PROGRAMMING SMP CLUSTERS: NODE-LEVEL OBJECT GROUPS AND THEIR USE IN A FRAMEWORK FOR NBODY APPLICATIONS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
İlker Cengiz
September, 1999

PROGRAMMING SMP CLUSTERS: NODE-LEVEL OBJECT GROUPS AND THEIR USE IN A FRAMEWORK FOR NBODY APPLICATIONS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

İlker Cengiz

September, 1999

B649443

QA
76.5
.C46
1998

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Attila Gürsoy (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Assoc. Prof. Özgür Ulusoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. İlyas Çiçekli

Approved for the Institute of Engineering and Science:

M. Baray

Prof. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

PROGRAMMING SMP CLUSTERS:
NODE-LEVEL OBJECT GROUPS and
THEIR USE IN A FRAMEWORK FOR NBODY APPLICATIONS

İlker Cengiz
M.S. in Computer Engineering and Information Science
Supervisor: Asst. Prof. Attila Gürsoy
September 1999

Symmetric Multiprocessor (SMP) cluster architectures emerge as a cheaper but powerful way of building parallel programming platforms. Providing mechanisms, layers of abstraction, or libraries gaining the power of SMP clusters is a challenging field of research. Viewing an SMP architecture as an array of processors would be insufficient, since such a model ignores essential possible gains over performance. We have stressed on reusable patterns or libraries for collective communication and computations that can be used commonly in parallel applications within a parallel programming environment utilized for SMP clusters. We introduce node-level replicated objects, since replicated objects provide a versatile abstraction that can be used to implement static load-balancing, local services such as memory management, distributed data structures, and inter-module interfaces. This work was motivated while we were developing parallel object-oriented hierarchical Nbody applications with Charm++. We discuss common paradigms that we came across in those applications and present a framework for their implementation on SMP clusters. If the bodies that an interaction needs are local then that interaction can be completed without any communication. Otherwise, the data of the remote bodies must be brought, and after the interaction calculation, the remote body data must be updated. Parallel object-oriented design of this framework hides communication details of bringing remote bodies from programmer and presents an interface to develop and experiment with nbody algorithms.

Keywords: SMP Clusters, Parallel Object-Oriented programming, Hierarchical NBody Methods.

ÖZET

BAKIŞIMLI ÇOKLU-İŞLEMCİ ÖBEKLERİNİ PROGRAMLAMAK:
DÜĞÜM SEVİYESİNDE DALLI NESNELER ve
SIRADÜZENSEL ÇOKLU-ETKİLEŞİM YÖNTEMLERİ İÇİN
TASARLANAN BİR ÇATI

İlker Cengiz
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yard. Doç. Dr. Attila Gürsoy
Eylül 1999

Bakışımlı Çoklu İşlemciye (SMP) sahip iş istasyonları üretmeye yönelik eğilim bu tür iş istasyonlarini hızlı ağlarla birbirine bağlayarak ucuz ama güçlü koşut programlama platformları oluşturma yönündeki araştırmaları arttırmaktadır. Bu tür platformları oluşturmanın yanı sıra, programcıların SMP öbeklerinin vaadettiği güçten yararlanmalarını sağlayacak farklı düzeylerde soyutlamalar, mekanizmalar ve yordam kütüphaneleri sunabilmek te başlıbaşına bir araştırma konusudur. Bir SMP mimarisini işlemciler dizisi olarak görmek yetersiz bir yaklaşım olacaktır, çünkü böyle bir model başarım açısından olası faydaları gözardı etmektedir. SMP öbekleri için yazılan koşut programlarda ortak olarak kullanılabilecek iletişim ve hesaplama örüntülerini içeren yeniden kullanılabilir yordam kütüphaneleri üzerinde çalıştık. Durağan yük dengelemede, bellek yönetiminde, dağıtık veri yapıları ve modüller arası arayüzler oluşturmada kullanılabilen dallı nesneleri SMP öbekleri için düğüm seviyesinde yeniden tanımladık. Bu çalışmada Charm++ koşut nesneye-yönelik programlama dili ile koşut sıradüzensel çoklu-etkileşim uygulamaları geliştirirken karşılaştığımız ortak kavramları tartıştık ve bu tür uygulamaları SMP öbeklerinden faydalanarak geliştirmek, deneysel amaçlarla kullanabilmek için bir çatı tanımladık. Bu tür yöntemlerde ortak olarak etkileşime konu olan iki parçacık eğer ayni adres uzayında ise düğümler arası herhangi bir iletişim gerekmez. Ancak aksine iki parçacık farklı adres uzaylarında ise etkileşimin hesaplanabilmesi için parçacıkların etkileşimle ilgili verilerinin birbirlerinin adres uzaylarına getirilmesi gerekir ki bu da SMP düğümleri arası ağ üzerinden yapılan iletişim demektir. Sundugumuz çatı ve çoklu-etkileşim uygulama arayüzü koşut nesneye-yönelik tasarımı ile programcının iletişim ile ilgili detaylardan soyutlanarak deneysel amaçlı hızlı uygulama geliştirmesine yardımcı olacaktır.

Anahtar sözcükler: SMP Öbekleri, Koşut Nesneye Yönelik Programlama, Çoklu-Etkileşim Yöntemleri.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest thanks and gratitude to my advisor Asst. Prof. Attila Gürsoy for his patient supervision of this thesis.

I am grateful to Assoc. Prof. Özgür Ulusoy and Asst. Prof. İlyas Çiçekli for reading the thesis and for their instructive comments. I would like to acknowledge the financial support of TÜBİTAK under the grant EEEAG-247.

I would also like to thank my parents and my sister Arzu for their encouragement; my friends Emek, Murat, Ümit, and Zeynep for their moral support; our department's administrative stuff especially Nihan for their help; Mr. Leslie Greengard for supplying his code; all other friends who contributed this study; and finally PC game producers for their games that helped me to have fun during the long lasting nights of thesis work.

To my lovely, the-one-and-the-only parents
Fahrünnisa and Süleyman Cengiz,

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success than to take the lead in the introduction of a new order of things. - Niccolo Machiavelli"

Symmetric Multiprocessor (SMP) platforms are going towards being a general interest in research. As workstations having multiprocessor architectures with shared-memory appear on market, it becomes attractive to built larger multiprocessor systems by connecting such workstations. Taking SMP nodes as basis and building clusters of them leads a new way of thinking in parallel computing research. Modeling cluster of n k-way SMP nodes as a flat network of nk processors would not be sufficient to extract possible gains of that architecture. Shared-memory structure of an SMP node and the availability of dynamic load balancing for processors within a node are points worth to take into account in designing software systems for SMP clusters. Both shared-memory and distributed-memory paradigms apply in such systems. Processors within the same node (SMP node) share memory, while the nodes within the same system are subject of distributed memory programming.

The simplest programming model for SMP clusters is treating an SMP cluster as a flat network of processors and using message-passing or distributed memory paradigms. However this model causes the interactions between computations within a single node to go through the message passing layer, and the program will experience message passing overhead. This overhead can be bypassed using the fact that the SMP node allows sharing of memory at hardware level and computations within node can interact using shared memory model to deliver better performance. In this case an SMP node will use message passing layer only for interactions with other nodes. Such hybrid programming approaches have been proposed to program SMP clusters [2] [25], which have modeled SMP cluster as a network of nodes, in which an application is developed with the distributed memory approach where nodes exchange messages, but within a node the application employs a multi-threaded model to take advantage of multiple processors and shared address space.

Parallel object oriented languages encapsulates message-passing and multithreading in the object based model. Charm++ [16] [5] system, developed at UIUC, is such a message-driven object-based parallel programming environment. Charm++ as a concurrent object-oriented language, built on top of Converse [6] runtime, is promising for irregular parallel applications, where modularity and encapsulation provide help for programmers to design and implement complex data and parallelism structures [14] [4]. Moreover its message-driven nature allows overlapping communication with computation.

Converse runtime, so does Charm++, provides each processor with a private memory segment, even memory is physically shared among processors within an SMP node. Due to this fact advantages of having shared-memory can not be extracted, which is believed to be an essential fault for SMP programming. Moreover since each processor owns its address space, we do not have the opportunity to have shared parallel objects, which is desired to built a work-pool of tasks that can be executed by any of the idle processors in a node. If we can have work-pool model employed in SMP programming, dynamic load-balancing will happen to be achieved. Messaging in Converse environment is based on pe-to-pe communication, all message send and broadcast routines address single processors, therefore there can not be inter-node communication in a cluster of nodes in means of a software layer.

In order to have Charm++ to support these new architectures, efficient and elegant mechanisms should be added to its runtime. Our emphasis is on reusable patterns or libraries for collective communication and computations that can be used commonly in parallel applications. And we introduce node-level object groups to perform such operations efficiently on SMP clusters. Replicated objects are known to provide a versatile abstraction that can be used to implement static load-balancing, local services such as memory management, distributed data structures, and inter-module interfaces [18].

The thesis is organized as follows. Chapter 2 introduces the Charm++ parallel programming environment, and presents a brief information on programming using Charm++. Underlying mechanisms of Charm++, as scheduler and message handling are explained in this chapter.

Chapter 3 presents our understanding of effective SMP cluster programming. In this chapter with an overview of previous work, we explain and discuss our design and decisions about the mechanisms and programming constructs to implement, followed by sample applications, and performance considerations.

In Chapter 4 after an overview of general issues about NBody problem, related data structures, and algorithms, we discuss and present a framework for NBody algorithms to have them utilized to run on SMP clusters and hide away the communication details.

The thesis finishes by concluding the studies in the last chapter. Including the critique of our design and implementation. The goals that are met are stated, and those left as future work are discussed.

Chapter 2

Programming using Charm++

Parallel object oriented languages encapsulates message-passing and multithreading in the object based model. Charm++ [16] [5] system, developed at UIUC, is such a message-driven object-based parallel programming environment. The Charm++ environment is built on top of an interoperable parallel runtime system called Converse [15] [6]. Charm++ as a concurrent object-oriented language is promising for irregular parallel applications, where modularity and encapsulation provide help for programmers to design and implement complex data and parallelism structures [14] [4]. Moreover its message-driven nature allows overlapping communication with computation. Replicated objects are known to provide a versatile abstraction that can be used to implement static load-balancing, local services such as memory management, distributed data structures, and inter-module interfaces [18].

In rest of the chapter, we provide vital information about Charm++ programming environment, Converse runtime system, and simple programming examples.

2.1 Converse

Converse is designed to form a framework for other parallel programming paradigms to be employed in a system. Its runtime includes components for communication, scheduling, and load-balancing. In each processing node a scheduler is maintained, which is a thread executing an infinite loop. Programmer should explicitly associate each message a handler function. When a message is received, it is stored in the incoming messages queue of the receiver processor. Converse scheduler then dispatches the message by invoking its handler function, whose knowledge is extracted from the message itself. The designed nature of Converse causes it to be not suitable for programming, instead it is a lower layer to serve for an exact parallel programming language/paradigm. Further details on Converse programming framework can be found at `http://charm.cs.uiuc.edu`.

2.2 Charm++

Charm++, is a portable object-oriented parallel programming language. Its syntax is similar to that of C++, with extensions for concurrent objects. Multiple inheritance, and overloading features of C++ are extended for concurrent objects, while operations and manipulations on concurrent objects are restricted to satisfy parallel execution needs. There are five categories of objects in Charm++:

- Sequential objects (same as C++ objects)
- Messages (communication objects)
- Chares (concurrent objects)
- Branched Chares (grouped concurrent objects)
- Shared objects (specific information sharing abstractions)

Messages are the communication objects of Charm++, which have specific definition syntax since they are an extension for C++ (Figure 2.1). On a shared memory system, a message can store pointers as data members. However on message-passing systems, a pointer is not valid across distributed address spaces. So the whole memory field pointed by the pointer must be packed in a continuous space to eliminate explicit pointers. This brings the packing and unpacking of messages containing pointers, where pack and unpack functions are associated with the message type. It is user's responsibility to provide these two methods for a particular message type. The invocation of pack and unpack methods is directed by the runtime system in case they are needed. Concurrent objects have methods that lets them to receive messages. Such methods are called entry points that define the code to execute when a message is received. Entry point invocation is performed as passing a message pointer to that particular method.

```
message messagename {
        List of data members
}

chare class classname [: superclass name(s)]{
private:

entry:          entrypointname(messagename* )
        { c++ code };
}
```

Figure 2.1: Charm++ message and chare class definition syntax.

The basic unit of parallelism in Charm++ is the *chare*, which infact is similar to a process, or a task. At runtime active chares may send messages to each other, where the runtime is free to schedule them in any way. Method of a chare that can be invoked asynchronously with sending a message it.

A Branched Office Chare (BOC) is an object with a branch on every processor; all of the branches answer to the same name. Branched chares can have public data and function members as well as private and members and entry points. One can call public functions of the local branch of a BOC, send a message to a particular branch of the BOC, or broadcast the message to all of its branches. BOCs provide a versatile abstraction that can be used to implement static load-balancing, local services such as memory management, distributed data structures, and inter-module interfaces.

```
branched chare class classname [: superclass name(s)] {
private:

public:

entry:          entrypointname(messagename* )
        { c++ code };
}
```

Figure 2.2: Charm++ branched chare class (BOC) definition syntax.

2.2.1 Message Handling

The version of Converse we are working on is utilized for processing nodes having more than one processor. If the source and destination processors of a message are lying in the same node, it is inserted to the incoming-message queue of destination processor. Since memory is shared between such processors, the operation is a single memory-write. Otherwise, which means if source and destination processors are from two different processing nodes, the message is sent using UDP datagrams. Low-level datagrams are transmitted node-to-node (as opposed to pe-to-pe), but still an SMP node is handled as a network of processors, corresponding to network of single-processor workstations. Therefore concept of node-to-node messaging is not supported.

Sender scheduler wraps the message with additional system handler function and user handler function information prior to sending, where system level handlers are routines each specialized for a type of message in kernel. They process the message, and adjust system variables, for example there is handler for chare creation request messages, and one for ordinary user messages. On receive messages are inserted in the local queue of the destination processor of the node. As scheduler detects the existence of a message in its queue, triggers the handler function of the message with the message as a parameter to the function (Figure 2.3).

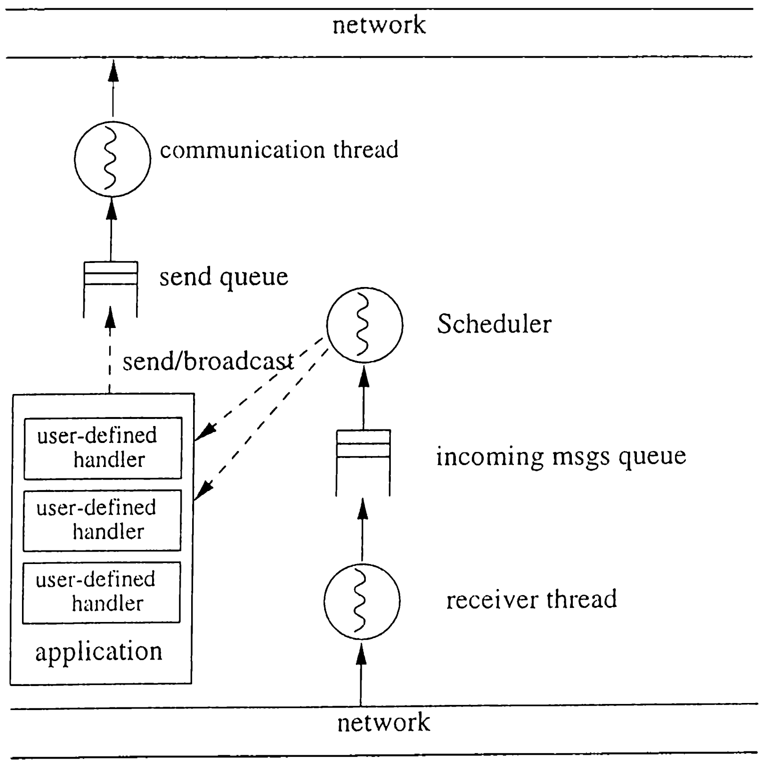

Figure 2.3: Converse level message handling.

```
class messagename : public comm_object {
        List of data members
}

class classname : public chare [,superclass name(s)] {
private:

public:
        void entrypointname(messagename* )
        { c++ code };
}
```

Figure 2.4: NonTranslator Charm++ message and chare class definition syntax.

2.3 Programming using Non-Translator version of Charm++

A Charm++ program contains modules, each defined in a separate file. A module may contain five type of objects mentioned above. The user's code is written is C++, and interfaces with the Charm++ system. A translator that is managing Charm++ constructs is used to generate ordinary C++ code that needs to be compiled with user's code.

There is an alternating way for programming using Charm++, which excludes the translator, and uses Charm++ as a library linked to C++ programs. When passed from Charm++ to non-translator version, syntax for class definitions change a bit,(see Figures 2.4 2.5). Superclass *comm_object* is base for user message classes, as *chare* for chares, and *groupmember* for BOCs. Using NT-Charm++ requires creating interface file for each module. The interface file is processed by a tool, that generates two header files per module. These two files must be included in user's C++ source files.

Programming with non-translator Charm++ is demonstrated in Figure 2.6. Module main contains a special chare named *main*, which should have a method with a reserved name *main*. Main chare has one copy over the whole system, and is executed on a system selected processor. Since the execution of the

```
class classname : public groupmember [, superclass name(s)] {
private:

public:
        ...
        void entrypointname(messagename* )
        { c++ code };
}
```

Figure 2.5: NonTranslator Charm++ branched chare class (BOC) definition syntax.

program starts from this entry, typically initializations, and object creations are performed within its block. *HelloBOC* is a branched chare class. In *main*, an instance (infact as many instances as the number of processors) is instantiated with the call `new_group`. This call gets the class name and a message pointer as parameters. The message is copied on each processor, which means the constructor entry of HelloBOC is invoked with same values in. Since InitMsg class contains an integer array, appropriate pack and unpack routines should be provided. Then main chare broadcasts a message to all chares of that instance to have them say "Hello". The order of which processor says hello is not predefined by any means. As the message arrives, runtime system picks and schedules the request on each processor independently, as dictated with the message-driven nature of Charm++.

The quiescence mechanism is useful where the user can not foresee when the program is going to be quiescent. To set the method to be invoked on quiescence, `CStartQuiescence` runtime call is used.

```
MODULE JunkMsg
Interface File JunkMsg.ci
message JunkMsg;
Source File JunkMsg.C
#include "ckdefs.h"
#include "chare.h"
#include "c++interface.h"
class JunkMsg : public comm_object { public:
    int junk;
};

MODULE HelloBOC
Interface File Hello.ci
packedmessage InitMsg;
extern message JunkMsg;
groupmember HelloBOC {
    entry HelloBOC(InitMsg *);
    entry sayHello(JunkMsg *);
}
SourceFile HelloBOC.C
#include "ckdefs.h"
#include "chare.h"
#include "c++interface.h"
#include "JunkMsg.h"
#include "HelloBOC.top.h"
#include "HelloBOC.h"

class InitMsg : public comm_object { public:
    int numParts;
    int* parts;
    void* pack(int *length)
        { }
    void unpack(void* in)
        { }
};

class HelloBOC : public groupmember {
private:
    ...
public:
    HelloBOC(InitMsg* msg)
        { initializations as an ordinary constructor }
    sayHello(JunkMsg* msg) {
        CPrintf("Hello Universe from pe %d",CMyPe()); }
};
```

Figure 2.6: Hello Universe program using NonTranslator Charm++.

```
MODULE main
Interface File main.ci
Source File main.C
#include "chkdefs.h"
#include "chare.h"
#include "c++interface.h"
#include "JunkMsg.h"
#include "main.top.h"
class main : public chare {
private:
    ...
public:
    ...
    void main(int agrc, char** argv) {
        InitMsg* initmsg = new (MsgIndex(InitMsg))InitMsg;
        initmsg-¿numParts = 5;
        initmsg-¿parts = new int[5];
        GroupIdType helloID = new_group(HelloBOC,initmsg);
        JunkMsg* hellomsg = new (MsgIndex(JunkMsg))JunkMsg;
        CBroadcastMsgBranch(HelloBOC,sayHello, hellomsg,helloID);
        CStartQuiescence(GetEntryPtr(main,quiescence),mainhandle);
    };
    void quiescence(JunkMsg* msg) {
        CPrintf(" Quiescence Reached ");
        CharmExit();
        delete msg;
    };
};
#include "main.bot.h"
```

output when executed on a 2 processor system:

```
Hello Universe from pe 0
Hello Universe from pe 1
Quiescence Reached
```

*can not make an assumption in order that processors say hello!

Figure 2.7: Hello Universe program using NonTranslator Charm++ (cont'd).

Chapter 3

Effective Programming of SMP Clusters

"Give me where to stand, and I shall move the world - Archimedes"

Trend towards producing workstations which have multiprocessors (SMPs), increases research endeavors to built cheaper but powerful parallel programming platforms through connecting such workstations. Besides building such platforms, providing mechanisms, layers of abstraction, or libraries to enable programmers to gain the power of SMP clusters is another challenging field of research. Viewing an SMP architecture as an array of processors would be insufficient, since such a model ignores essential possible gains over performance. In this chapter we have stressed on reusable patterns or libraries for collective communication and computations that can be used commonly in parallel applications within a parallel programming environment utilized for SMP clusters. We introduce node-level objects groups, since such objects provide a versatile abstraction that can be used to implement static load-balancing, local services such as memory management, distributed data structures, and inter-module interfaces [18].

3.1 Motivation

As workstations having multiprocessor architectures with shared-memory appear on market, it becomes attractive to build larger machines by connecting such workstations (SMP) by fast networks. Taking SMP nodes as basis and building clusters of them leads a new way of thinking. Modeling cluster of n k-way SMPs as a flat network of nk processors would not be sufficient to extract possible gains of that architecture. The main advantage of an SMP cluster is sharing the memory within a node. If the SMP cluster is viewed as a collection of single processor systems then the interactions between computations within a single node will go through the message passing layer (which supports communication between processors) and the parallel program will experience all the message passing overhead.

This overhead within a node is unnecessary because the SMP node allows sharing of memory at the hardware level and computations can interact using shared memory model (in which a better performance is expected). This overhead will be significant particularly for irregular an dynamic computations where shared memory programming much more easier to implement such cases.

A programming system where a cluster of n k-way SMPs are modeled as a collection of n nodes with appropriate support for expressing parallelism within a node will result in better performance. In this case, computations within an SMP node now coordinate their actions through the shared memory, and only for interactions with other nodes will use the message passing layer. A number of such hybrid models combining explicit message passing and multithreading are present in the literature. Bader et.al [2] presented a kernel of communication primitives with layers of abstractions to program clusters of SMP nodes. Their kernel combines shared-memory and distributed memory programming using threads and MPI-like message passing paradigm. The need for a hybrid model is also addressed by Tanaka et.al [25] in a previous work. Their model utilizes multi-threaded programming (Solaris threads) for intra-node part of SMP programming. For inter-node part of programming they have offered remote memory operations in conjunction with message passing, to overcome mutual exclusion on buffers and message copying overheads of message passing.

In this work, we want to support SMP clusters within an object based language environment, namely Charm++. The current implementation of Charm++ (version 4.0), parallel objects are assigned to processors and each processor (Unix process) has its own distinct address space. Such a model prohibits us to exploit the features of SMP clusters: that is parallel objects within the same SMP node can't share memory. And also, an idle processor cannot execute a parallel object assigned to a different processor within the same node. What we need is node level parallel objects, in addition to processor level ones and some abstractions with efficient implementations to allow us:

- ability to share the memory across objects on the same SMP node,
- ability to run a method of a parallel object by any processor,
- a framework which will support remote-object accesses easily and collective operations efficiently.

In this chapter, we will describe mechanisms (within Charm++ programming framework) to support SMP clusters. First, we will discuss how we can allow objects to share memory (using threads) and how an idle processor within a node can invoke parallel objects within the same node (Node Level Message Queue). More importantly, we will introduce *Node Level Object Groups* for effective implementation of collective operations across nodes.

3.2 Modifying Converse Runtime

Charm++ is built on top of Converse runtime, which serves as a lower layer for parallel programming paradigms and languages. In order to have Charm++ supporting SMP nodes in means of ability to share memory within a node, and ability to let any idle processor to invoke parallel objects within that node, Converse runtime should be modified. This section presents those modifications to Converse layer of our programming environment.

3.2.1 Shared-Address Space

In network of processors model employed by Converse runtime, memory can not be shared within a node. Assuming that the underlying operating system provides threads to gain access to those multiple processor of an SMP node, an interface layer is served in Converse runtime. This layer contains routines to start the threads, routines to access thread specific state, and routines to control mutual exclusion between them. If one process is created in each node and each scheduler is run by a thread, provided by the operating system, within the same address space, then parallel objects that are mapped to different processors can access each other within the SMP node directly. In a k-processor node, there will be k-threads each running Converse scheduler, and communication thread to handle incoming message from the network.

3.2.2 Node Level Message Queue

Even this configuration does not let a parallel object in an address space to be executed by any of the schedulers of the node, a desired case when we want to have node-level-shared parallel objects. As we have stated in Section 2.2.1 a message sent to a parallel object will be inserted to the queue of the scheduler owning that object. And only the scheduler which owns that queue can process this message even though some of the other schedulers might be idle in the same SMP node. This fact lead to create and use another message queue that will be shared between all processors of a node. We call this new queue as node-level message queue(NLQ). Defining node-level messages, messages that can be directed to a node instead of a processor, is the next step through our aim. Upon recieval a node-level message will be inserted into the NLQ by the communication thread. When we have the Converse scheduler modified to check both its own message queue and NLQ, we provide a node-level message to be picked up by any scheduler. There is a decision to be taken here as; whether a scheduler checks its own queue then NLQ or vice versa. Currently the shared queue is checked at first hand, cause we believe the node-level messages have higher priority when compared to standard messages. In accordance to message handling style of Converse runtime, we have added a new system level handler function utilized for handling node level messages. As a message is detected in NLQ, the receiving scheduler will invoke this new system handler that will trigger message associated user handler function upon processing and adjusting system variables.

In order to enable programmer to use node-level messages, we added two new functions to Converse kernel: `CmiSendNodeMsg` and `CmiBroadcastNodeMsg`. The first call sends a node-level message to a user-specified destination node. On recieval the message is inserted in the NLQ, so that it is available for the Schedulers. The latter call is used to broadcast a unique message to all nodes within the system. Again the message is inserted in NLQs of receiver nodes. There are two more optionally added functions in Converse kernel: `CmiBroadcastInNodeMsg`, to broadcast a unique message in the caller processor's owner node except the caller, and `CmiBroadcastAllInNodeMsg`, to broadcast a unique message in the caller processor's node including the caller.

As a summary of all, our modified Converse is able to support node-level messaging through shared message queue, and such messages can be picked up by any idle scheduler of the receiver node.

3.3 Moving from Converse to Charm++

Now our Converse is able to run on SMP nodes, and we have node-level messages. However we have work to do, so that the underlying mechanisms of Converse becomes usable by Charm++. First to look at is the new concept of node-level messages. According to the path a message traverses, a node-level message will be passed to Charm++ runtime by Converse layer. We introduce a new system-level handler function to process only node-level messages, which will be triggered instead of the standard one. This handler function works in same manner; it sets the handler for the message, and inserts it to queue. The queue mentioned here should not be the scheduler's own queue as opposed by unmodified version of Charm++, same arguments we have mentioned for Converse also applies here. In such a design a message will be tied to the processor, whose Converse scheduler retrieves message in Converse level. But we want the message to be available for any of the inner-node processors in Charm++ level. So a node-level message queue for Charm++ is needed. Instead of creating a new message queue and bothering with modifications on schedulers and cause cpu-time to be wasted by schedulers checking one more queue, the NLQ created for Converse level is used again. Since schedulers already check these queues, we don't need anything more, recall from Section 2.2 that Charm++ and Converse use the same scheduler code. When the Charm++ scheduler detects this message, it will extract the user-defined entry method pointer from the message and trigger it with the message as a parameter. Finally for Charm++, the system calls corresponding to those of Converse are added to Charm kernel. CSendNode(CmiSendNode), CBroadcastNode(CmiBroadcastNode), CBroadcastInNode (CmiBroadcastInNode), and finally CBroadcastAllInNode (CmiBroadcastAllInNode), are representatives of Converse functions, when moved to Charm++.

As inner-node memory sharing is available we can now look for the ability to share parallel objects in nodes.

3.4 Node-Level Object Groups - The NLBOC Pattern

A BOC is a group of chares that has a branch/representative on each processor, with each branch having its own data members. Branched chares can be used to implement data-parallel operations, which are common in irregular parallel applications. Messages can be broadcasted to all branches of a branched chare as well as sent to a particular branch. There can be many instances corresponding to branched chare type: each instance has a different handle and its set of branches on all processors.

The effective use of branched chares in data-parallel operations, and usability in irregular parallel algorithms brought a new concept; Node-Level Branched Chares - NLBOC [14]. A NLBOC is a group of chares that has a branch on each node. Having an instance of an NLBOC means there exists

```
class obj : public gen_shared_object{
private:

public:
        obj();
        method1(param1);
        method2(param2);
}
```

Figure 3.1: A sample sequential C++ object to share.

a representative on each node, such that these representatives can be reached just by using the NLBOC handle. Communication can take place between the branches of an NLBOC. Moreover NLBOCs may used to encapsulate node level shared objects, a node level shared object is an object that is shared among all processors in a node. Suppose we have a sequential object having two methods, as shown in Figure 3.1. Shared Object must be derived from the base class `gen_shared_object` to enable type casting in implementation. There is no other restriction on the design and implementation of this class, unless it is a legal C++ class. We want it to be shared in a SMP node, such that each processor in that node may access it, moreover may execute any of its methods.

To satisfy such a request, shared object may be encapsulated within a NLBOC object, which will serve as an interface for initialization and method execution of it. During our implementation of NLBOC's, we have covered this concept and provided an interface.

3.4.1 Implementation of NLBOCs

Instead of modifying Charm++ language, we preferred to implement NLBOC, using standard BOC class and inheritance. We have developed a base class, `NodeBOC` as seen in Figure 3.2, which can be used by Charm++ programs. NodeBOC base class is derived from Charm++ BOC class groupmember. Thus

```
class NodeBOC : public groupmember {
private:
        void *shared_obj;

protected:
        void initSharedObject(object *,callbackfn*);
        void exec(...);
public:
        NodeBOC();
}
```

Figure 3.2: NodeBOC class interface.

when a NodeBOC object is instantiated, Charm++ runtime creates a branch on each processor. But this time all branches in a node act as a single instance of NodeBOC.

NodeBOC class maintains reference named `shared_object`, so that it is possible to encapsulate a shared object within a node. Classes derived from NodeBOC must call `initsharedObject` method to have this reference set. In a node the branch mapped to processor ranking zero is responsible for initialization of locally shared references. The shared object is created by that branch, and reference for this object is broadcasted in node, so that each branch has the reference. Programmer does not need to bother about this process, as he doesn't need to know whether the reference is set in each branch of a node. The method `exec` provides the programmer with the facility to call any method of the shared object. It is possible to make direct calls to shared object without using exec method, but since the global-in-node reference can not be guaranteed to be set, this type of action is not advised. Passing the method pointer and parameters for shared object's desired method to exec method means having that particular method executed. Currently only one parameter may be passed to the shared object, and since this parameter is of `void` pointer type, appropriate cast must be performed in the methods of the shared object. Using structs may be an answer in increasing the number of parameters to pass in shared object's methods. In fact that's how the parameter passing is performed

Macro	Explanation
Shared_Object_Init(...)	Start initialization of shared object. Processor ranking 0 in each node crates an object of specified type, then broadcasts the pointer in its node. Receiving processors execute the callback function to set the reference.
Shared_Object_Init2(...)	Works same as the one above. But this macro lets to pass parameters for the constructor of shared object.
StartExec(...)	Deposits the request to execute the specified method of shared object. There is no synchronization check in this execution.
StartExec2(...)	Same as the one above, but does not let concurrent execution of the specified method. If ticket is not available, the request is enqueued in NodeBOC queue.

Figure 3.3: Macros associated with shared object operations.

in Charm++ messages: enclosing many variables.

Our sample class `A` derived from NodeBOC base, has methods `method1` and `method2` which can considered to be interfaces for methods of shared object class, illustrated in Figure 3.4. To hide the details in initialization and method invocation of shared object, we have provided two macros, see Figure 3.3.

Assume a node-level message is directed to class `A` for method1, which in fact stands for a request for execution of method1 of shared object. Here raises a question of which of the processors in a node should execute the methods of this shared-object in a node. Any of the processors may handle the messages directed to this shared object. That means when a message is detected in NLQ by one of the schedulers, it will be picked up and associated method will be triggered. If a second message is detected during this period, and handled by one of the other schedulers, two processors will then be executing the same object's method(s). This case is new for Charm++, since it does not allow intra-object parallelism. However with SMP support, more than one method

```
class A : public NodeBOC {
private:
        obj *object;
public:
        void A(){
                ...
                Shared_Object_Init(obj,&A::fn);
        }
        void method1(MSG *msg){
                extract parameters from msg and produce param
                StartExec(obj,&object::method1,(void*)param);
        }
        void method2(MSG *msg){
                extract parameters from msg and produce param
                StartExec(obj,&object::method2,(void*)param);
        }
        void fn(void* ref){
                object = (obj*)ref;
        }
}
```

Figure 3.4: Simplified sample class A derived from NodeBOC.

of an object can potentially be invoked at the same time. With NLBOC's intra-object parallelism, programmer may need to deal with synchronized access to shared data with locks etc. Many applications might need a NLBOC where only one method can be executed at a given time. NLBOCs should provide synchronization when desired by the programmer. We have implemented a ticket-based algorithm, to solve the problem of synchronization (illustrated in Figure 3.5.

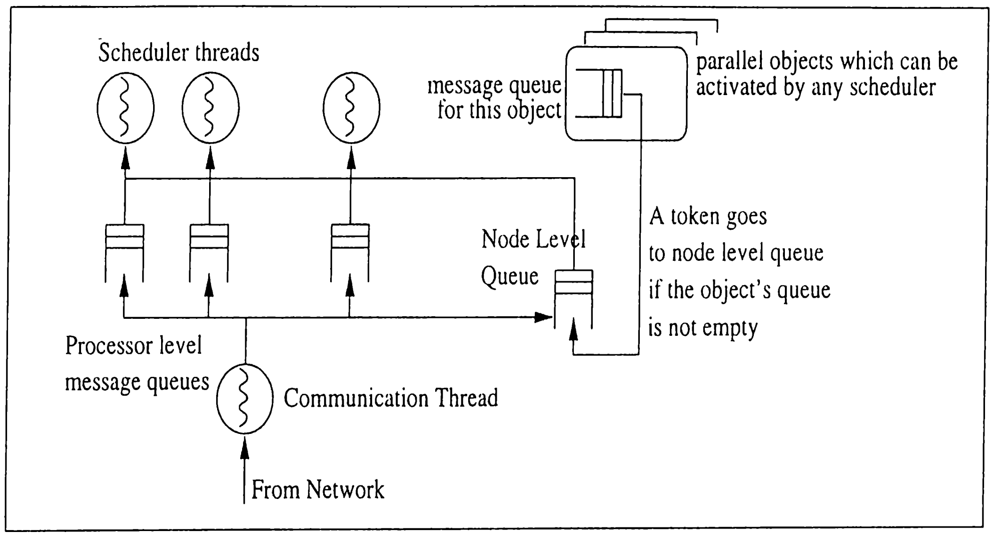
Scheduler threads
message queue for this object
parallel objects which can be activated by any scheduler
Node Level Queue
A token goes to node level queue if the object's queue is not empty
Processor level message queues
Communication Thread
From Network

Figure 3.5: Efficient implementation of NLBOC.

3.4.2 Ticket Algorithm

This algorithm is provided with a queue, NLBOC-Queue maintained in Node-BOC base class, see Figure 3.6. There is a ticket per NLBOC branch in each address space. This ticket is created by branch on zero ranking processor in a node and broadcasted to others in that node in constructor of NodeBOC class. In case that synchronization is required, to execute a method of the shared branch object, this ticket is needed. After owning the ticket only the processor can execute any method of shared branch object. If any of shared-object's methods is being executed, then the request for ticket will fail. The NLBOC-Queue is designed for failed requests to store the message of the request in. When ticket holder releases the ticket, it sends a specific node message named `token` to its node directed to `ticketReleased` method, which means "ticket

```
class NodeBOC : public groupmember {
private:
        TICKET *ticket;
        NLBOCQ *waiting_msgs;
        void *shared_obj;
        int tryTicket();
        void releaseTicket();
        void enqueueMsg(msg);
        void *dequeue();
protected:
        void initSharedObject(object *,callbackfn*);
        void exec(...);
        void exec2(...);
        void ticketReleased(token);
public:
        NodeBOC();
}
```

Figure 3.6: NodeBOC interface after Ticket Algorithm.

is released, check if there is any message left in our queue".

To clarify the distinction for synchronized execution NodeBOC class is supplied with one more method `exec2` in addition to normal execution method `exec`.

Then method2 of class `A` may be modified as shown below to ensure that method2 of shared object will not be executed concurrently in a node.

```
      void method2(MSG *msg)
          extract parameters from msg and produce param2
          exec2(&obj::method2, param2);
  }
```

In this algorithm using a ticket provides mutual exclusion and a queue keeps non-handled messages for future use. We might have such messages

# requests	Scheduler Queue	Node-Level Queue	Local Queue
100	564	562	340
500	2778	2876	1700
1000	5600	5828	3488
1500	8310	8710	5102
2000	10980	11580	6780

Table 3.1: Timings for queuing strategy (in millisecs) on a node of two processors. Each request takes $O(N^2)$ time, and two requests in a node are not executed concurrently(synchronization needed).

inserted back in receiver scheduler's queue or NLQ, to ensure that they will be handled. But this approach will cause schedulers to poll for same messages repeatedly until they can be handled. To demonstrate the performance gained via inner-node shared queue, we have compared performances of three different NodeBOC implementations: one having inner-node queue, one using NLQ, and last one using scheduler queue for keeping waiting messages. Table 3.1 illustrates results of this comparison. We are currently using this local queue for other purposes such as keeping messages directed to shared object if the shared object reference is not set yet on arrival of message.

3.5 Performance

We have conducted experimental applications to ensure about the usage and performance of our design and implementation of SMP support and NodeBOC base class.

3.5.1 Ring of Nodes

Scheduling communication on an SMP node is a point worth to take into account when designing applications for SMP Clusters. Several research activity is going on in this field. Work proposed in [8] addresses two policies for scheduling communication in an SMP node: `fixed`, where one processor is dedicated

```
class RingBOC : public groupmember {
private:
    ...
public:
    RingBOC(...) ;
    RingTurn(RingMsg* msg) {
       process the msg
      CBroadcastInNode(RingBOC,RingTurn,msg,thisgroup,CMyNode());
      CSendNodeMsgBranch(RingBOC,RingTurn,msg,thisgroup,nextnode);
    }
}
```

Figure 3.7: Simplified BOC class used in Ring example, which employs floating policy for communication-processor selection.

for communication in a node, and `floating`, where all processors alternately act as communication processor. The decision for choosing a policy is closely related with the application to be implemented.

In this particular experiment we compared the two policies when the application is the well-known ring operation, see Figure 3.7. Node-level messages allows us to implement floating policy as a software protocol, as well as fixed policy with standard messages of Charm++. A ring is intended to turn between nodes of an SMP cluster. A BOC object is created upon starting execution, and the flow of ring is achieved between branches of that object. The first approach achieves fixed policy using standard message send calls of Charm++, where processor ranking zero in a node will always receive and forward the message to next node's processor that has rank zero. The second approach employs floating policy, where any of the processors in a node receives the message, then forwards to next node using node-level message send calls. Another implementation of both approaches includes dummy work assigned to processors in a random manner independent of the ring turning. Results for this experiment are illustrated in Table 3.2.

msg size(bytes)	fixed	floating	fixed+dummy W.	floating+dummy W.
100	20	55	880	837.5
1000	75	60	1026.6	940
5000	850	730	1660	1700
10000	1270	1640	2520	2310

Table 3.2: Timings for Ring of Nodes (in millisecs) on one 2-processor and two single-processor nodes.

3.5.2 Broadcast, a collective communication primitive

NLBOCs can be used to implement efficient collective operations on SMP clusters. These include broadcast, reduction, and gather/scatter type collective communications. We have chosen broadcast operation for our experiments.

Broadcast operation can be optimized to take advantage of shared memory within SMP nodes. Across SMP nodes, a spanning-tree based algorithm can be used. In a k-processor SMP node, however, instead of k branches, only one NLBOC can handle work to be done. If the broadcasted data is read-only and large-sized, then by keeping one copy within the NLBOC and distributing pointers to the shared area to the objects that are the recipients of the broadcasted data can deliver better performance over ordinary broadcast operation. Table 3.3 shows results towards developing efficient collective operations for SMP clusters. The promising results encourage us in developing libraries or reusable patterns for implementing such algorithms and operations on SMP clusters.

The idea of providing an effective broadcast operation for SMP cluster programming may be extended to other communication operations, such as reduction. For the case of reduction operation, which involves processors within the system, optimizations can be achieved. In first phase of the operation, reduction takes place in each node on a selected processor within that particular node. Then the second phase will be executing reduction with the selected processors of all nodes involved. The final results may be broadcasted in each single node without overhead of inter-node communication.

msg size(bytes)	BOC based broadcast	NLBOC based broadcast
1000	6.68	5.94
5000	29.14	24.4
10000	56.85	45.57
20000	113.35	93.15

Table 3.3: Timings for Broadcast (in millisecs) on a 2-processor SMP node.

3.5.3 Simple Particle Interaction

Since our implementation of node-level messages allows any processor of node to pick up the message directed to the node it belongs, we have the potential to achieve dynamic load-balancing. If tasks mapped to a SMP node are atomic, then they can be shared between the processors of that particular node. An idle processor will detect any message on NLQ, and by picking it up will perform the requested task. In this experiment a simple application is carried out to simulate particle interactions due to gravitational force. Our application employs two level quadratic division of particle space. Each level-1 cell is then assigned to one processor. Number of particles in each cell may not be same for all cells, so loads of each processor may vary. An interaction manager object directs the processors through simulation of interactions between particles. Interaction between particles belonging to non-neighbor level-2 cells is approximated, which means particles that are far from a particle are thought to be just one virtual particle representing all of them. The atomic job is performing calculations for a level-2 cell basicly.

Object model employed Each level-2 cell is a sequential C++ object. Each processing node employs a special `compute-object`, that provides an interface to perform computations on cells. Compute-object in a node is shared between the processors of that node via use of a NLBOC object. Passing the reference for a cell to the compute-object is enough to have appropriate method of that cell to be called. Normal version is implemented using a BOC, each processor performs the computations for all of its level-2 cells without using the shared memory facility. On the other hand in SMP version processors in a node may share the computation of cells assigned to processors of that

# particles on p0	# particles on p1	T1	T2
200	250	499.7	391.9
200	300	783.5	558.4

Table 3.4: Timings for simple particle interaction (in secs) on a 2-node SMP node. T1 is completion time gathered from BOC version. T2 is completion time of NLBOC version through use of node-level messages.

node. Only one compute-object is employed in each node, and processors in that node have the reference for the compute-object. Since level-2 cells of two same-node processors lie in shared-memory, compute-object can access all cells in a node. This means work assigned to processors of a node may be shared. In our experiment we have changed load ratios of processors to observe the ability to share work Table 3.4.

Chapter 4

A Framework for NBody Algorithms on SMP Clusters

"Artificial life is about finding a computer code that is only a few lines long and that takes a thousand years to run - Rudy Rucker"

4.1 NBody Problem

The nbody problem is the problem of simulating the movement of a number of bodies under the influence of gravitational, electrostatic, or other type of force. The force acting on a single body arises due to its interaction with all other bodies in the system. The simulation proceeds over time steps, each time computing the net effect on every body and thereby updating its attributes. An exact formulation of this problem therefore requires calculation of n^2 interactions between each pair of particles. Typical simulations comprise of millions of particles. Clearly, it is not feasible to compute n^2 interactions for such values of n.

The n-body simulation problem, also referred as to as the many-body problem finds extensive applications in various engineering and scientific domains. Important applications of this problem are in astrophysical simulations, electromagnetic scattering, molecular biology, and even radiosity.

4.2 Algorithms and Related Data Structures

Many approximate algorithms have been developed to reduce the complexity of this problem. The basic idea behind these algorithms is to approximate the force exerted on a body by a sufficiently far away cluster of bodies with computing an interaction between the body and the center of mass (or some other approximation) of the cluster. Most of this algorithms are based on hierarchical representation of the domain using spatial tree data structures. The leaf nodes consist of aggregates of particles. Each node in the tree contains a series representation of the effect of the particles contained in the subtree rooted at that node. As bodies are grouped into clusters by the tree data structure, the interaction between leaf boxes, inner boxes, and bodies needs to interact with each other. A separation condition usually called as Multipole Acceptance Criteria (MAC) determines whether a cluster is sufficiently far away. Selection of appropriate MAC is critical to controlling the error in the simulation. Methods in this class include those of Appel [1] [7], Barnes-Hut [3], and Greengard-Rokhlin [13] [12] [11].

4.2.1 Barnes-Hut

The Barnes-Hut algorithm, based on a previous one by A.Appel in 1985, was proposed in 1986. Being one of the first algorithms in the field, it has been studied by many researchers. It addresses far field force in divide-and-conquer way.

Barnes-Hut algorithm is one of the most popular methods due to its simplicity. Although its computational complexity of $O(nlogn)$ is more than that of the Fast Multipole Method, which is $O(n)$, the associated constants are smaller for the Barnes-Hut method particularly for simulations in three dimensions. It uses quad-tree to store particle information in 2D, as opposed with oct-tree in 3D. The tree stands for the hierarchical representation of the global domain of all particles in the system. At the coarsest level root of the tree stands for the computational domain. Tree partitions the mass distribution of localized regions so that when calculating force on a given particle, tree regions near are detailly explored, and each distant region is treated as single virtual particle.

A cell is considered to be well-separated from a particle if

$$\frac{D}{r} = \frac{\text{size of box}}{\text{distance from particle to center of mass of box}}$$

is smaller than a parameter θ, which controls accuracy.

Serial Barnes-Hut Algorithm:

1. Built tree corresponding to domain
   At first step the tree is built, which means an hierarchy of boxes refining computational domain into smaller regions is created. Refinement level $l+1$ is obtained by subdividing each box at level l into two equal parts in each direction (4 for quad-tree, and 8 for oct-tree). Subdivision continues till each subcell has at most one particle. This property requires large amount of auxiliary storage.

2. Upward pass
   The tree is traversed in post-order, so that child cells of a cell are processed before it. The information of particles lying in the subtree rooted in an inner cell are reflected in that cell as center of mass and total mass.

3. Force Computation
   For each particle, or say leaf node, the tree is traversed to compute forces acting on that particle, due to others in the system. If the cell lies within the region defined by θ, then is said to be a near cell, and its child cells are traversed. Otherwise, the cell is thought to be a representative for subtree rooted at it, and is treated as a single virtual particle having mass of the total mass of particles lying in that subtree, and position of the center of mass due to particles in that subtree.

4. Update
   Due to the forces computed in previous step, attributes of particles are updated, and time step is advanced.

Barnes-Hut is effectively used for galaxy simulations in astrophysics. It is not as accurate as FMA, but simpler to implement.

A number of variants of the original Barnes-Hut algorithm have been implemented, such as by Barnes that allow better vectorization of the code at the cost of higher floating point operation counts. Salmon and Warren analyzed the performance of Barnes-Hut algorithm, and proved that its worst case errors can be quite large for its original θ criterion. They have defined a different method for deciding interacting cells. Using moments of the mass distribution within each cell provides better worst error case results for the same amount of computation.

4.2.2 Fast Multipole Algorithm (FMA)

FMA uses an octtree similar to that of the Barnes-Hut algorithm, except that leaf cells are permitted to contain a number of particles where this value is less than a constant m. The non-adaptive version builds a balanced tree, unlike Barnes-Hut. In the Barnes-Hut algorithm, interactions between bodies and sufficiently far away clusters are used to reduce the number of interactions from $O(n^2)$ to $O(nlogn)$ in the uniform case. FMA goes one step further by allowing interactions between two clusters. The effect of particles in a cell is reflected as a Taylor series called multipole expansion of that cell. Upon the tree construction, multipole expansions of leaf cell computed, then in a buttom-up manner tree is traversed, and multipole expansions of parent cells are constructed by shifting and adding the expansions of its children. After the tree is built, it has up-down pass in which the local expansion of the parent cell is shifted to the center of each child, and added to the multipole expansions of the cells in the child's interaction list, to form its local expansion. The number of terms in the multipole expansions control the accuracy of the algorithm. FMA has four different type of interactions, which are executed depending on certain conditions about the relative size and location of the two interacting nodes in the tree. Primary difference between the FMA and the Barnes-Hut lies in the fact that the Barnes-Hut algorithm computes particle-cell interactions, whereas the FMA computes cell-cell interactions, means reducing complexity. In fact it can be said that Barnes-Hut is a variant of FMA with order-0 multipole expansions. That is to say it uses monopole (center of mass) approximation.

4.2.3 Other variants

Several other researchers have implemented various n-body algorithms either from scratch or from existing algorithms. Among those most known ones are PMTA and Anderson's method.

- Parallel Multipole Tree Algorithm (PMTA) PMTA is a hybrid method of Barnes-Hut and FMA algorithms. It uses a rule similar to that of Barnes-Hut to determine well-separatedness of two cell. Two cells are said to be well-separated from each other if the size of the bigger cell divided by the distance between two cells is less than the parameter α, which corresponds to θ of Barnes-Hut. The tree is built as in Barnes-Hut method, but a cell is recursively subdivided until it contains no more than m particles , instead of one particle as in the case of the Barnes-Hut algorithm. During traversal of tree top-down for each leaf cell, if a cell is found to be well-separated from the leaf cell, its multipole expansion is translated into a local expansion about the center of the leaf cell, and the rest of the subtree below that cell is not visited. All the local expansions are added and the gradient is found to get the force due to the far far field on every particle in the leaf. The particles in the leaf cell interact directly with the particles in all cells that are not well separated from it. The number of terms p and the separation parameter α can be both varied to control accuracy. A theoretical error bound for this algorithm is not known.

- Anderson's Method (FMA without multipoles) The only difference between FMA and Anderson's Method is in the way they approximate the force field of a cluster of bodies. While FMA uses Taylor and Laurent expansions in 2D and expansions based on spherical harmonics in 3D, Anderson's Method is based on Poisson's formula. This makes it easier to implement, while it appears to be still unclear which method gives the better accuracy/performance trade-off.

4.2.4 Spatial tree structures

The original Barnes-Hut and FMA make use of quadtree to represent the hierarchy of the computational domain. Some researchers goes beyond this trivial representation especially when it is the parallelization of the algorithm they are studying.

- Quadtree (Octtree in 3D)
  The quadtree begins with a square in the plane, that is the root of the tree. This large square is broken into four smaller squares of half the perimeter and a quarter the are each. Each child can recursively divided into 4 subsquares till a predefined depth is reached. For non-adaptive methods this threshold for depth of the tree is employed, however adaptive methods recursively subdivide each non-empty cell till each leaf cell has at most m bodies, m is 1 for Barnes-Hut. The idea of using adaptive quadtree arises from the non-uniformity of the problem domain.

- Binary Tree
  Sanjeev et.al [19] proposed their modified FMA, which uses a binary tree produced by Orthogonal Recursive Bisection method, instead of regular FMA quadtree. The change in spatial representation of domain requires devising a new MAC, since the resulting cells of binary tree are not cubical anymore. A disadvantage of binary tree is its depth against octtree.

- Linear Array
  Hashed Octtree [22] proposed by Warren and Salmon, is a linear representation of the regular octtree. Key values are generated for particles/cells using their coordinate data. A hash function used to map this key values on the linear array. Conflicts in addressing the array are managed using linked lists. Infact this representation is developed for parallel implementations rather than serial algorithms.

4.3 Parallelization of Hierarchical Algorithms

Hierarchical n-body methods are that based on a insight look into the nature of body interactions, are being used to solve a wide variety of scientific and engineering problems. Such applications, however, typically have characteristics that make it challenging to partition and schedule them for effective parallel performance. In particular, the workload distribution and communication patterns are both nonuniform and also subject to change as the computation proceeds. This complicates the intention to provide load-balancing and data locality. Hierarchical radiosity [24] as an example is the most challenging application due to its nature causing it impossible to decide a static mapping of bodies among processing units.

Warren and Salmon proposed a fast message-passing implementation of the Barnes-Hut algorithm [26]. They have used ORB, which is described in section 4.3.1, partitioning technique to obtain both load-balancing and data locality. They propose the use of locally essential trees to obtain a purely sender-driven protocol for replicating nodes that are accessed by several processors during the force computation phase. This approach was refined by Liu and Bhatt [20] in their optimized implementation on the CM-5 machine.

In a later Barnes-Hut implementation [22], Warren and Salmon used a partitioning scheme based on space-filling curves of Morton ordering, and a distributed tree structure in order to achieve more flexibility in terms of application domains and MAC. A similar scheme is used in the shared-memory implementation by Singh et.al [23], who also performs a comparative study of several partitioning and load-balancing schemes.

4.3.1 Spatial Partitioning

The partitioning methods based on decomposing the space are classified as spatial techniques.

- Geometric/Uniform Partitioning
  This method uniformly subdivides the space into two equal sized subspaces in each dimension as it is for the regular Barnes-Hut and FMA trees. These subdomains keep the list of objects lying in their volumes. The main advantage of this kind of subdivision is that it lets fast traversal algorithms to be constructed to trace the tree. If the distribution of bodies is uniform, this simple approach may perform better than some advanced methods [10]. This partitioning of space directly fits into both Barnes-Hut and FMA, since the subspaces also correspond to cells of the octtree.

- Orthogonal Recursive Bisection (ORB)
  In each iteration the space is divided into two almost equal spaces in means of associated work estimate, which can be based on some kind of sampling or gathered after one iteration of the simulation. This subdivision process goes until a specified threshold value of number of subspaces is reached. Although it is an efficient method that preserves physical locality in problem domain, it tends to be complicated to built and maintain. Unlike geometric partitioning ORB has problems when applied to FMA, since this time the subdomain borders may not fit the octtree cells, which may cause some cells to be partitioned among processing nodes. Singh proposed a modified ORB for FMA that preserves the cell structure, while claiming that even this kind of additions to ORB may not be enough for FMA. In their work Sanjeev et.al uses the tree structures proposed by ORB, and they have changed the original idea of FMA about the spatial tree to generate. Their tree representation is a binary tree, which in fact is formed by ORB routine.

4.3.2 Tree Partitioning

- Costzones
  This technique developed for shared-memory architectures, benefits from the fact that the octtree already represents the spatial distribution. This idea led them to partition the tree instead of the space directly. With a subcell numbering scheme, they laid out the tree in two-dimensional plane. Each inner cell has an idea of the cost of the bodies under it. Using

this knowledge partitioning which preserves locality while providing load-balancing is tried be achieved. The globally shared tree is partitioned topdown.

- WS
  This technique may be called as a variation of costzones applied to distributed memory architectures. Since there is no globally shared tree in such architectures, the tree is partitioned bottom-up, using the associated cost values.

4.4 The Framework

"You know you have achieved perfection in design not when you have nothing more to add, but when you have nothing more to take away - ?"

This work was motivated while we were developing parallel object-oriented hierarchical nbody applications with Charm++. In this section, we will briefly discuss the common paradigm that we came across in those applications and the motivation behind developing a framework to exploit the features of the SMP clusters. [24] presented a parallel object-oriented approach for hierarchical radiosity on distributed architectures is presented. The main focus was to develop a framework such that the details of parallelization are hidden from the computational algorithm. The framework uses the idea of *proxy objects*, that are representative of remote objects.

With proxy patches, local representative of a remote patch, the design of the computational parts of the radiosity algorithm are greatly simplified, since they are freed from how the patches are distributed or when they migrate from one processor to another due to dynamic load balancing. As shown in Figure 4.1, interaction objects has no idea if the patches they are dealing with are local or remote. However, there must be an efficient mechanism, a proxy patch manager, which maintains proxy patches at required processors. The proxy manager is replicated on each node, and makes sure that proxy objects are created at the node where they are needed, and the manager exchanges

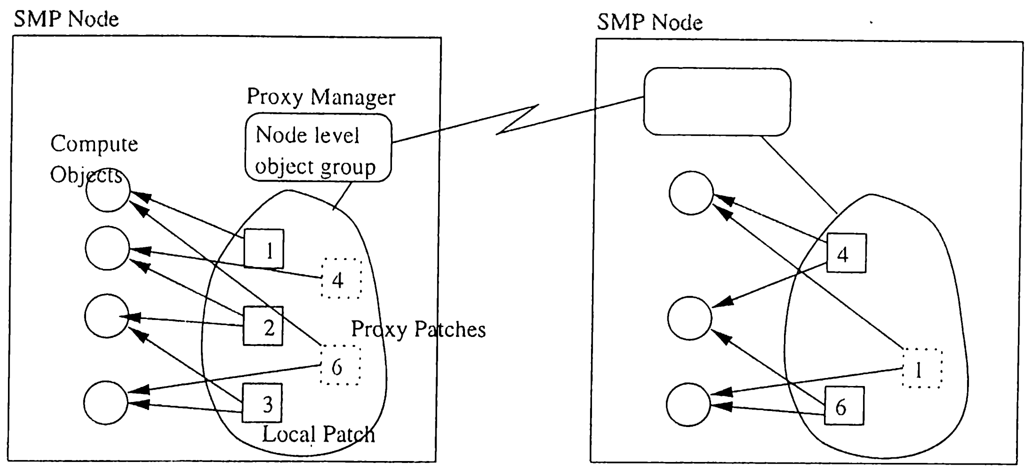

Figure 4.1: Node level object groups and proxies

information with other replicas to maintain consistency of the proxies. Secondly, the calculation objects can be run immediately whenever the patches they need are ready to be used.

If the underlying machine is an SMP cluster, and if the programming model treats the cluster as a network of processors, we end up using the it inefficiently. First of all, if an interaction needs a body assigned to a different processor within the same node, a proxy of the body need to be maintained at the processor where the interaction is calculated. However, on an SMP node, bodies can be accessed directly if they are in the same address space. Secondly, since interactions are assigned to processors, an idle processor cannot calculate another ready interaction which might be waiting for its processor to become idle. Using an object-based programming environment which supports a network of processors prohibits us to exploit these features. What we need is node level parallel objects, in addition to processor level ones.

Our research through supporting SMP clusters in programming as means of utilized reusable patterns got along together with providing a frame that covers

- distributed tree construction that includes bringing remote data in case of need for computation (proxy model).

routine	explanation
PMTAinit	Initializes the slave processing and creates various internal data structures. It requires initial application and system parameters.
PMTAregister	Performs registration of any slave processes that wish to make calls to PMTAforce(). No parameters at all.
PMTAforce	Performs the force calculations on an array of particle information. Number of particles and an array of particle information is passed as parameters for this function. Returns resulting force and potential values as arrays.
PMTAresize	Resizes the simulation cube.
PMTAvirial	Returns virial pressure tensor and potential.
PMTAexit	This routine should be called once by each process that called PMTAregister.

Figure 4.2: Interface calls of DPMTA library

- hiding communication details
- running on SMP clusters
- availability for different applications of nbody problem

4.4.1 Previous Work

Distributed Parallel Multipole Tree Algorithm (DPMTA) Library

The purpose of DPMTA [21] is to provide user applications with a flexible implementation of numerous multipole algorithms to compute N-body interactions for a variety of system sizes and particle configurations. It is the distributed version of PMTA algorithm mentioned in Section 4.2.3. The DPMTA code is written using PVM distributed computing tool-set and runs on a variety of platforms. Interface calls of DPMTA library are listed in Figure 4.2.

DPMTA designers tried to keep the programmer's interface to the DPMTA procedures as simple as possible. To this end, DPMTA provides four basic routines that perform initialization (2 routines), force calculation, and process cleanup. The particle data are supplied as simple arrays of floating point values which specify position and charge (or mass) for each particle. The DPMTA implementation makes no assumptions about the nature of the data beyond these values.

In order to support the integration of DPMTA into existing codes, two calling structures are provided.

- The first structure provides for passing all data to DPMTA from a single process. DPMTA will distribute the data among its processes, compute the resulting forces and potentials, collect the data, and return the results back to the calling process in a single array. This method provides a simple means to integrate DPMTA with existing serial codes, which can make sense when the N-body solve is the dominant time-consuming step of a program.
- DPMTA can also be called from an existing distributed application. Several processes may call DPMTA with each providing a subset of the particles and their associated data. DPMTA will redistribute the data among its own processes, compute the resulting forces and potentials, collect the results, and for each application process return only the results for the particles originally sent from that process. DPMTA makes no assumptions about how the data is partitioned across the calling processes. While this may result in some degradation of performance due to the overhead of particle redistribution, it vastly simplifies the application interface.

As with many other multipole codes, DPMTA decomposes the simulation space into an octtree representation. In the DPMTA implementation, the octtree structure is stored as a linear array and is addressed using the rapid indexing scheme of Warren and Salmon. Cells and their accompanying data are assigned to individual processes. All multipole and force calculations for an individual cell are accumulated by the process to which that cell is assigned. Cells are evenly distributed among the processors in spatially contiguous groups. In addition, the simulation space is equally divided among the processes, independent of the particle distribution.

M-Tree: A Parallel Abstract Data Type for Block-Irregular Adaptive Applications

M-Tree [27] is an hierarchical abstract data structure used to organized block-irregular computations generated by recursive domain-decomposition. It captures both the data structures and computational structures that are common to many adaptive problems. The M-Tree data structure itself is defined as below:

```
struct MeshTree {
       int status;
       struct Region domain;
       struct NeighborTree** nbTree;
       struct MeshTree* parent;
       MeshTree newlevel[rx][ry][rz];
       NodeType* vdata;
}
```

Each node represents a region of the domain and its subtrees are subregions overlaying the region of the parent node. A tree is called a quadtree when Rx=Ry=2 and Rz=1. So the structure above may be regarded as a generalization of quadtree that is gathered by recursive decomposition of space.

M-Tree is implemented as a C library based on MPI for messaging. Commonly used computation and communication patterns are tried to be extracted from particle-based problems. The set of functions covered by M-Tree are listed in Figure 4.3.

To use M-Tree, user needs to supply his own node data details, with functions to be executed to performs computations needed for any particular N-body problem. Some of the functions need communication stencils to be described by user also. As a last note on M-Tree; in order to provide efficient access to randomized tree nodes, a hashing scheme is used.

routine	explanation
MT_Init	uses userdefined partition operation to distribute global data to initialize MTree.
MT_Map_Leaf	applies userdefined operation to each leaf node in parallel.
MT_Map_Level	applies userdefined operation to each node on a given level in parallel.
MT_Reduce_Leaf	performs reduction for all leaf nodes in parallel.
MT_Reduce_Level	performs reduction for all nodes on a given level in parallel.
MT_Bcast_Leaf	performs broadcast to all leaf nodes in parallel.
MT_Bcast_Level	performs broadcast to all nodes on a given level in parallel.
MT_Up_Pass	traverses tree from bottom up and applies userdefined operations to nodes.
MT_Down_Pass	traverses tree from top down and applies userdefined operations to nodes.
MT_Adaptive	updates tree as required.
MT_Gather	collects elements into userspecified data structure.

Figure 4.3: Selected interface calls of M-Tree.

Comparison and Discussion

FMA and Barnes-Hut are the most popular hierarchical tree algorithms that are drawing attention in field of particle-based applications. Providing a library for such common algorithms with a simple-to-use interface may assist in developing larger applications such that namd [17], where electrostatic force computation is just a part of many other computations.

A library implementation for a class of problems is difficult in the sense that, common properties/patterns of such problems are needed to be extracted. Such properties should be used to define appropriate interfaces that will form up a frame. A library as a framework demands implementation of application-specific data types, and functions, which may be a problem for the user in case that the library is needed just as a part of a large application as mentioned above for namd2. This may cause the user to loose interest for using such a library, since details of implementation have to be dealt with. To avoid such user-oriented problems, implementation of particular applications as Barnes-Hut, and FMA may be included as layers above the abstract layer of the library. See Figure 4.4 for a simplified Barnes-Hut implementation.

```
... set up initial global-particles global-domain ...
for  (iter=0; iter<MAXITER; iter++) {
     Init(global-particles, global-domain, partitionFN);
     UpwardPass(centermassFN);
     ApplyLeaf(calculate_accelerationFN);
     ApplyLeaf(updatebodyFN);
     Gather(global-particles, global-domain);
}
```

Figure 4.4: A sample Barnes-Hut implementation using MTree.

Parameters for library routines contain user-implemented functions. We have to predefine the parameters/parameter-skeletons for user-defined functions. Let's take `updatebodyFN` of Figure 4.4 as an example which is a parameter for `ApplyLeaf` routine. There may be a predefined signature for that leaf-level function as it must take a parameter of node type pointer. This function then may update necassary fields of each particle that is covered by a leaf node, which is passed as a parameter.

DPMTA library may have a benefit over a FMA implementation using a generic library, in the sense that it specializes in the field. It provides a careful analysis of that particular algorithm. Integrating DPMTA to a particular application is not a big deal, since it is available for central-demand and distributed-demand applications, and it has 4 main routines to use. The parameters for these routines are strictly defined, and user does need to deal with the library but just calling routines.

Trying to keep everything general may cause to miss optimization options of particular applications. On the other hand, a possible generalization of particle based problems, may lead to an important step in such a challenging research field. Moreover such a library will surely provide an open environment that is flexible for particular applications and experimental study.

4.4.2 Providing an Interface for such Libraries

There are two possible interfaces that can be employed in such libraries. The first one is to supply an interface for the simulation steps as DMPTA does. This interface does not let the user to change any implemented part of the library unless a careful analyze through the source code is performed. The second one is to give access to low level functions, such that user specified or say implemented functions will be executed on the data structures supplied by the library. This latter approach is flexible since the tree is built and maintained by the library, and user just needs to change the functions to change the simulation from, say, gravitational force computation to electrostatics. We take the idea employed in second approach one step further, so that user will supply his tree representation, domain decomposition strategy, as well as application specific body and cell informations. The spirit of Charm++ dictates the asynchronous communication, so with an interface similar to MTree needs more to be done in our work. For example letting the user know that his specified computation functions is executed on all processors. This means that return functions are needed for library calls, so that user can view the flow as steps are accomplished. Another possibility may be hardwiring all simulation steps into the library. For example all nbody methods includes a bottom-up tree traversal after it is built, to initialize the cell data such as center of mass. If we provide a routine BottomUp(function to execute on cells), instead of requiring user's care for checking the completion of function, the library will guarantee to not to start the next step until BottomUp() finishes. But such an implementation will need careful analysis of in-library parallelization, between library steps that all need user-implemented functions, which is difficult to achieve.

Our Interface

The interface is formed up of 4 routines (Figure 4.5). The initialization routine LibInit() must be called from one process among the ones that will join into simulation by supplying particles. This routine takes just one parameter of type:

```
LibInitStruct {
              int numnodes;
              int* nodes;
              int numsources;
}
```

Within use from an application there may be cases where user does not want to have all nodes in the system to involve in the simulation. To ensure such a need LibInitStruct has two variables: `numnodes` is the number of nodes that user wants in the simulation, and `nodes` is an integer array which holds ids of the nodes that will participate in simulation.

In order to integrate the simulation into existing codes, two calling structures are provided: just one processor submits the particles that are subject to simulation, and for the other more than one processors may send particles. We have chosen to gather all particles in a central processor prior to processing. And this decision requires the library to know the number of distributed processes that will supply particles. The final variable `numsources` is used for getting the number of processors that will supply particles in the simulation. In order to have particles involved in simulation, the distributed processes should call LibParticles() routine. This routine packs particles of a process and then sends them to the central processor. Only `numsources` number of processors should call this routine.

LibIterate() routine should be called by all processes involved in simulation, and it provides a defined number of steps to be advanced prior to returning results to the specified return function. The type of this user-defined function is restricted as to be `void Functionname(int number_of_particles, Particle* particles)` The pattern imposed here for returning particles after iteration is a result of asynchronous nature of Charm++.

routine	explanation
LibInit(initdata)	initializes the library. The initialization includes building local trees, performing bottom-up pass, and remote domain test.
LibParticles(Particle*, int)	Processes call this function to deploy their particles to the simulation space.
LibIterate(retfn)	Iterates simulation one step, and returns the particles to the return function specified with `retfn`. All processes that supplied particles should call this function. If the initialization phase of simulation is not completed, SMPNodeManager does not iterate while ensuring to invoke this iterate request as soon as the initialization is complete.
LibKill()	Only one process should call this function to kill library processes prior to quitting from application.

Figure 4.5: Interface routines to the library.

4.4.3 Object Oriented Design

Starting from bodies we present the essential classes of our object oriented design in this section. The next section will be based on information provided here.

```
Particle {
        Vector pos; // position in space
        int nid; // owner processor of the particle
        BaseParticleInfo* info; // application specific info
}
```

Particle object has `position` property that is common to different type of bodies in nbody simulations. There may be cases that each node supplies its particles to the simulation. Therefore each particle's owner's id is kept with it to provide a safe return back after the simulation step(s). The interesting point about the particles is to supply their application dependent information, such as mass for gravitational force simulations. Application specific properties of a particle are enclosed in an object derived from `BaseParticleInfo` class. This base lets to keep a pointer in particle object, and essential just for type casting.

```
Cell {
     Vector center; // position of center in space
     Vector size; // size of cell
     BaseCellInfo* info; // application specific info
     Cell* parent;
     Cell* childs;
     ParticleArray* particles;
}
```

The provided cell class is designed to reflect properties of orthogonal cells. Since such a cell needs not to be cubical, it is provided with a `size` property. The concept of keeping application specific information of cells is managed via pointers to information objects. This time the base information class is not trivial as it was so for the particle objects. If the bodies that an interaction object needs are local (i.e. within the same processor where interaction is calculated) then the interaction can be completed without any communication. Otherwise, the data of the remote bodies must be brought, and then after the interaction calculation, the remote body data must be updated.

```
BaseLocal CellInfo : public BaseCellInfo {
          virtual void BottomUpPass(Cell*) = 0;
          virtual void Pack(PackedCellInfo* ) = 0;
}
BaseRemoteCellInfo : public BaseCellInfo {
          virtual BaseRemoteCellInfo* Unpack(PackedCellInfo ) = 0;
}
```

Our intention of using proxy pattern throughout the simulation, drives the class hierarchy providing classes for both local and proxy cells. `BaseLocalCellInfo` and `BaseRemoteCellInfo` classes arise from this fact. Both these two base classes are also derived from the `BaseCellInfo` class. Pack and unpack methods are required for communication. As we have stated in Section Charm++, message that are leaving an address space should not contain pointers. Such message classes should supply pack and unpack methods. Our message class can pack cells but the cell info packing needs assist from the user, since it's a user-defined class. Similarly this dependence to user executes for unpacking a packed cell. As will be mentioned later, these are the only communication related details that user should involve.

```
CellInfo AbstractFactory {
        virtual BaseLocalCellinfo* LocalInfoInstance() = 0;
        virtual BaseRemoteCellInfo* RemoteInfoInstance() = 0;
}
```

In library code it can be decided where to use which object with associated cell, but creating instances is not possible since C++ does not let class constructors to be virtual. To reach local and remote cell instances when needed, the *Abstract Factory* pattern [9] is employed. The concrete class derived from `CellInfoAbstractFactory` class should have two methods; one returning local cell info and the other returning remote cell info object.

```
BaseTree {
        Cell* root; // each tree has a root
        int childpercell; // number of child cells per cell
        virtual void InsertCell(Cell& cell) = 0;
        virtual ParticleLinkedList* Particles(int* ) = 0;
        virtual CellArray* LeafCells() = 0;
        virtual int NumChildPerCell() = 0;
}
```

BaseTree class serves as an interface that should be implemented in concrete tree classes. Deriving the tree structure is a case of matter as long as the interface is fulfilled. The idea behind this base class is to have a flexible environment for the spatial representation of the computational domain, which permits using OctTree, Binary Tree, etc.

```
BaseTree Constructor {
        BaseTree* ConstructTree( domain );
}
```

Since the inner structure of the derived class can not be known in library code, the tree can not be built in library routines. The following class solves this problem of building trees of user defined concrete classes. Since it is the user providing the derived tree class, then he should provide the necessary knowledge for constructing it.

```
BaseSpace {
        ParticleList* particles;
        BaseSpace* subspaces;
        int numSubspacesPerSpace;
}
```

Representation of the space is a decision directly effecting tree construction and spatial partitioning patterns. BaseSpace has particles, since the computational domain is itself simply a space instance. As we have chosen spatial partitioning as our strategy, each space instance needs pointers to its subspaces.

```
Spatial PartitioningStrategy {
        virtual Partition(BaseSpace* domain) = 0;
        virtual BaseSpace* Partition4Node(int node) = 0;
}
```

During the design of the framework, we had to make a decision in choosing the partitioning technique to use. Parallelization of tree partitioning is a problem since we do not want to make assumptions about the concrete tree class provided by the user. Moreover, the nature of tree partitioning may cause a parent and its child cells to be distributed over many processors. WS is a good example for tree partitioning techniques on distributed architectures. It is obvious that, if a tree structure which uses pointers for accessing subtrees will cause problems, since pointers will loose their meaning across address spaces. In their work, WS solved this problem by changing the tree representation resulting in a linear distributed array, which they call Hashed OctTree [22]. Such an alternative that is dictating its tree structure is not desirable for a general purpose framework. Having this fact in mind we have provided `SpatialPartitioningStrategy` base class to user, which is an abstract class defining an interface for its descendant classes.

```
ComputeObject {
              virtual Compute(void*, void*) = 0;
}
```

`ComputeObject` abstract class is designed to enable user to plug his computation functions into the library. Nbody algorithms computes at most 3 interactions in type: particle-particle, particle-cell, cell-cell. If the application to be created using the library employs two of them as Barnes-Hut does, two classes should be derived from the base, each dedicated for one type of interaction. Parameters passed to Compute method are in void* type, in order to avoid any difficulties while trying to distinguish between 3 type of interactions. This lets as to supply just one base class for all computation objects.

```
Library {
virtual CellInfoAbstactFactory* CellInfoFactory() = 0;
virtual BaseTreeConstructor* TreeConstructor() = 0;
virtual SpatialPartitionStrategy* PartitionStrategy() = 0;
virtual BaseSpace* DomainInstance(ParticleLinkedList*) = 0;
virtual void PackSpace(PackedSpace*) = 0;
virtual void UnpackSpace(BaseSpace*, PackedSpace*) = 0;
virtual CellArray* MACTreeSpace(BaseTree*, BaseSpace*) = 0;
virtual bool MAC(Particle*, Cell*) = 0;
virtual void PackCellInfo(BaseCellInfo*, PackedCell) = 0;
virtual BaseParticleInfo* ParticleInfo(PackedParticleInfo) = 0;
virtual void PackParticleInfo(BaseParticleInfo*, PackedParticle) =0;
virtual ComputeObject* PPComputeObject() = 0;
virtual ComputeObject* PCComputeObject() = 0;
virtual ComputeObject* CCComputeObject() = 0;
}
```

Among all the presented classes, this is maybe the most important one, since this class serves as a gateway between abstract library layer and user-defined discrete classes. Instantiating just one object of this type, infact of derived class type, provides access to specialized objects and methods through the whole simulation. In-library objects that has something to do with user objects, has a pointer for Library instance. Through use of this instance requirements occurred for accessing concrete classes of user are fulfilled. Library is a base class that forces its descendants to provide methods that give access to user-coded classes.

Parallel Objects and Flow of Simulation

SMPNodeManager

SMPNodeManager is the heart of the simulation, which itself is a grouped object derived from NodeBOC.

```
SMPNodeManager :  public NodeBOC {
                SMPNode* mynode;
                Library* library;
}
```

The data that we need to share among processors of a node is enclosed in SMPNode object, which is a node-level shared object.

```
SMPNode :  public gen_shared_object {
        BaseSpace* GlobalDomain;
        int numLocalDomains;
        BaseSpace* LocalDomains[];
        int numRemoteDomains;
        BaseSpace* RemoteDomains[];
        BaseTree* LocalTrees[];
        BaseTree* RemoteTrees[];
}
```

Tree of subspaces resulting from the domain decomposition is used as the globally shared part of the global tree, which is not built infact but local trees of nodes comprise the global tree. This shared part of the global tree makes sense in computation since we are freed from communication to receive upper part of the tree while traversing for computation. This global domain representation in a node does not contain particles in the domain, except for local domains that are assigned to particular node. Also having remote domains in each node lets us to achieve sender-initiated communication for creating proxy cells.

SMPNodeManager has an Library object, that will provide it access to user-defined concrete classes during execution.

RemoteCellManager

```
RemoteCellManager :  public NodeBOC {
                  SendQueuee* sendQ;
                  int numRemoteDomains;
                  BaseTree* RemoteTrees[];
}
```

The sender-initiated communication pattern is executed by SMPNodeManager in co-operation with RemoteCellManager. The deposited cells are kept in the SendQueue of this manager, and just as the deposit is complete, RemoteCellManager branches starts inter-node communication. SendQueue is the node-level shared object of this NLBOC class. Receiving RemoteCellManager unpacks cells and inserts them in remote trees for further use during computation.

ComputeManager

```
ComputeManager :  public NodeBOC {
                ComputeObject* ppCompute;
                ComputeObject* pcCompute;
                ComputeObject* ccCompute;
}
```

Computational structure of nbody applications requires particle-particle, particle-cell and cell-cell interactions to be computed. ComputeManager forms an interface between representation of domain and computation. It does not update particle or cell properties, rather through computation objects required calculations are performed.

When a request for computation of an interaction is received by ComputeManager, it just invokes the associated computation object.

Flow of Simulation

It is the SMPNodeManager that drives these steps in simulation. During the simulation SMPNodeManager co-operates with both RemoteCellManager to send cells to remote nodes and ComputeManager to perform required calculations. NLQs of SMPNodeManager, RemoteCellManager, and ComputeManager virtually forms a work-pool in each node, which is filled with atomic tasks that are infact node-level messages sent to parallel objects. Since any of the idle processors in a node may pick a node-level message, these messages form a shared work-pool.

Spatial partitioning takes place as a first step of the whole execution. Since central-processor manages this step, all particles are submitted to it, and as particles are received in that processor the computational domain is formed and partitioned. Then each node is assigned with a number of subdomains according to the user implemented partitioning strategy. The partitioning info sent to nodes contains the knowledge of remote and local domains for each particular node. To avoid all particles to be sent to all nodes, only subdomains local to a node carry the list of particles while being transmitted to it by the central processor.

When partitioning is complete, all SMP nodes joined to the simulation receive this partition information with subspaces assigned to it. Then in each node; for each local subdomain a corresponding local tree needs to be built, after performing bottom-up pass on that tree, it is tested against remote domains to decide which cells of it to send to the owners node of that domains. Upon completion of this sender-initiated communication, the environment will be ready for computation phase of the simulation.

During inititialization phase, as the nodes receive the spatial partitioning info, work-pool in the node starts to be filled with tasks. It is a known fact that these tacks must be atomic so that they can be independently executed. Since subdomains are subject to the tree construction, and each node is expected to have a number of subdomains assigned, we can claim that building tree of a local subdomain is an atomic task. This approach allows all processors in node to join in tree building provided that the partitioning scheme assigns subdomains in accordance to the number of processors within a node.

Once a local tree is built, it should be traversed from leaf cells to the root, in bottom-up manner to set the approximated representation of bodies lying in the subtrees rooted at each cell, the representations are center of mass and total mass for gravitational Barnes-Hut for example. Traversing a local tree is independent from other processes, so it can be placed in the work-pool. The processor that picks this task from the pool, performs the bottom-up pass. A tree construction and a bottom-up pass can be executed concurrently for different trees.

As a local tree is traversed, it can now be tested against remote domains. Since each node knows the global partitioning, and which subdoain belongs to which remote node, sender-initiated communication can be used to prohibit request-send pattern, which will produce extra communication. Testing a local tree against a remote domain is achieved using the user-defined MAC-CellSpace() method, which returns the list of cells need to be sent to the owner of the remote domain. This process itself another atomic task, which can be thrown into the work-pool. As can be guessed, while tree building, and traversals continue remote domain test on built and traversed trees can take in place concurrently with other tasks. When a processor tests a tree with a remote domain, it gathers a list of cells to send. SMPNodeManager deposits this list of cells are deposited to RemoteCellManager, where they are all enqueued. As being a node-level grouped object RemoteCellManager has a branch on each node, and provides inter-node communication via these branches. The aim of enquing cell lists is to avoid from disadvantages of a number of communication for a few number of data. When the test and deposit execution completes, RemoteCellManagers are triggered to send deposited cells to the intended destination nodes. Instead of sending all data of cells, just computationally required portions of them are sent. Moreover all cells deposited for a target node, are packed together and send at once. When the branch of RemoteCellManager of the target node receives the cells, it unpacks them to form remote tree representations made up of proxy cells. Once all send-receive of cells is accomplished, the initialization phase is said to be done.

If the LibIterate routine is called by application, as described in Section 4.4.2, the computational phase of the simulation starts. SMPNodeManager branches initiates the computations on local tree cells. As an interaction is formed, SMPNodeManager triggers the ComputeManager to perform the computation required by that particular interaction. Since each interaction can be computed independently, the work-pool of node is now filled with the interactions prepared to be computed. All processors in a node will take place in this computation by picking interactions from the pool.

4.5 Using the Framework : A case study, the Barnes-Hut algorithm

"Anyone who uses the phrase 'easy as taking candy from baby' has never tried taking candy from a baby - Unknown"

4.6 Application

In order to demonstrate the usability of our framework we have implemented Barnes-Hut algorithm to compute gravitational force among astrophysical bodies. With a careful examination of original Barnes-Hut code, we have extracted the properties and method implementations for user-defined classes. Acting as a user, we have provided all required classes, and tested the library on our only SMP machine.

Deriving classes for Barnes-Hut Algorithm
A complete listing of required concrete classes: ParticleInfo, LocalCellInfo (packed info), RemoteCellInfo (packed info), CellInfoFactory, Tree, TreeConstructor, Space (packed space), Spatial partitioning strategy, Compute Objects, Library object. Among these only packed info structs are communication related. As mentioned in Chapter 2, messages needs to be packed and unpacked. Since cell info classes are not known from the library point of view, user should supply the pack structs.

- ParticleInfo
  Particles, or say bodies, have mass, velocity, acceleration, and potential attributes in gravitational force computation. One can ask about the force being exerted on a single particle. In this computation we will use a physical fact that the gradient of potential gives the force. Since calculation of potential on p1 due to p2 is easier than calculating force that is not scalar as potential.

- CellInfo
  LocalCellInfo and RemoteCellInfo distinction has nothing to do in this

application, since center of mass and total mass values are common to all inner cells, whether local or proxy.

- Tree and Its Constructor

  Being devoted to original Barnes-Hut implementation, we use octtree as our tree object. This decision affects cells such that each cell has at most 8 child cells.

- Space and Spatial Partitioning Technique

  Kumar et.al. [10] proposed a simple spatial partitioning technique for their Barnes-Hut implementation, which can be extended for use with FMA type of algorithms. Recursively subdividing the computational domain using geometric center until a threshold value is reached for depth of division, produces a tree structure describing the domain. The depth of this space-tree can be decided according to the uniformity of bodies existing in the domain. This structure is used in mapping the subspaces to processors, or nodes for our work. Using geometric coordinates in division lets each subspace infact correspond to a subtree for the oct tree used in Barnes-Hut. Using this structure as a globally shared part of the global tree reduces the overhead in accessing this part of the tree during computation. This is especially essential for Barnes-Hut algorithm since the global tree needs to be traversed for each leaf cell in order to compute forces acting on a single body. Moreover advanced load-balancing techniques such as costzones proposed by Singh et.al. [23] may be applied in mapping the subspaces to processing units.

  ORB can also be replaced with geometric partitioning if we have an appropriate interface for space and partitioning classes.

- Compute Objects

  There are two interaction types in Barnes-Hut algorithm; particle-particle and particle-cell. Therefore supplying two computation object BHPP-Compute and BHPCCompute with Compute methods implemented as algorithm dictates, is enough for us.

- Library

  BHLibrary class derived from Library, supports all required methods of its interface. While employing original θ criterion as our MAC, we have also implemented the required MAC function that will operate on cell and space pairs. Recall that this function of Library class is required to satisfy sender-initiated communication. That is when comparing a local cell with respect to a remote space, we assumed the worst case to be true; there is a particle in space at the point that is closest of all for the examined cell. If the original MAC is satisfied for the cell and assumed particle, the cell is decided to be sent to the node that is owner of that remote space.

4.7 Usage and Preliminary Results

Processor having id zero is set as the central processor in the simulation. And it is the only processor that is submitting particles. As LibInit() is called, the library is set up and as the particles are deposited initialization phase starts. The last call LibIterate is queued till the initialization ends. The return function is `mainiterate` and it loops over time steps calling LibIterate again and again. Since the interface is simple to use, creating such an application is not a big deal. But rather it may be annoying to supply required concrete classes.

```
void main(int argc, char** argv) {
    read in particle data from input file
    LibInit(..);
    LibParticles(...);
    LibIterate(1,&mainiterate);
}
void mainiterate(int nparts, Particle* particles) {
    while (tnow < tstop) {
    update particle positions and velocity;
    LibIterate(1,&mainiterate);
    advance time;
    }
}
```

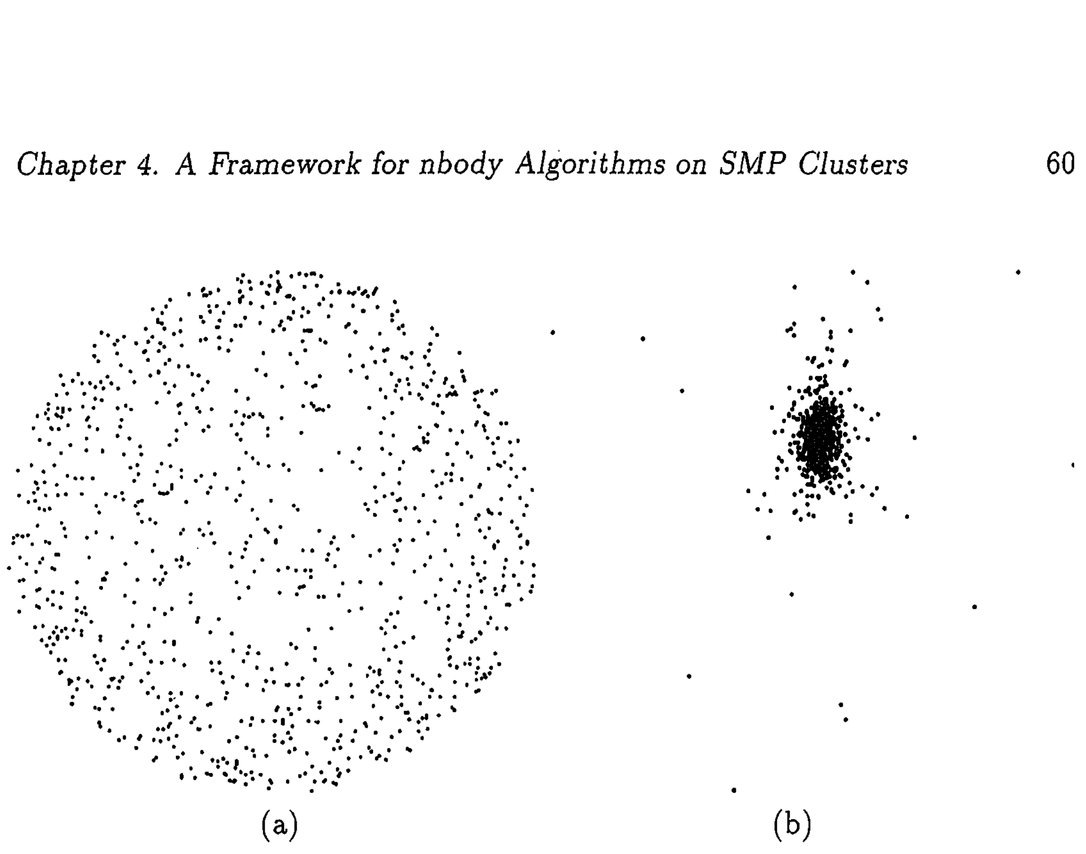

Figure 4.6: 1024 bodies in (a) Uniform Sphere (b) Plummer distribution.

We have currently one SMP node with 2 processors at hand. The performance tests managed using Charm++ environment property of starting as many processes as wanted on a processing node. If the number of processes exceeds the number of processors in the node, then more than one process may be assigned to each processor. To not to enforce the physical limitation of our SMP node, we simulated a two-node cluster each having two processors on it. For the alternative architecture usage of our machine, 4 single-processor nodes each with its own memory segment are simulated. Since all processes are running in physically same memory, we had a memory bottleneck for testing with large number of bodies. Moreover, since there was not an actual network but the memory, differences between array of processors and cluster of nodes are minimized. So the performance study may not be very reasonable, since we couldn't test the application on a real cluster of SMP nodes.

We have used two different test input models for the distribution of particles in the space; uniform sphere Figure 4.6-a and plummer distribution Figure 4.6-b. For uniform sphere with 100 particles SMP execution completes in 4.52 seconds, where it is 5.13 seconds for array of processors execution. And when we increase number of particles in sphere to be 250, SMP run takes 8.47 seconds, while the other takes 8.77 seconds. Plummer distribution with 100 particles running on SMP clusters takes 3.96 seconds, and running on processor array completes in 4.3 seconds. Finally for 250 particles distributed in Plummer model, execution times are 8.8 and 8.93 respectively for SMP and processor array configurations. We see slight performance gain, but, again these are preliminary results and performance tunning in both nbody frame and possibly in NLBOC implementation will be needed.

Chapter 5

Conclusion

"Even if you persuade me, you won't persuade me - Aristophanes"

A programming model which treats n node k-way SMP clusters as a network of nk processors might prevent us to extract maximum performance from SMP clusters. In this thesis, we studied programming mechanisms, in an object oriented programming environment, that will allow us to exploit features of SMP nodes: (a) sharing physical memory within a node , (b) and dynamic load balancing within a node. We introduced *Node Level Object Groups* (NLBOC) to implement collective operations such as broadcast, ring communication, and reduction NLBOCs allows interactions within a node to be done through the shared objects and allows any idle processor to invoke methods of shared objects at the node level. The communication across branches at different nodes are automatically handled by asynchronous method invocations.

For many parallel applications, interactions with shared objects often requires exclusive accesses. A common way to enforce exclusive method invocation is to use locks. When a processor picks a message for a shared object and finds that the object is locked (that is, another method is already being executed by some other processor within the node), the message can be put back in the node level message queue to be processed later. However, other idle processors can keep selecting and and putting back the message in the message queue which results in loss of useful processor time. We have developed an algorithm to prevent such cases. Each NLBOC is augmented with a private message queue. If a processor finds the NLBOC locked, then, it enqueues the message into the NLBOC message queue. When the execution of the method (which is holding the lock) is finished, a control message put back into the node level message queue. This mechanism greatly reduced the conflict on the shared message queue.

We have also emphasized other reusable patterns or libraries for collective communications and computations that can be used in many parallel algorithms.. We designed an object oriented framework to support fast algorithms for nbody problem. The framework hides details of communication from the programmer and allows the programmer to exploit SMP clusters. The use of advantages of SMP nodes (mentioned above) has been achieved by NLBOC abstraction.

In this thesis, we set out a way to program SMP clusters effectively. Our future work includes improvement on the performance issues and implement more libraries for common parallel computations. We believe that the node level object groups will be widely used to implement such libraries for SMP clusters.

Bibliography

[1] A. W. Appel. An efficient program for many-body simulation. *SIAM Journal of Computing*, 1985.

[2] David A. Bader and Joseph Jaja. Simple: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (smps). Technical report, UMIACS, May 1997.

[3] J. Barnes and P. Hut. A hierarchical o(nlogn) force calculation algorithm. *Nature*, 1986.

[4] A. A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. Zhang. Evaluating high level parallel programming support for irregular applications in icc++. Department of CS, University of Illinois at Urbana-Champaign.

[5] Department of CS, University of Illinois at Urbana-Champaign. *Charm++ Programming Manual*, 1996.

[6] Department of CS, University of Illinois at Urbana-Champaign. *Converse Programming Manual*, 1996.

[7] K. Esselink. The order of appel's algorithm. *Information Processing Letters*, 41:141–147, 1992.

[8] Babak Falsafi and David A. Wood. Scheduling communication on an smp node parallel machine. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, Feb 1-5 1997.

[9] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] A. Grama, V. Kumar, and A. Sameh. Scalable parallel formulations of the barnes-hut method for n-body simulations. In *Supercomputing'94 Proceedings*, 1994.

[11] L. Greengard. The rapid evaluation of potential fields in particle systems. *ACM Press*, 1987.

[12] L. Greengard and W. Gropp. A parallel version of the fast multipole method. *Parallel Processing for Scientific Computing*, pages 213–222, 1987.

[13] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comp. Physics*, 73:325–348, 1987.

[14] A. Gursoy and I. Cengiz. Mechanisms for programming smp clusters. In *Proceedings of International Conference on Parallel and Distributed Techniques and Applications*, July 1999.

[15] L. V. Kale, M. Bhandarkar, N. Jagathesan, and S. Krishnan. Converse: An interoperable framework for parallel programming. In *Proceedings of IPPS'96*, pages 212–217, 1996.

[16] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object-oriented system based on c++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, ACM Sigplan Notes, pages 91–108, Sep-Oct 1993.

[17] L. V. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Shulten. Namd2 : Greater scalability fro parallel molecular dynamics. *Journal of Computational Physics*, 1998.

[18] L.V. Kale and A. Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proceedings of 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 738–743, 1995.

[19] S. Krishnan and L.V. Kale. A parallel adaptive fast multipole algorithm for n-body problems. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 46–51, August 1995.

[20] P. Liu and S. N. Bhatt. Experiences with parallel n-body simulations. In *6th Annual ACM SPAA'94*, pages 122–131, 1994.

[21] William T. Rankin and Jonh A. Board jr. A portable distributed implementation of the parallel multipole tree algorithm. Technical report, Department of Electrical Engineering, Duke University, 1995.

[22] W. Salmon and J. Salmon. A parallel hashed oct tree n-body algorithm. In *Proceedings of Supercomputing Conference*, 1993.

[23] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27:118–141, Jun 1995.

[24] Resat Sireli. Parallelization of hierarchical radiosity algorithms on distributed memory computers. Master's thesis, Department of Computer Engineecring and Information Science, Bilkent University, January 1999.

[25] Y. Tanaka, M. Matsuda, M. Ando, K. Kubota, and M. Sato. Compas: A pentium pro pc-based smp cluster and its experience. In *IPPS WorkShop on PC-NOW'98*, volume 1388, pages 486–497. Springer-Verlag, 1998.

[26] M. Warren and J. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing Conference*, 1992.

[27] Q. Wu, A. J. Field, and P. H. J. Kelly. M-tree: A parallel abstract data type for block-irregular adaptive applications. In *EuroPar'97 Parallel Processing*, 1997.