for every processor and whenever the data dependencies of any task in this list are satisfied the task will be executed by that processor. cases when there are more ready *tasks* than free processors.

- Data Versions are upgraded Version of the data of any executed task is upgraded accordingly.
  After finishing a *task*, the *version* of its output data will be updated denoting that a new version of that data is ready now. Then other *tasks* which were waiting for this version of the data can be detected easily and then executed.

- Idle processors steal tasks form other busy ones Different processors can steal tasks from each other when they become idle.
  As another optimization in this method, **SuperGlue Library** lets the idle processors steal *tasks* from other busy processors for better load balancing purposes.

SuperGlue and Framework

The method mentioned in the previous section has been implemented and used in some researches [27, 24] using the Framework mentioned in section 1.5 **Existing Framework** of Chapter 1. **SuperGlue Library** [27] uses Pthread for running codes in parallel and provides interfaces for other programs to issue tasks to the library. This interface is successfully used in connecting the library to Fortran programs and has gained very good results [25]. Using this library one can prepare procedures in Fortran and pass their addresses and also their required data to the library to be checked against dependencies and then be executed. When the procedure (*task*) can be executed, **SuperGlue Library** internal functions call the procedures and pass them the data specified already at issuing time. We have used this library for parallelization in the *shared memory architecture* where multiple cores or processors are available in a

single machine. However, since our project is aimed for parallelization in a *hybrid* model (both shared and distributed memory architectures), we reused the positive features of this method for designing and implementing a new program which can act in the same way as **SuperGlue Library** but for *distributed memory architectures*. The next sections of this chapter explain the details of what ideas have been used and how this method is implemented in the project.

2.2 Distributed Memory Task Based Parallelization

In the Distributed Memory environment the processors are not necessarily hosted in a single machine and hence their cooperation may be done by data transferring among them through the network interface which interconnects the machines together. Although this kind of communication between processors is more expensive relative to the case where they are hosted in a single machine, it gives a great opportunity to parallelization due to the fact that all the communications between remote processors can be performed in parallel to their local activities by the network interface card (NIC). That is, all processors can run programs locally while they send or receive data from the remote processors in parallel at the same time. On the other hand, when a processor needs a piece of data from other processors to run a program, it has to wait for the data being received. Therefore processors have to be synchronized in different occasions of the program execution to get sure that their required data are received from remote processors. These synchronizations, like the ones mentioned in the shared semory srchitecture parallelization, may make some processors idle until all their other siblings reach to the synchronization point in the program code and communications between them complete. Thus by removing global synchronization points we can gain better performance if we can balance the parallel data transfer opportunity and the possible performance decrease due to synchronization. The objective of Distributed Task Based Parallelization is finding this balance by minimizing the idle times of processors while at the same time transferring required data to the requesting processors. This can be achieved by using the same concepts of *tasks* whose dependencies are determined by their type of data access. This time, however, the *tasks* and data may be remote and the data-flows can happen across the network.

Following the same concept of tasks and data, an algorithm can be decomposed to some *global* tasks and data which are hosted by different nodes of machines in a networked environment. Figure 2.4, shows the same tasks and data dependencies as depicted in Figure 2.1 but this time with these new global tasks and data. The boxes in this figure show the boundaries of nodes and it can be seen that reading from data may require transferring data from one node to another. These global tasks are larger tasks that can be subdivided locally when they are received in every node. Then in every node, these local and subdivided tasks and data can be delivered to the **SuperGlue Library** as mentioned before for parallel execution, see Figure 2.5.

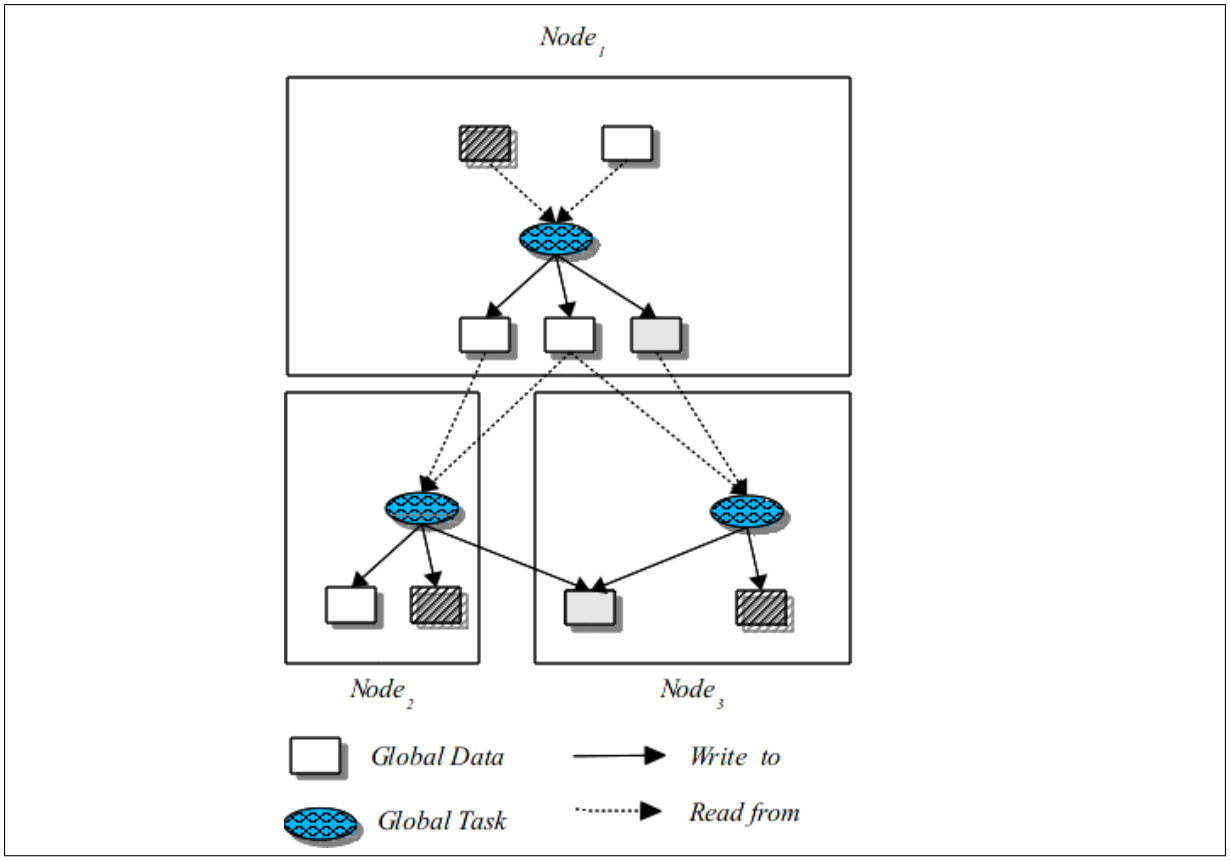


Figure 2.4: Global Task-Data Dependencies

All the required functionalities for communication of global tasks and data and interacting with **SuperGlue Library** for their parallel execution are provided as another library in the Framework, Distributed **SuperGlue Library** or `DuctTeip` in Figure 1.1, so that all the concurrent execution considerations of the program be *transparent* to the end user of the Framework. In the next section we show how we used these concepts for achieving the balance between benefits of parallel communications and costs of idle times due to synchronizations.

2.2.1 Techniques and Main Ideas

For implementing the Task Based Parallelization in hybrid (both shared and distributed memory archtictures) model we need features and facilities which let us issue data transfer requests and at the same time assign the *tasks* to processors to run. This is generally referred to as *overlapping computations and communication*, and is one of the ways to reduce the overhead of communications in the distributed memory systems. In other words, we want to make the communication as independent of computations as possible. To reach this goal, we designed our program in an abstract and general level that can be suitable for any kind of *tasks* dependencies. We have selected the message passing interface (MPI) API to provide us the aforementioned features and facilities. This API contains a rich list of functions that can be combined together

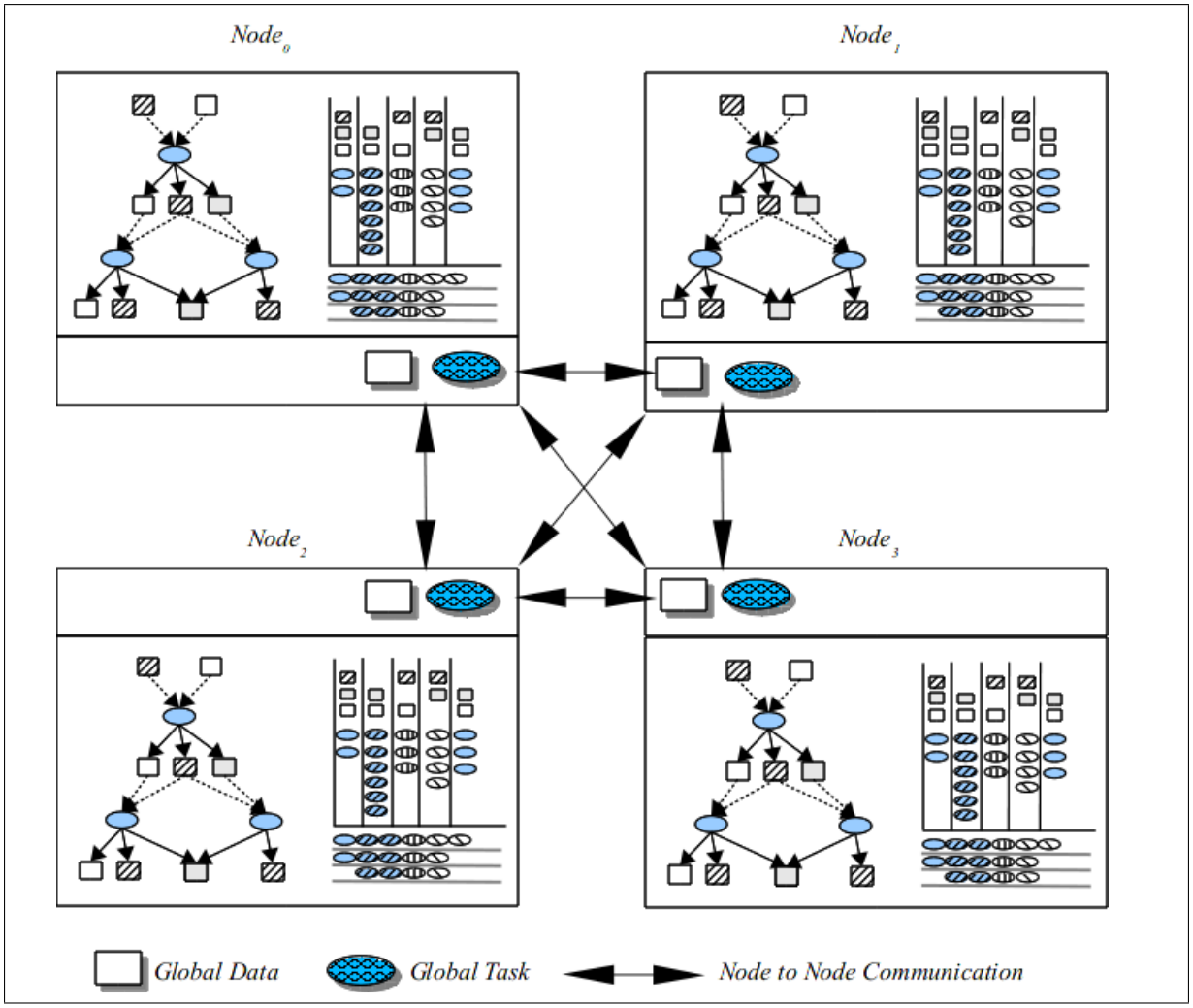


Figure 2.5: **Hybrid Task-Data Dependencies**. *Global* tasks and data are transferred across nodes and inside every node they are subdivided into *local* tasks and data. Local tasks and data in every node are delivered to **SuperGlue Library** and there they get executed concurrently.

to fulfill all our needs. We used special functions of the MPI for three major purposes each of which increases the independence of the communications from the computations. First, we used the *non-blocking* versions of the send and receive operations that gives us the possibility to issue the data transfer requests and continue computations without the need to wait for their completion. Second, in the receiving side of the communications between any two processor, we used *probing* from *any* sender instead of *waiting* for a *specific* sender. Thus, the receivers don't have to be aware of the sender(s) in advance and they will accept data as soon as they arrive. Finally, for checking whether any data transfer is completed or not (either in the sending or the receiving side of the communications) we used *testing* the status of the issued communication requests instead of *waiting* for their completions. These features are powerful enough and act as building blocks for our program. We can summarize them as follow:

- **Overlapping Computation and Communication**

– **Non-Blocking Communications**: Using `MPI_ISEND` instead of `MPI_SEND` that only initiates the communications and returns immediately back to the caller of the function.

– **Probing vs. Receiving**: Using `MPI_IPROBE` instead of `MPI_PROBE` and `MPI_RECV` to not wait for receiving data from a specific processor and also do this in a non-blocking mode of communication in which the function immediately returns if no data is received and let the caller continue its computations. To let the function probe for received data regardless of the sender(s), we pass the `MPI_ANY_SOURCE` to it as the source parameter. Although there is a similar argument for this function for specifying which kind of messages are we interested in to probe for, we did not use it since we need to distinguish different kinds of messages and follow a specific sequence of events accordingly.

– **Testing the communication vs. Waiting**: Using the `MPI_TESTxxx` instead of `MPI_WAITxxx` to not wait for completing the requested communication and just testing whether it is finished or not and then continue the computations. Also using the `MPI_TESTANY` instead of `MPI_TESTALL` to return immediately when any requested communication is completed and not necessarily all of them. This makes different communications independent of each other.

With this level of generality and flexibility in the communication layer of our program, we can proceed to handle the *task* dependencies and define various components that are involved in the implementation.

2.2.2 Design of Framework Components

The starting point of the design is answering the question of how can we implement the *task* dependencies in our program. The *data* are as important as *tasks* in our method particularly when the flow of *data* between *tasks* should happen remotely via the network. In addition, the *data* can flow from one *task* to multiple *tasks* as their input and also multiple *data* can flow from multiple *tasks* as their outputs to a single *task*. Thus, there is many-to-many relationship between *tasks* and *data* Since the communications between processors are performed asynchronously, the order of sending and receiving data are not haappened deterministicaly and any processor that needs a specific data requests it from its owner and whenever a data gets ready in a processor it will be sent to all its requesters. To implement such a relationship we need also to another component, named *listener*, that handles the flow of *data* between *tasks* when they require them as input. In other words, any *task* that needs any *data* that resides on a remote processor, creates a *listener* for that *data* and sends it to its remote owner. It is like the publisher-subscriber design pattern in which the *tasks* are the subscribers and the owner plays the role of publisher and the *listener* is the subscription.

Using these three components and following the same scheme as of the **SuperGlue Library** mentioned in section 2.1.1, we can define the static structure of the *tasks* using the following objects:

- **Task**. Encapsulates all information for the *tasks* including: status, the host machine that will run them, data requirements and their access types.
- **Data**. Holds information like: unique label of the data, host of data,version numbers and status of the data.
- **Listener**. Determines which processor requires what data from which other processor.

Therefore, the *tasks* dependencies can be compiled into lists of these three objects which are related to each other according to the flows of data between *tasks*.
The dynamic behavior of the program in our method can be explained more efficiently by considering the following steps :

1. The distribution pattern of the *tasks* and *data* over the processors is decided.
2. Hosts for all **Task** and **Data** objects are set accordingly.
3. All List of **Tasks** are is distributed over the network to their corresponding hosts.
4. Every processor that receives **Tasks**, owns them and processes them to find out which **Data** objects they it needs.
5. For every **Data** object which is not owned by the **Task** owner, a **Listener** object is created and sent to the owner of that **Data**.
6. Every processor that receives **Listeners** keeps them until their requested **Data** gets ready. Then it sends the **Data** to the requesting processor.
7. Every processor that receives **Data**, saves it and sets the status of the **Data** to ready and two procedures have to be performed: proceed in parallel: 1) The **Tasks** that are dependent to this **Data** are checked that whether all their required input **Data** are ready or not. If yes, they get scheduled to run. 2) All the **Listeners** that have been listening to this **Data** get *activated*, that is , this new ready **Data** will be sent to the requesting processors.
8. When any **Task** gets finished, the status of its output **Data** are changed to 'ready'.
9. The steps 7-8 repeat until all the **Tasks** get finished and all **Listeners** sent their **Data** to their destinations.

The first two steps are step is problem dependent and are is explained in more details in Chapter 5. Figure 2.6 on page 36 illustrates a complete example of remaining general steps. In this figure three sample nodes of networked machines are shown that are communicating the global tasks and data with each other. $Node_1$ does not need any global data and produces the global data x using its own local data (not shown in the figure). $Node_2$ reads the global data x and produces global data s. $Node_3$ reads as its input both the global data x and s and produce some other global data. There are icons in the figure for every node which are smaller versions of the Figure 2.3 and show that inside every node the global tasks and data are subdivided into local tasks and data and run concurrently using the **SuperGlue Library**. A sample execution of the above mentioned steps can be seen in the figure as follow:

- Three types of global tasks are distributed over three nodes.
- Tasks are checked upon arrival for their data dependencies:
  - The global **Task** in $Node_1$ needs only local data, then it will be delivered to the **SuperGlue Library** for execution.
  - The global **Task** in $Node_2$ needs global data x from $Node_1$, then it generates a **Listener** for x and sends it to $Node_1$. (Shown by dashed arrows).
  - The global **Task** in $Node_3$ needs global data x from $Node_1$ and s from $Node_2$. Thus it generates **Listeners** for these data and sends them to owners of the data (i.e. $Node_1$ and $Node_2$ respectively).
- Nodes that receive **Listeners** store them locally for future invocations.
- $Node_1$ finishes its task and thus the global data x gets ready. The corresponding **Listeners** are woken up and the data x will be sent to the remote listening nodes (i.e. $Node_2$ and $Node_3$).
- Nodes that receive Receivers of global data checks their own tasks to find that whether all their input data of any of the local tasks are ready or no.
- The task in $Node_2$ can start running using the received global data x. The task in $Node_3$ must still wait for receiving the global data s as well.
- The global task and global data x in $Node_2$ are subdivided and given to **SuperGlue Library** and are executed there concurrently.
- After finishing the task the global data s in $Node_2$ gets ready.
- The Listener of s in $Node_2$ is waken up and global data s is sent to the remote listening node ($Node_3$).

- The global data s is received by $Node_3$ and then tasks are checked there for possible execution.
- Task of the $Node_3$ can run and thus is subdivided and delivered to **SuperGlue Library** for concurrent execution.

These steps implicitly show two more required objects for controlling the execution of the program. One of them handles the underlying communications between processors in the manner which described so far. The other one is responsible for checking and running *tasks*. These two objects can be defined in this way:

- **Mailbox** object: acts as a container of the messages being passed between processors. Every message is put into the **Mailbox** for being delivered to the destinations, and whenever any message is received from other processors the **Mailbox** will detect it and notify corresponding objects to react.
- **Scheduler** object: checks the **Task** objects to find whether they can run or not. If any **Task** can run it will be subdivided and delivered to **SuperGlue Library** for execution on the available local processors.

In addition to these objects, one can observe that the steps also imply an intrinsic state machine in the dynamic behavior of the method. Every object will operate differently in different situations of the program. All the communications are being made asynchronously between processors and various objects may have to react to some specific *events* happening in **Mailbox**. It is also possible that a single action in an object causes multiple reactions in other ones. This kind of interaction between objects by *events* or *notifications* based on *states* of the objects, gives us an important clue to consider a *state-transition* mechanism for implementing the interconnections between various objects. That is, in an abstract description, every object is in a single *state* among all its possible ones, and based on which *event* happened performs something and goes to another *state* while at the same time *notifying* other related objects to react upon its new *state*. For example some important *states* and *events* can be listed as follow:

- Important *States* of the **Data** object are: *ready to read, ready to write, wait for task finish*
- Important *States* of the **Task** object are: *waiting for data, ready to execute*
- Important *States* of the **Listener** object are: *waiting for data, activated*(its data received)
- Important *Events* of the program are: **Task**/**Data**/**Listener** *received, sent* and **Task** *finished*.

All these steps can be summarized as Algorithm 1 **Distributed Task Execution** including a general syntax of **Object**.MethodName([parameter1 | , parameter2 | , ...]). Generally this

Algorithm 1 Distributed Task Execution

```
1: while ¬ Tasks.Finished() do
2:   event ← Mailbox.CheckInbox();
3:   if event = Task Received then
4:     Tasks.Add(received task);
5:     Scheduler.CheckTaskDependencies(received task);
6:   else if event = Listener Received then
7:     l ← Listeners.Add(received listener);
8:     if Data.GetStatus(l.Data) = ready to read then
9:       Mailbox.Send( l.RemoteNode , Data.GetData(l.Data));
10:    end if
11:  else if event = Data Received then
12:    Data.SetStatus(received data,ready to read);
13:    Scheduler.CheckTasksForReadyData(received data);
14:    Listeners.CheckForActivation(received data);
15:  end if
16:  CheckStatusAll();
17: end while
```

algorithm checks the global communications as long as there exists any (line 2) and performs their corresponding reactions until all tasks are finished (lines 1,17). It checks whether any **Task** is received or not and if *yes* it adds the received **Task** locally and check its data dependencies using **Scheduler**.CheckTaskDependencies() procedure (lines 3–5). If *no*, checks whether any **Listener** received or not, if *yes* it just adds it locally for future references(line 6–7). If the requested data by the listener is already available, then the data will be sent immediately to the remote listener (lines 8–10). Otherwise, if any **Data** received, first it sets its status to *ready to read* and then checks which **Task** can be run by this new received **Data** using **Scheduler**.CheckTasksForReadyData() procedure and checks which **Listeners** are waiting for this **Data** using **Listeners**.CheckForActivation() procedure (lines 11–15). After processing all *events*, the status of all objects will be checked to find whether any further state transitions should happen or no (line 16).

Algorithms 2–7 show inside the procedures mentioned above. CheckTaskDependencies() procedure of **Scheduler** (Algorithm 2) sends **Listeners** for all remote input data of the *task* given as input (lines 1–8) and then checks that whether the *task* can be executed or no using **Scheduler**.CheckTasksForRun() procedure (line 9). For all other types of data accesses of the *task* and all the local input data there is no need to check the dependencies for the *task*. **Scheduler**.CheckTasksForReadyData() procedure (Algorithm 3) checks all **Tasks** that need the given *data* can be executed or no using **Scheduler**.CheckTasksForRun() procedure.

Algorithm 2 **Scheduler**.CheckTaskDependencies(*task*)

```
1: for all d in task.Data do
2:   if d.DataAccess = Read then
3:     if d.Host ≠ local Host then
4:       l ← Listeners.CreateListenerFor(d);
5:       Mailbox.Send(l);
6:     end if
7:   end if
8: end for
9: Scheduler.CheckTasksForRun(task);
```

Algorithm 3 **Scheduler**.CheckTasksForReadyData(*data*)

```
1: for all t in Tasks do
2:   if data ∈ t.DataList then
3:     Scheduler.CheckTasksForRun(t);
4:   end if
5: end for
```

Scheduler.CheckTasksForRun() procedure which is shown in Algorithm 4, checks that whether all input data of the given *task* are *ready to read* or no and all the output data of the *task* are *ready to write* or no; and if yes to both, it executes the *task*. In this checking also only the *read* access types to data are checked to be ready or not and other types of data accesses have no impact on the *task* execution. Algorithm 5 shows the **Listeners**.CheckForActivation() procedure that checks which **Listeners** are waiting for the given *data* and sends the *data* to remote listening nodes and after that the listener can be removed from the list of **Listeners**. In CheckStatusAll() algorithm shown in Algorithm 6 on page 35, all the **Listeners** and **Tasks** are checked again for reacting to state changes of other objects (lines 1–3 and 11–13 respectively). State of **Data** objects whose states are *ready to read* and all their **Listeners** are activated will be changed to *ready to write* to let depending tasks to write on them (lines 5–9).

In Algorithm 7 the **Tasks**.Finished() function is shown which checks whether the program can be finished or no and returns *true* or *false* respectively. If there is no local *task* in **Tasks** list and there is no global **Task** in its way from other nodes(detected by calling **Tasks**.IsFinalTaskReceived() function), then the program can finish (lines 13–15). Final Task is a specific predefined **Task** which is used for signaling the possibility of ending the program, more details on this will be given in Chapter 3. However, as long as there are some local finished *tasks* (lines 1–12), their output **Data** have to get *ready to read* status (lines 3–5) and thus any other dependent **Tasks** and **Listeners** have to be rechecked again for this new ready **Data** (lines 6,7). All the local *tasks* that are finished and rechecked will be removed from the local

Algorithm 4 **Scheduler**.CheckTasksForRun(*task*)

```
1: can execute = TRUE;
2: for all d in task.DataList do
3:    if d.DataAccess = Read then
4:       if d.Status ≠ ready to read then
5:          can execute = FALSE;
6:       end if
7:    end if
8: end for
9: if can execute = TRUE then
10:   Scheduler.Run(task);
11: end if
```

Algorithm 5 **Listeners**.CheckForActivation(*data*)

```
1: for all l in Listeners do
2:    if l.Data = data then
3:       Mailbox.Send( l.RemoteNode, data);
4:       Listeners.Remove(l);
5:    end if
6: end for
```

list of **Tasks** (line 10).

The details and techniques for performing the operations mentioned above will be presented in the Chapter 3.

Algorithm 6 CheckStatusAll()

```
1: for all l in Listeners do
2:   Listeners.CheckForActivation(l.data)
3: end for
4: for all d in Data do
5:   if Data.GetStatus(d) = ready to read then
6:     if ¬ Listeners.IsPendingFor(d) then
7:       Data.SetStatus( d, , ready to write);
8:     end if
9:   end if
10: end for
11: for all t in Tasks do
12:   Scheduler.CheckTasksForRun(t)
13: end for
```

Algorithm 7 **Tasks**.Finished()

```
1: for all task in Tasks do
2:   if task.Status = Task Finished then
3:     for all d in task.DataList do
4:       if d.DataAccess = Write then
5:         d.Status = ready to read;
6:         Scheduler.CheckTasksForReadyData(d);
7:         Listeners.CheckForActivation(d);
8:       end if
9:     end for
10:    Tasks.Remove(task);
11:  end if
12: end for
13: if Tasks.IsEmpty() ∧ Tasks.IsFinalTaskReceived() then
14:   return TRUE
15: end if
16: return FALSE
```

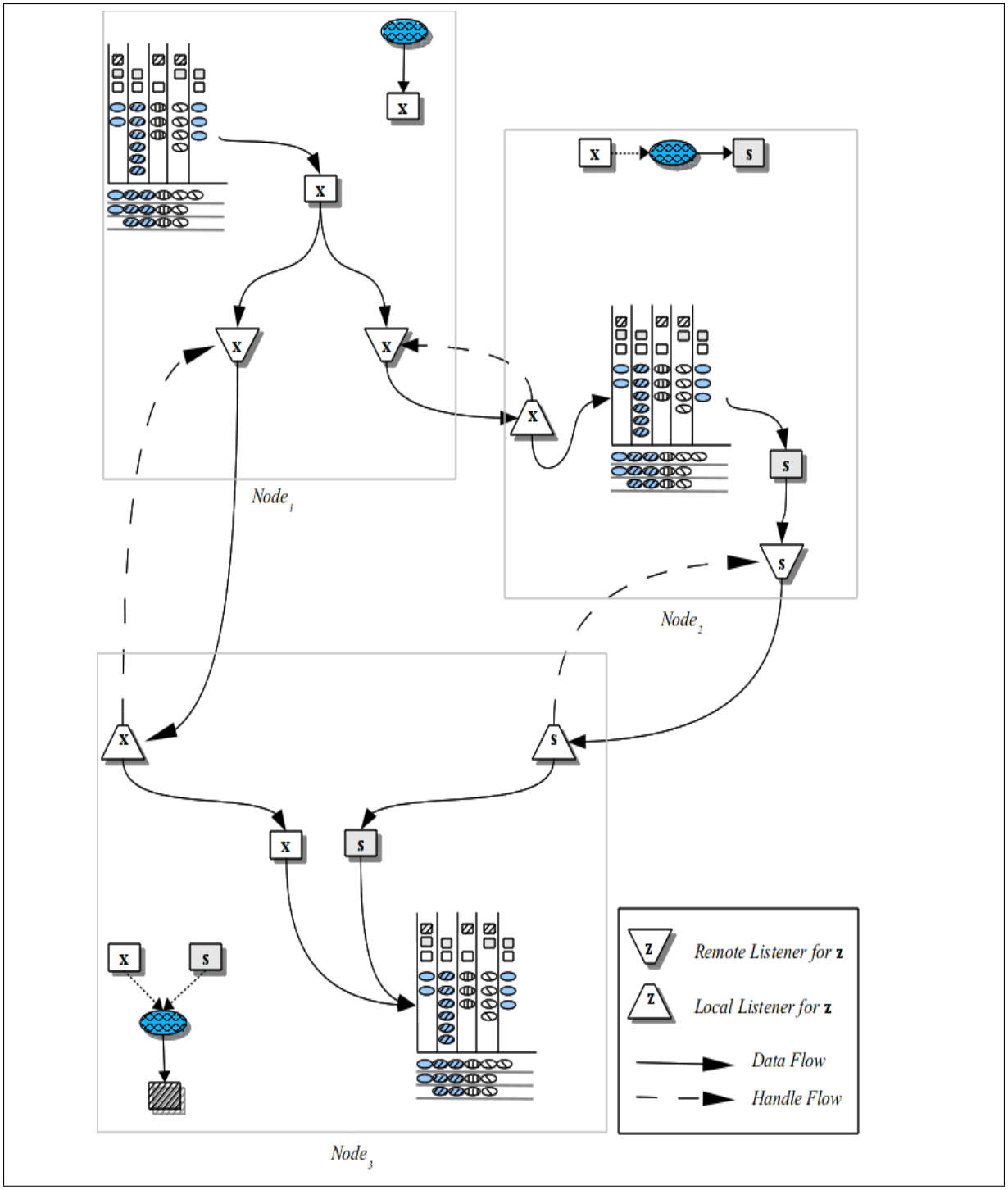


Figure 2.6: Three nodes communicate global data x and s for their global **Tasks** through **Listeners**. When data is received locally, the global Task and its data are subdivided into smaller local tasks and data and then delivered to **SuperGlue Library** for parallel execution on available local processors. In every node, there may exist *multiple* **Listeners** : for a single data requested by multiple remote nodes($Node_1$), for sending and receiving data ($Node_2$) and for multiple data ($Node_3$).

Chapter 3

Design Details

Introduction

In this chapter, more detailed discussions about the design model of the program are provided. To keep the general discussions in Chapter 2 more understandable some details are deferred here. The important details are related to the state-transitions of the objects based on different events and the way different nodes can be coordinated together for entering or leaving some stages of the program. There are at least two stages for any program: 1) when sending **Tasks** to nodes is completed, and 2) when all **Tasks** are finished and the program can terminate. There may be other stages between these two ones, for example one that all listeners are distributed and nodes can be sure that there is no listener on its way and they can deallocate memory for any unused **Data**. So, entering into this *data-cleaning* stage requires coordination with all other nodes.

3.1 Communication

As stated in Chapter 2, the communications are performed in non-blocking mode which means all send and receive operations are performed asynchronously. In this method, all the send operations are actually *requests to send* and their completion is not recognized immediately so it is necessary to check the receipt of messages by explicitly receiving parties, which we call them Acknowledgment in our program. Therefore, for each send of *data*, *task* or *listener* we will have *data ACK*, *task ACK* and *listener ACK* events respectively showing that the corresponding requests are completed. These events together with other ones that shown in Table 3.1 cause the status of **Task**, **Data** and **Listener** objects to change from one to another. Whenever the status of any object changes, other objects may need to be notified. Table 3.1 summarizes these relationships between objects, events, state-transitions and notifications in the program. This table shows that after some events happened for objects, which operations of them have to be performed that cause their status change from one to another and finally which other objects

have to be notified. For example reading the first row of the **Data** section in the table says that will be read as: when a **Data** object detects a *Task Finished* event, *if* it is output of the finished task its **Get Ready** operation will be executed and changes its status from **Wait** to **Ready (to Read)** and notifies **Mailbox** and **Listener** objects for probable sending of new available data.

3.2 Coordination

There are some cases in the program that the communicating nodes have to be coordinated together to fulfill some specific criteria. One of these cases, for example, is when they can finish the program and terminate. This case is mentioned first in Algorithm 7 on page 35 where the **Tasks**.Finished() function has been shown. Here in this function, one of the conditions for finishing the program is checked by IsFinalTaskReceived() of **Tasks** that returns TRUE if no global task is remained to be received. To detect this condition, program sends a special predefined task, say Final Task, to all nodes signaling that there will be no more tasks to be sent/received. Only after Final Task is received, nodes can finish the program if their local list of tasks gets empty (line 13 in Algorithm 7).

As another example case, the same may happen when there are some listeners in their way to receive a node which decided to finish the program. Therefore, that node is not allowed to terminate until gets sure that no incoming listener is remained otherwise the data request from other remote listening nodes will be left unsatisfied. Generally speaking, there may exist cases when nodes *race* each other to reach or pass a certain stages of the program. For correctness of the program, it is required to coordinate the nodes for entering or leaving these stages. This coordination between nodes is implemented by introducing a mechanism of *seeking* for specific predefined messages in every node, like Final Task, to allow/disallow the program to enter into a stage or to leave it. In addition to Final Task message in this category, there are other instances of such messages like Final Listener that is used to allow node to deallocate memories holding Data objects when all corresponding local listeners are cleaned.

Implementing this mechanism is simply done by sending these special messages and probing their corresponding ACK messages from other parties. Sending these messages means asking for permission of doing something and receiving acknowledgment means positive response. In this mechanism, when a node wants to enter/leave a stage sends specific message to others and won't enter/leave until receives acknowledgments (positive responses).

Table 3.1: State Transitions

Object	**Events**								**When**	**Operation**	**From**	**To**	**Notify**				
	Task Received	Task Started	Task Finished	Task ACK	Data Received	Data ACK	Listener Received	Listener ACK					Mailbox	Task	Scheduler	Data	Listener
Task										Create Task	Cleaned	Init	√		√	√	√
	√									Add	Cleaned	Init	√		√	√	√
		√									Ready	Run					
									Is Local	Clean	Finished	Cleaned					
			√							Output Data Ready	Run	Finished	√			√	√
				√					Is Sent to Others	Clean	$-^*$	Cleaned				√	√
					√				∀ Input Data = Ready	Check For Run	Wait	Ready	√		√	√	√
					√				∃ Input Data ≠ Ready	Check For Run	Init,Wait	Wait					
Data			√						Is Output Of Task	Get Ready	Wait For Task	Ready To Read	√				√
					√					Get Ready	Wait For Receive	Ready To Read	√	√	√		√
							√		New Data	Add	–	Init					
									Is Local	Add	Cleaned	Init					
										Check Task Dep.	Init	Wait for Task/Receive					
									No Pending Listener	Check Status	Ready To Read	Ready To Write		√			
									No Pending Task	Clean	Ready To Write	Cleaned					
Listener							√			Add	Cleaned	Init					
									Is Local	Add	Cleaned	Init	√				
								√		Check Status	Init	Wait For Data					
			√						*data* = Task Output *data*	Check Status	Wait For Data	Active	√				
					√					Check Status	Wait For Data	Data Received		√	√	√	
						√				Check Status	Active	Data Received					
										Clean	Data Received	Cleaned					

* : '–' *means any state*

Chapter 4

Experiments

Introduction

In this chapter we will show details of the implementation by following the required steps for solving a given problem. The problem here is assembling the RBF Matrix for a PDE problem over a set of scattered points. We first give a definition of the problem and its solution and then enter into more details of decomposing the solution into some global and local tasks that help us to run the method on parallel processors in a hybrid model using both shared and distributed memory architectures.

4.1 Problem Definition: RBF Matrix Assembly

In the RBF method mentioned in Section 1.4 **Introduction to RBF**, the vector **x** contains data points and the objective here is to compute the $\varphi(||x_i - x_j||) \quad \forall i, j$; where $\varphi(\cdot)$ is the *basis* function and $||.||$ is the *distance* of two points. Computing the basis function between all pairs of points results in a matrix which is the coefficients of a linear system of equations. The construction (computation) of this matrix is the problem that will be solved using parallel execution of tasks in the **hybrid** model.

Figure 4.1 on page 45 shows the steps of the solution in a single view. In this figure it is shown that first geometry of the problem is discretized in x and y dimensions (generally it can be N_d dimensions in the program) and the resulting points will be held in a vector **x**. The point vector **x** is divided evenly into N_n (number of nodes of networked machines) partitions, one partition for each node. All nodes are responsible of computing the *distance* of points and $\varphi(\cdot)$ function for their own partitions P_i and all other ones pairwise. In other words, node$_i$ for example computes the distance and $\varphi(\cdot)$ function of its P_i with all other $P_j, j \in [0, N_n)$ partitions from other nodes and saves the result as the $block_{ij}$ of the final RBF Matrix. Therefore, all partitions have to be distributed to all nodes. This can be performed in various ways discussed

later in this chapter. When partitions P_a and P_b get ready in a node, the node will subdivide them into smaller blocks(B_i and B_j in the bottom part of the figure) and create local tasks for computing the *distance* and $\varphi(\cdot)$ functions reading these blocks. The result of *distance* computations are *written* in a local sub-matrix of D_{ij} which is then *read* by local task of $\varphi(\cdot)$-computations and then written to R_{ij} of Result Matrix. These *write* and *read* dependencies are handled automatically and transparently by the **SuperGlue Library**. When all the global and local tasks are finished the **row**-blocks (all columns of partitioned adjacent rows) of resulting RBF Matrix will be available in every node.

Algorithm 8 Assembling The RBF Matrix-Sequential Version

1: **for all** x_i, x_j in **x do**
2: $d_{ij} \leftarrow (x_i^2 - x_j^2)^{1/2}$;
3: $M_{ij} \leftarrow \varphi(d_{ij})$;
4: **end for**

4.1.1 Implementation

The sequential version of the solution for the RBF Matrix assembly problem is shown in Algorithm 8. For implementing this algorithm in a distributed task based parallelization method, we need first to partition the data points into available number of nodes. This implementation is summarized in Algorithm 9 whose preconditions are existence of the partitions. This algorithm uses **DuctTeip** object that provides the functionalities of the **DuctTeip Library** through its methods. The lines 1–23 of the Algorithm 9 on page 46 repeat the computations for all pairs of the partitions. For every pair of partitions, a global task is generated (line 3) and added to the library(line 21) for later delivery(line 25). The destinations of the global *tasks* are determined in line 4 of the algorithm which sets the **Host** of the *task* to the index of the main partition, i.e P_i. Other patterns of distribution for global *tasks* will affect only this line by replacing the $node_i$ right-hand-side to any other value determined by the specific patterns. For every partition we have to tell **DuctTeip** how the global *tasks* access the partitions and from which node they have to be requested (lines 6–14). Decisions about pattern of data communications between nodes are reflected only in lines 7 and 12 where the **Host** of the global data are determined. Later in this section, we will show various possibilities of these patterns. The result of the computations for any two pair is *written* to a local data by setting the **Host** of the data to wherever the *task* resides (lines 16–19).

When all the global tasks are created, their version can be set for their different access types to global data by simply traversing the list of tasks using the **DuctTeip**.CreateVersions() function in line 24 of Algorithm 9. Then all the tasks will be distributed to their destinations which are determined in their **Host** using the **DuctTeip**.DistributeTasks() function in line 25.

Finally when all global tasks are sent to their destinations, by calling the **DuctTeip**.Execute() function the main loop of the program also starts and the communications between nodes are handled in the same way as specified in the Algorithm 1 in Chapter 2. In addition to communications, the main loop of the program is also responsible of computations which are encapsulated in **Scheduler**.Run() function in the Algorithm 1.

The execution of RBF Matrix assembly method, is performed locally in every node and implemented in the **Scheduler**.Run() function as specified in the Algorithm 10 on page 47. The preconditions of this function enforce that the first two DataAccess members of the given global task are the two partitions for which the RBF Matrix have to be computed; and the number of subdividing blocks inside every partition and the block sizes are known before calling this function. This function extracts the pair of partitions from the global task (lines 1,2). Then for every two subdivided blocks (indexed by i and j in the range of number of blocks N_b) inside every partition, it creates some local tasks together with their data access types to the blocks (lines 3–35). The block B_i is mapped to the ith slice of the partition P_i and the same for B_j of P_j (lines 5,6). For every pair of blocks two local tasks are created, one for *distance* (mathematically shown as $||.||$) and another for $\varphi(\cdot)$ computations (lines 7 and 23, respectively). The local task of *distance* has to *read* from blocks (lines 10,14) and *writes* the result into a local matrix D_{ij} (lines 17,18). The local task of $\varphi(\cdot)$ has to read from the result matrix of the *distance* task (line 25 where d_t still holds the D_{ij} created earlier at line 17) and *writes* the final result to matrix R_{ij} (line 29). Then these two local tasks are delivered to the **SuperGlue Library** for execution in parallel (lines 21,32). Down in the **SuperGlue Library** when it is the time for running these tasks, corresponding actual functions provided as kernels inside the Framework will be executed.

4.2 Executing Program

After implementing the solution as explained in previous sections, the program has been executed with different values of parameters to investigate its behavior in terms of scaling and performance.

4.2.1 Task Execution

One of our main objectives by Task Based Parallelization is to remove any unnecessary idle time of processors due to using explicit synchronization between different pieces of code. So, to assess achievement of this goal we instrument the code for measuring the duration of various events during the program execution. Among these events, starting to execute a task and finishing it are the most important and relevant ones by which we can observe the actual execution time of the tasks and the idle time of the processors between any two consequent tasks. The results of these experiments are shown in Figure 4.2 that shows the execution time of tasks running

on eight nodes. The scheduled plots in these figures have separate sub-plots for every node and inside each node (sub-plot) the cores are shown along the y-axis. The tasks are shown by boxes whose lengths show their execution duration and whose colors distinguish between two types of tasks in our program (i.e *Distance* and $\phi(\cdot)$ calculation). The starting time of each sub-plot is the time of the first Data-Ready event for the corresponding node and it is the time that some tasks also get ready to run. The vertical black lines at the right edge of the boxes show the finish time of the global tasks (which have been sub-divided into these local tasks).

Figure 4.2 shows how the tasks are executed one after another for the mentioned computations. Figure 4.2a depicts the durations for solving a problem of 8000×8000 matrix assembly by 8 nodes of 7 cores each. It can be seen in this figure that there are very small/few gaps between finishing a task and starting another although in our program there are dependencies both on global data and between several local tasks. That is, the computations have been performed while global data (and tasks) have been communicated between nodes without explicit waiting for communication to complete. This is true while there is sufficient local computation work to overlap with the communication work of sending or receiving global data. If the problem size, for instance, is not too large then the required time for the computations will be shorter than the communication time. This can be observed in Figure 4.2b where the global matrix size is 800×800 and every node computes matrices of size 25×25 (800 partitioned over 8 nodes and there sub-divided into 4×4 blocks). Here, since the computations complete faster than the communication then threads will be idle waiting for their required input data to be received from remote nodes. The elapsed time for communication consists of times for network latency and message transfer. The latency time depends only on underlying network characteristics (independent of message size) while message transfer time is almost a constant multiplied with message size. Therefore, by decreasing the problem size (hence the message size) time for transferring messages decreases as well but the latency time remains the same. So regardless of the message size, there is a lower bound on communication time whereas the computation time will decrease by decreasing problem sizes. Therefore for small problems there may be times when the computations finish much sooner than their data communication. Specifically for the machines used in these experiments; Intel Xeon E5420; the minimum sizes of the input data after which this behaviour can be seen are approximately 180 for 7 cores,150 for 5 cores and 30 for 1 core. These values could stem from the processor's L1 cache size which is 16K and can contain completely a 40×40 matrix of double precision elements plus the required input vectors of points. So, when the input data of the tasks are small enough to fit in the L1 cache of the cores the cache hit ratio will be higher and we expect that the task's execution time be much shorter due to less accesses to main memory.

4.2.2 Speed Up

We have designed and executed experiments for measuring the speed up of the program in which we provide more processors to the program while keeping the total problem size fixed and compare the execution time relative to the one-core configuration. Figure 4.3 on page 49 shows the results of these experiments. Together with the experiment results the ideal speedup; when 100% of the program is parallelizable; is shown as dashed line. The program is executed for computing RBF matrix of size 8400×8400 and the results are the best ones among different nodes and cores configurations which are shown in boxes in the figure for some sample points. In this figure we can observe that up to almost 16 cores the speedup of the program is aligned with the theoretical speedup line and thereafter for more cores it starts deviating from it. For the last point in the graph, the speedup is 52 out of 56 cores or 92% which is very good.

4.2.3 Scale Up

To measure the scale up performance of the program, we have investigated both the weak and strong scaling behavior of the program. In weak scaling, the problem size for every core is kept the same while more cores are provided to the program whereas in the strong scaling the total problem size is fixed and more cores made available to the program. In both experiments the relative execution times to the one-core configuration are measured. Figure 4.4a on page 50 shows the results of strong scaling experiments. For these experiments the program executed for a matrix size of 8400×8400 and the execution time of the program is normalized to the one-core instance of the program. In this figure, the curves are categorized in terms of the number of cores per node configurations and for each configuration the perfect scale up lines are also shown as dashed lines. As can be seen from this figure, the scale up behavior of the program for 1- and 3-cores per node configurations are completely aligned with the perfect ones. For the 5- and 7-cores per node configurations the scale up is not perfect but is as high as 97% and 95% of the ideal for the 40 and 56 cores, respectively.

Figure 4.4b on page 50 shows the weak scaling results of the program execution. In this experiment, the work per core is fixed at one million elements or computing a 1000×1000 matrix. The figure shows the speed of program relative to the one-core configuration. Since the work per core is fixed for all configurations of this experiment, it is theoretically expected that the relative speed be 1. That means the time of computation of a fixed work by one core should be the same as computing n works of the same size by n cores. This figure shows that for all configurations, the scale up improves by increasing number of cores and for 1- and 3-cores per node configurations the scale up is better than 1 which means the scale up is magnified by more than number of cores. For 5- and 7-cores per node configurations, the scale up is improving by adding more cores and eventually is close to the perfect value of 1.

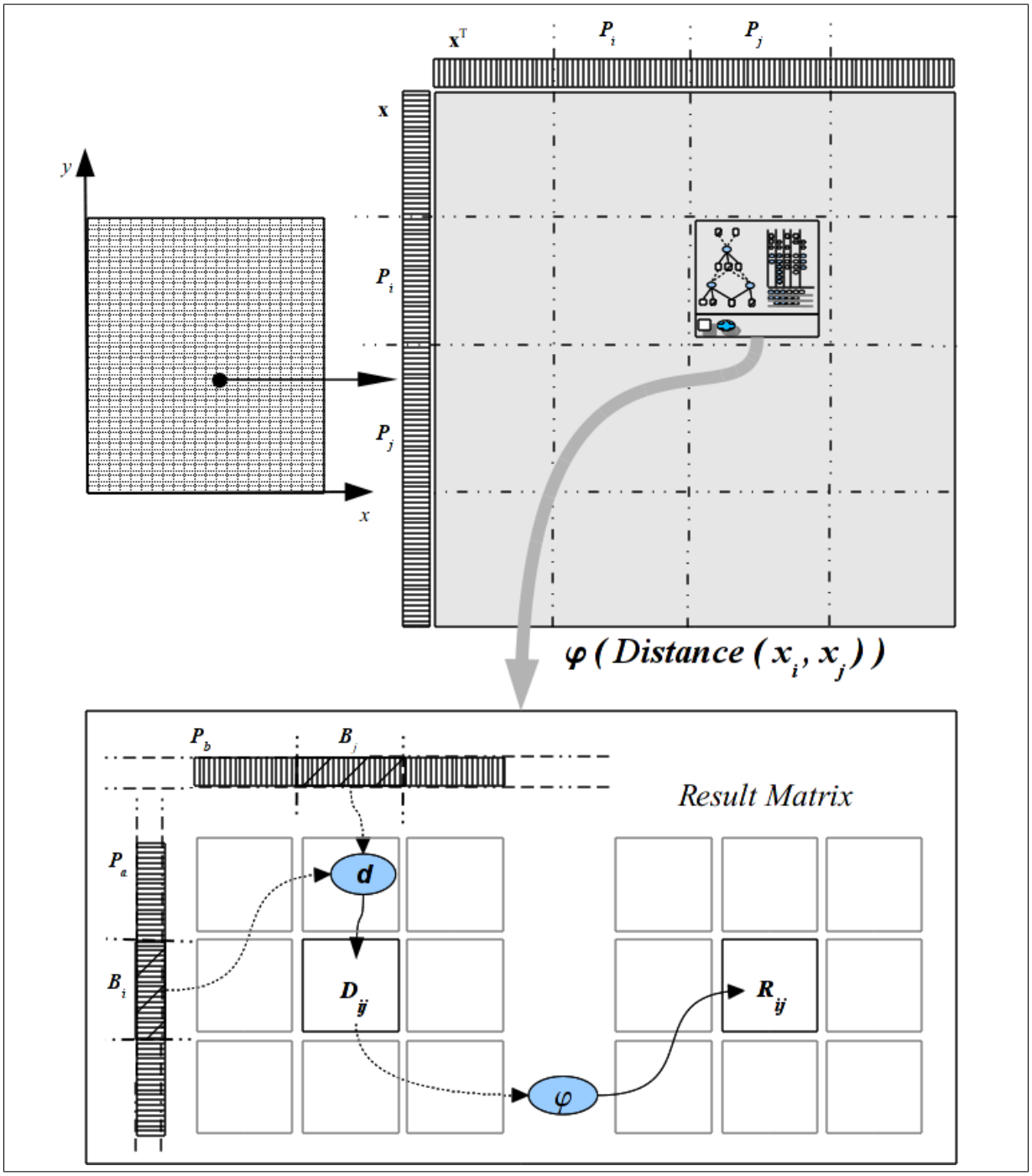


Figure 4.1: RBF Matrix Assembly

Algorithm 9 Assembling The RBF Matrix-Distributed Tasks Version

Require: $\forall i \quad 0 \leq i < N_n; \quad P_i \subset \mathbf{x}.$

Require: $\forall i, j \quad 0 \leq i, j < N_n \wedge i \neq j; \quad P_i \cap P_j = \emptyset.$

Require: $\bigcup_{i=0}^{N_n - 1} P_i = \mathbf{x}.$

for $i, j \in [0, ..., N_n)$ **do**

  $task \leftarrow$ **DuctTeip**.CreateTask ('Assemble');

  $task$.Host $\leftarrow node_i$;

  $d_t \leftarrow$ **DuctTeip**.CreateDataAccess (P_i);

  d_t.Host $\leftarrow node_x$;

  d_t.Access $\leftarrow$ **Read**;

  $task$.DataAccess$_0$ $\leftarrow d_t$;

  $d_t \leftarrow$ **DuctTeip**.CreateDataAccess (P_j);

  d_t.Host $\leftarrow node_y$;

  d_t.Access $\leftarrow$ **Read**;

  $task$.DataAccess$_1$ $\leftarrow d_t$;

  $d_t \leftarrow$ **DuctTeip**.CreateDataAccess (D_{ij});

  d_t.Host $\leftarrow task$.Host;

  d_t.Access $\leftarrow$ **Write**;

  $task$.DataAccess$_2$ $\leftarrow d_t$;

  DuctTeip.Tasks.Add ($task$);

end for

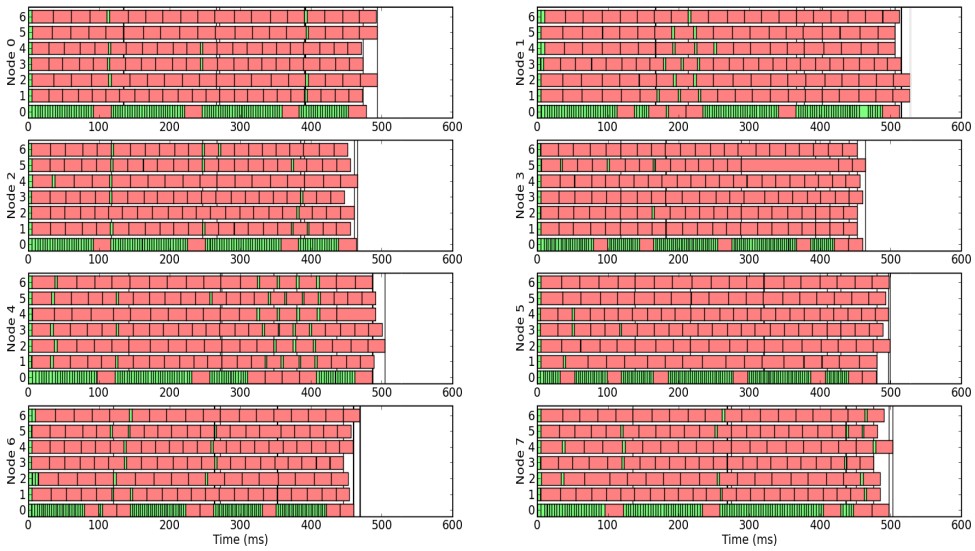
DuctTeip.Execute();

Algorithm 10 **Scheduler**.Run(*global_task*) for *RBF Matrix Assembling*

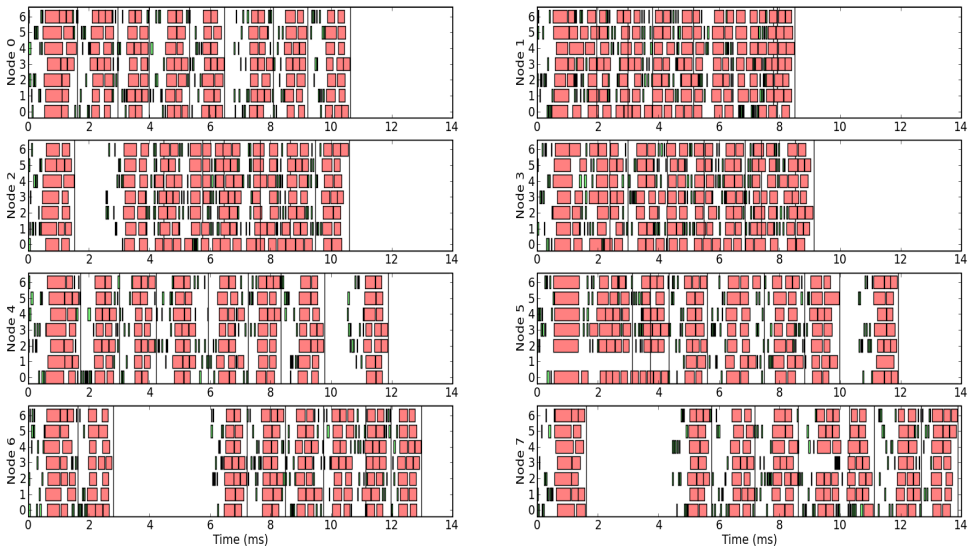
Require: $N_b > 0$ is the number of blocks for points vectors $P_i\,, i \in [0, N_n)$.

Require: $S_b > 0$ is the size of each block in $P_i\,, i \in [0, N_n)$.

1: $P_0 \leftarrow$ *global_task*.DataAccess$_0$;
2: $P_1 \leftarrow$ *global_task*.DataAccess$_1$;
3: **for** $i \in [0, N_b)$ **do**
4: **for** $j \in [0, N_b)$ **do**
5: $B_i \mapsto P_0[i.S_b : (i+1).S_b]$;
6: $B_j \mapsto P_1[j.S_b : (j+1).S_b]$;
7: *local_task* $\leftarrow$ **SuperGlue**.CreateTask($||.||$);
8:
9: $d_t \leftarrow$ **SuperGlue**.CreateDataAccess (B_i);
10: d_t.Access $\leftarrow$ **Read**;
11: *local_task*.DataAccess$_0$ $\leftarrow$ d_t;
12:
13: $d_t \leftarrow$ **SuperGlue**.CreateDataAccess (B_j);
14: d_t.Access $\leftarrow$ **Read**;
15: *local_task*.DataAccess$_1$ $\leftarrow$ d_t;
16:
17: $d_t \leftarrow$ **SuperGlue**.CreateDataAccess (D_{ij});
18: d_t.Access $\leftarrow$ **Write**;
19: *local_task*.DataAccess$_2$ $\leftarrow$ d_t;
20:
21: **SuperGlue**.Tasks.Add (*local_task*);
22:
23: *local_task* $\leftarrow$ **SuperGlue**.CreateTask($\varphi(\cdot)$);
24:
25: d_t.Access $\leftarrow$ **Read**;
26: *local_task*.DataAccess$_0$ $\leftarrow$ d_t;
27:
28: $d_t \leftarrow$ **SuperGlue**.CreateDataAccess (R_{ij});
29: d_t.Access $\leftarrow$ **Write**;
30: *local_task*.DataAccess$_1$ $\leftarrow$ d_t;
31:
32: **SuperGlue**.Tasks.Add (*local_task*);
33:
34: **end for**
35: **end for**



(a) Node Problem Size=1000000 elements



(b) Node Problem Size=10000 elements

Figure 4.2: Task Execution of 8 nodes of 7 Threads, Different Problem Size

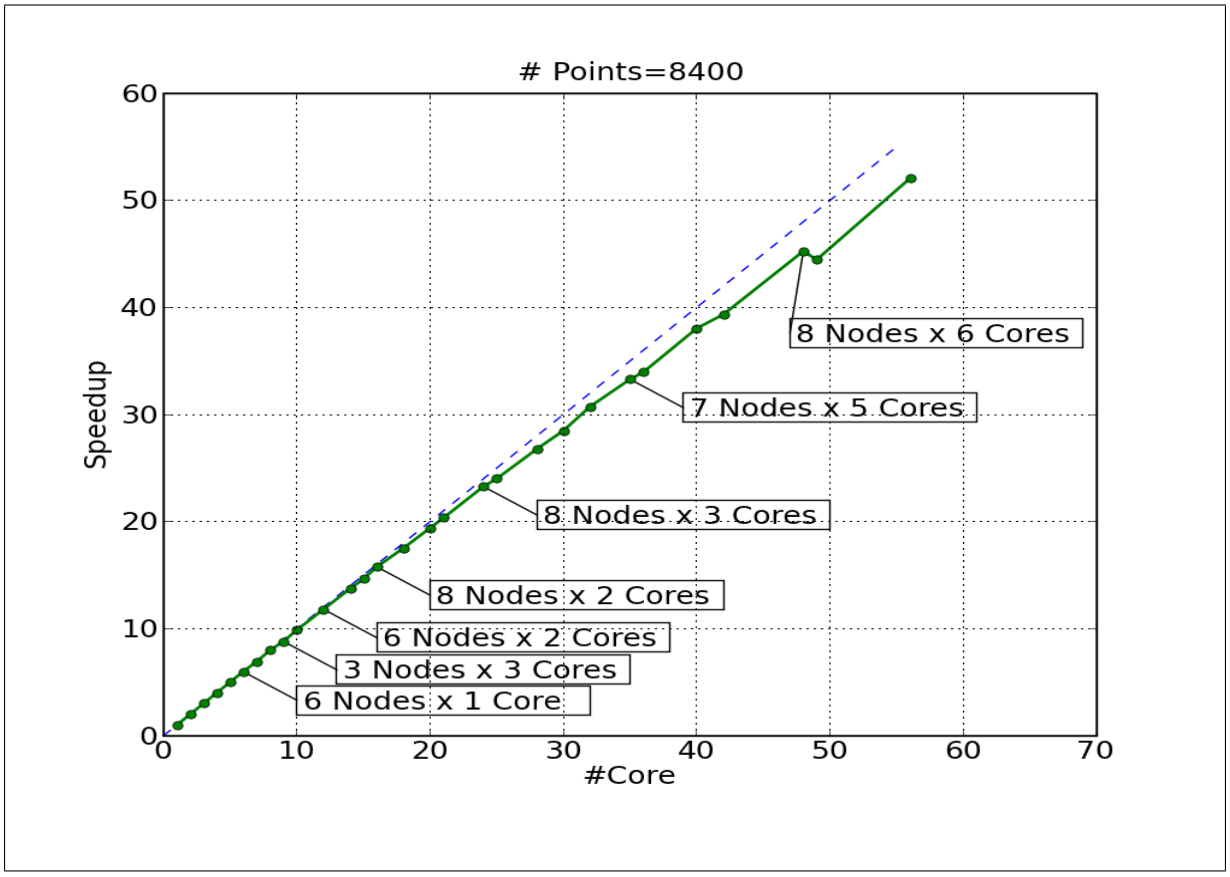
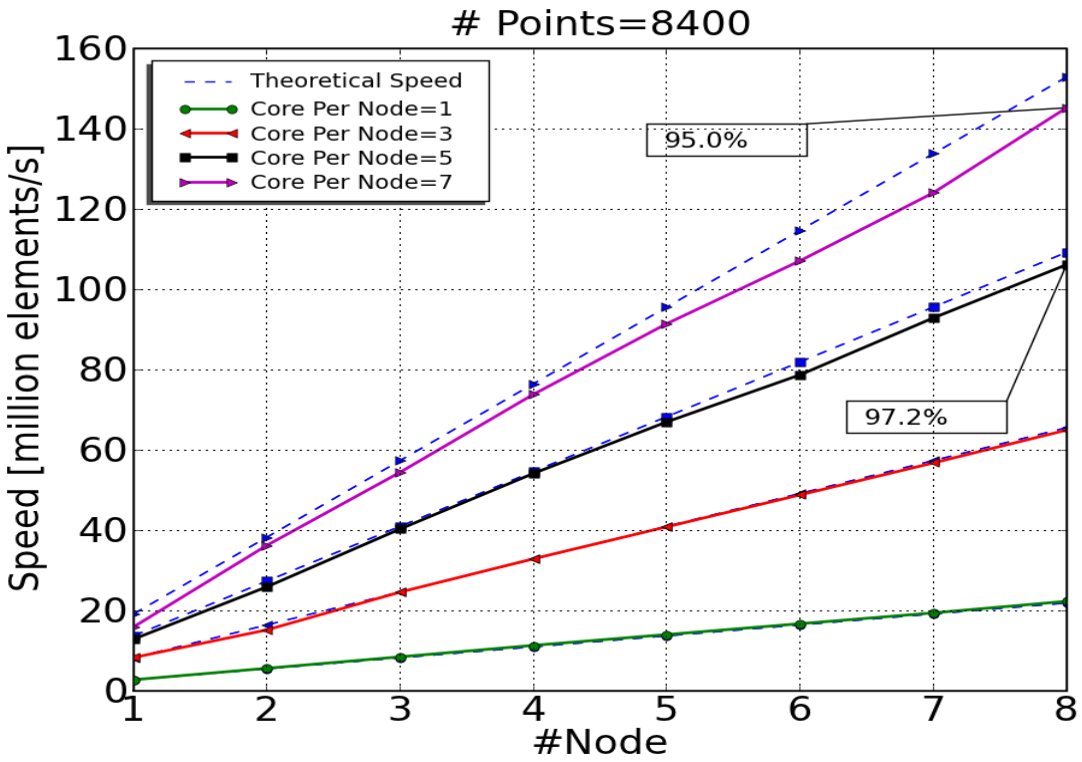
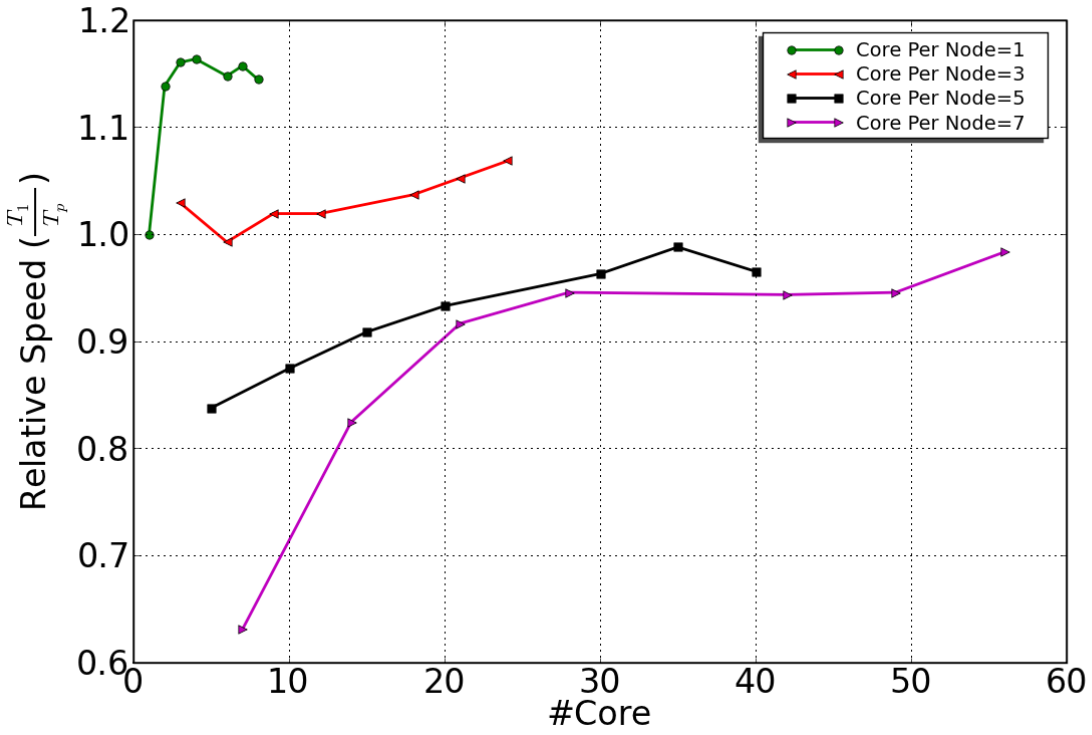


Figure 4.3: Speed Up of the Program,Matrix Size=8400 × 8400



(a) Strong Scaling



(b) Weak Scaling

Figure 4.4: System Performance for Different Sizes of the problem and Number of Cores

Chapter 5

Summary

5.1 Summary

This project started with some needs to parallel execution of programs for radial basis functions problems which may require great amount of computations due to, for example, either the size of the problems or investigating optimum values of some parameters for a single problem. The required parallelism has been provided by **SuperGlue Library** for shared memory architectures and has shown exciting and competitive results in several efficiency and performance experiments. The key point of this library is using Task Based model for providing or implementing concurrent execution of programs and this project aimed at extending it further toward for distributed memory architectures. In this model, a program is viewed as a set of tasks which read data and produce other set of data that other tasks may need. All the dependencies between tasks are translated by data dependencies and are tracked by data versions attached to both data items and also data accesses by tasks. All the tasks that their required input data at specific versions are ready can run in parallel. To translate a program to set of tasks and data needed by framework, it is sufficient to determine tasks and annotate their input/output data accesses accordingly. Then at run time, framework will execute tasks whose input data are ready and upgrade the version numbers of involved data. All ready tasks are queued at available threads for execution in parallel.

The same model of tasks and data versioning is used for distributed memory architectures in which processors talk about tasks and data which may reside in remote nodes using message passing. All the communication between processors (nodes) are implemented in non-blocking mode to be overlapped with computations. Tasks are distributed into nodes and nodes who need to any remote data, send request for the data to its owner and the owner will send it whenever the data gets ready locally. Whenever a task is ready to run, it will be sub-divided locally and delivered to **SuperGlue Library** for being executed in parallel on available threads which constructs a hybrid of message passing and thread parallelizations models. All the cooperation among nodes are performed in an asynchronous way which is implemented by using of state

transitions and event handling methods for controlling the program execution.

5.2 Conclusion

In this project, we have designed and implemented Task Based Parallelization for distributed memory architecture by extending an existing shared memory architecture framework. The main objective of this project was to improve the parallelism of a program by avoiding unnecessary explicit synchronizations and barriers for different spots of the program. This provides us good opportunities to remove the idle times of processors waiting for other ones to complete their work and reach the synch point and make the total make span of the program shorter. One of the key aspects of this project is using hybrid method of parallelizations for shared and distributed memory archtitectures. Our experiments show that this work reveals acceptable scalability and speedup results for solving large problems with more processors in differrent cores and nodes configurations.

5.3 Future Works

This project prepared a suitable foundation for examining other capabilities of the Task Based Parallelizations. In its current state, the framework contains a few simple matrix operations (multiplication and assembly) which are implemented for assessing the new hybrid method of parallelization. To extend its usability, developers need to provide more operations (e.g basic linear algebra ones) by implementing task-generating programs and corresponding kernels. Once these more general operations are implemented in the framework, they can be used by more applications than RBF methods. To reach to this state of generality, the framework needs to be equipped with more functionalities (e.g general data definition and partitioning) and hence can be extended in several research directions. In one direction, for example, it is possible that new computational kernels be implemented and the program be examined and compared with other available frameworks in Task Based Parallelization for solving the same computational problems. In other direction, it is also possible to enhance the features of the work by introducing such functionalities as load balancing and automated task generation to the program. Enabling the work for very large scale parallel computers in future clusters which will certainly consist of heterogeneous processors can also enrich the framework. To let the end user to utilize all the intrinsic features of this work for parallelization, it can be extended toward a user friendly framework which hides from its end user the technical details and hardware dependent considerations of writing programs for parallel execution.

Bibliography

[1] *ScaLAPACK Users' Guide*, May 1 1997.

[2] *A User's Guide to the BLACS v1.1*, May 5 1997.

[3] *Intel Cilk Plus Language Extension Specification*. `http://software.intel.com/sites/default/files/m/4/e/7/3/1/40297-Intel_Cilk_plus_lang_spec_2.htm`, 2011.

[4] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, *PLASMA Users' Guide*, Electrical Engineering and Computer Science,University of Tennessee; Electrical Engineering and Computer Science,University California at Berkeley;Mathematical & Statistical Sciences,University of Colorado Denver.

[5] E. Bollig, N. Flyer, and G. Erlebacher, *A multi-CPU/GPU implementation of RBF-generated finite differences for PDEs on a sphere*, AGU IN77, (2011).

[6] G. Bosilca., A. Bouteiller., A. Danalis., M. Faverge., A. Haidar., T. Herault., J. Kurzak., J. Langou, P. Lemarinier., H. Ltaief., P. Luszczek., A. YarKhan., and J. Dongarra., *Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA*.

[7] B. S. Center, *Nanos++ Overview — Programming Models @ BSC*. `http://pm.bsc.es/nanox`, 2012.

[8] ——, *SMP superscalar*. `http://www.bsc.es/computer-sciences/programming-models/smp-superscalar`, 2012.

[9] ——, *SMPSs Programming Model*. `http://www.bsc.es/computer-sciences/programming-models/smp-superscalar/programming-model`, 2012.

[10] N. I. for Research in Computer Science and Control(INRIA), *StarPU Handbook*. `http://runtime.bordeaux.inria.fr/StarPU/starpu.html#StarPU-MPI-support`, 2012.

[11] Fraunhofer ITWM Inst., *Global Programming Interface (GPI),User Manual*.

[12] ——, *GPI - Global Address Space Programming Interface Efficient-Scalable-Multicore.*

[13] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra, *Analysis of Dynamically Scheduled Tile Algorithms for Dense Linear Algebra on Multicore Architectures.*

[14] M. Ingber, C. Chen, and J. Tanski, *A mesh free approach using radial basis functions and parallel domain decomposition for solving three-dimensional diffusion equations*, International Journal For Numerical Methods In Engineering, 60 (2004), pp. 2183–2203.

[15] F. I. Inst., *The building blocks for HPC:GPI and MCTP,Efficient Scalable Multicore.*

[16] Intel, *Intel Threading Building Blocks, Tutorial.*

[17] M. Krishnan, B. Palmer, A. Vishnu, S. Krishnamoorthy, J. Daily, and D. Chavarria, *The Global Arrays User Manual*, Pacific Northwest National Laboratory, Novermber 12 2010. Technical Report Number PNNL-13130.

[18] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, *Scheduling dense linear algebra operations on multicore processors*, Concurrency Computat.: Pract. Exper., 22 (2010), pp. 15–44.

[19] E. Larsson, K. Åhlander, and A. Hall, *Multi-dimensional option pricing using radial basis functions and the generalized Fourier transform*, Journal of Computational and Applied Mathematics, 222 (2008), pp. 175–192.

[20] D. Lazzaro, *A parallel multivariate interpolation algorithm with radial basis functions*, International Journal of Computer Mathematics, 80 (2003), pp. 907–919.

[21] OpenMP ARB, *OpenMP Application Program Interface*, 2011.

[22] U. Pettersson, E. Larsson, G. Marcusson, and J. Persson, *Improved radial basis function methods for multi-dimensional option pricing*, Journal of Computational and Applied Mathematics, 222 (2008), pp. 82–93.

[23] T. Rendall and C. Allen, *Parallel efficient mesh motion using radial basis functions with application to multi-bladed rotors*, International Journal For Numerical Methods In Engineering, 81 (2010), pp. 89–105.

[24] F. Song, A. YarKhan, and J. Dongarra, *Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems.*

[25] L. Sunde, *Parallelizing a Software Framework for Radial Basis Function Methods*, 2011.

[26] Supercomputing Technologies Group,MIT Laboratory for Computer Science, *Cilk 5.4.6 Reference Manual.*

[27] M. Tillenius and E. Larsson, *An efficient task-based approach for solving the n-body problem on multicore architectures.*

[28] R. Yokota, L. A. Barba, and M. G. Knepley, *PetRBF-A parallel O(N) algorithm for radial basis function interpolation*, Elsevier, (2009).