CHARISMA: A COMPONENT ARCHITECTURE FOR PARALLEL PROGRAMMING

BY

MILIND A. BHANDARKAR

M.S., Birla Institute of Technology and Science, Pilani, 1989
M.Tech., Indian Institute of Technology, Kanpur, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Abstract

Building large scale parallel applications mandates composition of independently developed modules that can co-exist and interact efficiently with each other. Several application frameworks have been proposed to alleviate this task. However, integrating components based on these frameworks is difficult and/or inefficient since they are not based on a common component model. In this thesis, we propose a component architecture based on message-driven in-process components.

Charisma, our component architecture, has Converse message-driven interoperable runtime system at its core. Converse allows co-existence of in-process components with implicit data-driven control transfers among components. Message-driven objects, based on Charm++, provide encapsulation, and a uniform method of accessing component services. Although, the Charm++ model allows coexistence and composition of independent modules, it is not adequate for independent development of modules. We describe an interface model for Charisma based on the *publish-require* paradigm.

Pure message-driven components lack in expression of control-flow within the components. One way to clarify expression of control flow within a component is by introducing threads. However, overheads associated with threads cause inefficiency. We have developed a notation, Structured Dagger, for building message-driven components that retains the message-driven nature of components efficiently without using threads.

Support for legacy codes is vital in the success of any new programming system. We describe how legacy components written using message-passing paradigm could be converted to use Charisma. Our efforts are based on AMPI, our implementation of the MPI library on

top of Charm++.

To Vidul.

Acknowledgments

I would first like to thank my adviser, Professor Laxmikant Kale, for his guidance and support during my doctoral research. He was a constant source of encouragement, support and guidance. It has been a pleasure to work with him for the last eight years. I would also like to thank Prof. Michael Heath, Prof. David Padua, and Prof. Eric deSturler, who agreed to be on my thesis committee, and for providing valuable suggestions.

I would like to particularly thank Prof. Michael Heath for giving me an opportunity to be associated with the Center for Simulation of Advanced Rockets (CSAR) as a research programmer.

Thanks to my colleagues at the Parallel Programming Laboratory: Orion Lawlor, Gengbin Zheng, Terry Wilmarth, Jay DeSouza, Sameer Kumar, Arun Singla, Joshua Unger, Chee Wai Lee, Guna, Mani. They have provided valuable inputs to this work. Orion and Gengbin particularly helped me a lot by taking over the responsibilities of maintaining the Charm++ system, leaving me free to devote time for my research. Thanks also to my ex-colleagues at Theoretical Biophysics Group, Robert Brunner and Jim Phillips, who have been instrumental in keeping the Charm++ software in shape at all times. I wish to express my appreciation to my colleagues at CSAR: Michael Campbell, Jim Jiao, Jay Hoeflinger, Ali, and Prasad for providing codes and working with me for their conversion to Adaptive MPI.

I would like to thank my parents, who encouraged me in all my endeavors. Their guidance and support has always been very valuable asset for me. I would like to mention the considerable understanding shown by my kids, Shivani and Shaunak, that belies their age. The most important of all, I am forever indebted to my wife, Vidula, for everything. She has been with me every step of the way through this arduous process without a complaint. This thesis would not have been possible without her.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Developing scalable parallel applications for complex physical system simulations is a difficult task. Increased computational power and capabilities of parallel computers have led to increased expectations from parallel applications. High fidelity simulations with greater sophistication are needed in order to meet these expectations. Multi-scale and multi-physics simulations are emerging as solutions to achieve high fidelity in complex physical system simulations. Applicability of different physics, computational techniques, and programming models demands that these programs be developed as a largely independent collection of software modules. Several frameworks such as POOMA [6], Overture [32], SAMRAI [35], ALEGRA [17], ALICE (Advanced Large-scale Integrated Computational Environment [57, 26]), and SIERRA [72] have been developed to facilitate the implementation of large-scale, parallel, simulations. These frameworks typically focus on a limited set of techniques and applications. No single framework is suitable for all numerical, computational and software engineering techniques employed and under development in academia and research laboratories. Thus, one would need to employ different application frameworks for different modules of a complex application. However, it is currently impossible to efficiently couple together the software components built using different frameworks. This is mainly because these application frameworks have not been built with a "common component model".

As an illustrative example of such a complex simulation application, we are developing an integrated multi-physics rocket simulation code from several stand-alone pieces, at the Center for Simulation of Advanced Rockets (University of Illinois). The individual codes include a computational fluid dynamics (CFD) code (*Rocflo*), and a structural analysis code (*Rocsolid*). These codes are coupled together with an interface code (*Rocface*). Together, these codes simulate solid propellant rockets. *Rocflo* solves the core flow equations in the inner part of the rocket. *Rocsolid* models the deformation of the solid propellant, liner, and casing. *Rocface* takes care of data transfer and interpolation of temperature, pressure, and displacement between these two components.

Several researchers have been working on different physics of the rocket as part of the center, and several additional codes to model these physics, such as combustion (*Rocburn*), crack propagation (*Rocfrac*), and behavior of aluminum particles ejected into the core (*Rocpart*) have been developed. These codes have to be integrated to perform a complete multi-physics simulation of the rocket. While some of these new codes have to be integrated more tightly within the individual applications (such as ejection of aluminum particles to *Rocflo*), some of them (such as crack propagation) need to interface with both *Rocflo* and *Rocsolid*. Some of them need efficient coupling with the existing codes, since the interaction among them will be frequent (such as between *Rocflo* and the combustion code.)

These codes are being developed more or less in isolation (many are based on legacy codes) by different researchers. One of the approaches for integrating them has been to run them as separate applications, while interactions among them are serialized by dumping interface data on the file system or through socket-like mechanisms[1]. But given the nature of the required integration, this approach results in inefficient coupling for large systems. While we want the individual software components to be modular and independently developed, these components may not always have large enough grainsize to ignore the efficiency of coupling. Also, in order to be scalable to a large number of processors, especially while solving a fixed-size problem, it is required that the coupling efficiency be maximized. Therefore, it

[1]Even in this approach it was necessary to make all code developers agree on a common format for data exchange, specifications for invoking various components, and had a glue code that binds all these components together in a single "application".

is imperative that the communicating software components be part of the same process in order to have efficient coupling between software components (known in the component terminology as the "in-process components" or "inproc servers" [68].) One of the approaches taken earlier for developing the integrated rocket simulation is to have a single orchestration module that invokes these codes as in-process subroutines, while these codes interact with each other using shared data. This approach forces a common programming paradigm on individual codes, while requiring them to have intricate knowledge of each other's data structures and control transfer strategies, thus hampering modularity.

The current approach for integration of rocket simulation codes is based on *Roccom* [37]. *Roccom* is a component architecture motivated by the need to integrate different components of the rocket simulation code together into a single application. *Roccom* client components can exist as independent processes or within the same process. These components register data such as the mesh geometry, connectivity, and values of various physical parameters on the mesh entities with *Roccom*, which facilitates exchange of these data among components. *Roccom* has a prespecified set of data types (that are required for physical simulations) that can be exchanged between components. If a new type of data needs to be exchanged, such as in the case of integrating a component code that models the physical domain as an oct-tree, one has to extend *Roccom* to support it. We believe in the need of a general purpose component architecture, which does not limit the domain of parallel applications to physical simulations. In addition, one can easily implement domain-specific component architectures such as *Roccom* on top of such general purpose architectures.

This thesis describes our work in building a general-purpose component architecture, *Charisma*, for parallel applications with in-process components. Goals of such a component architecture are:

- Coexistence of in-process components

- Efficient interaction among components

- Ease of development of components

- Migration path for existing codes

In this thesis, we present work on each of these aspects of Charisma.

1.1 Component Architectures

Software components allow composition of software programs from off-the-shelf software pieces. They aid in rapid prototyping and have been used mainly for developing GUIs, database as well as Web applications, and have been found very successful in building these software applications. Object-oriented programming aids in developing software components because of its support for encapsulation, and in separating interface from implementation. Based on these principles, various component models have been developed. Of these, a few such as Microsoft's COM [68], OMG's CORBA [64], and Sun Microsystems' JavaBeans [58] have become popular as distributed application integration technologies. A software component is a set of objects with published interfaces, and it obeys the rules specified by the underlying component model. A component model specifies rules to be obeyed by components conforming to that model. A component model along with a set of "system components" defines a component architecture.

Current component technologies cross language barriers, thus allowing an application written using one language to incorporate components written using another. In order to achieve this, components have to provide interface specification in a neutral language called Interface Description Language (IDL). All component models use some form of Interface Description Language. Though these IDLs differ in syntax, they can be used to specify almost the same concept of an interface. An interface consists of a set of "functions" or "methods" with typed parameters, and return types. These functions or methods can be synchronous (caller waits for return of method) or asynchronous (one-way). Caller of a method is referred to as "Client", whereas the object whose method is called is referred to as "Server". Client and Server need not reside on the same machine if the underlying component model supports remote method invocation.

Remote method invocation involves marshaling input parameters (or serializing into a message), "stamping" the message with the component and method identifier, dispatching that message, and in case of synchronous methods, waiting for the results. On the receiving side, the parameters are unmarshalled, and the specified method is called on the specified component (actually an instance of the component). A similar technique is employed for creating instances of components, or locating components, by invoking methods of system component instances, which provide these services. The server machine, which contains the server component instances, employs a scheduler (or uses the system scheduler) that waits continuously for the next message indicating a method invocation, locates the specified component instance, verifies access controls, if any, and calls the specified method on that component instance. Other services, such as security, privacy, accounting, logging etc are either built into the scheduler, or are available as system services that are invoked as preprocessing and/or post-processing stages for each method invocation.

1.2 Limitations of Current Component Architectures

Demand for rapid prototyping and cross-project reuse of object libraries has stimulated growth of component architecture in industry. Individual companies (e.g. Microsoft, Sun) and consortia (e.g. OMG) have developed different component architectures to meet this demand in the distributed computing world. Component architectures in the distributed programming community (such as COM, JavaBeans, CORBA) do not address efficiency issues that are vital for coupling parallel software components. Though all of them support in-process components, they incur overheads unacceptable for the needs of parallel component integration. Microsoft COM has limited or no cross-platform support necessary for the emerging parallel computing platforms such as the Grid [23]. JavaBeans have no cross-language support, and needs components to be written only using Java, while FORTRAN dominates the parallel computing community. CORBA supports a wide variety of platforms and languages, but does not have any support for abstractions such as multi-dimensional arrays.

Parallel computing communities in academia and various U.S. national laboratories have recently formed a Common Component Architecture forum [22] (CCA-forum) to address these needs for parallel computing. Common Component Architecture (CCA) [5] is one of the efforts to unify different application frameworks. The CCA approach tries to efficiently connect different applications developed using various frameworks together by providing parallel pathways for data exchange between components. The CCA Forum has developed a Scientific Interface Description Language (SIDL [21]) to allow exchange of multi-dimensional arrays between components, and have proposed a coupling mechanism (CCA-ports) using *provides/uses* design pattern. For parallel communication between components, they have proposed collective ports that implement commonly used data transfer patterns such as broadcast, gather, and scatter. In-process components are connected with Direct-Connect ports, which is close in efficiency to a function call. However, these component method invocations assume blocking semantics. Also, CCA restricts itself to SPMD and threaded programming paradigms, and does not deal with coexistence of multiple non-threaded components in a single application. Hooking up components dynamically is listed among future plans, and it is not clear how the efficiency of coupling will be affected by that.

The most suitable way of combining multiple components using CCA is by developing wrappers around complete application processes to perform parallel data transfer, delegating scheduling of these components to the operating system. Heavyweight process scheduling by the operating system leads to coupling inefficiencies. If the communicating components belong to different operating system processes even on the same processor, invoking a component's services from another component requires an inefficient process context-switch. For example, on a 500 MHz Intel Pentium III processor running Linux, invocation of a "reflector service" (that returns the arguments passed to it) takes 330 microseconds when the service is resident in another process, while it takes 1.3 microseconds for service residing within the same process (Corresponding times on a 248 MHz Sun Ultra SPARC workstation running Solaris 5.7 are 688 and 3.2 microseconds respectively.)

1.3 Charisma Approach

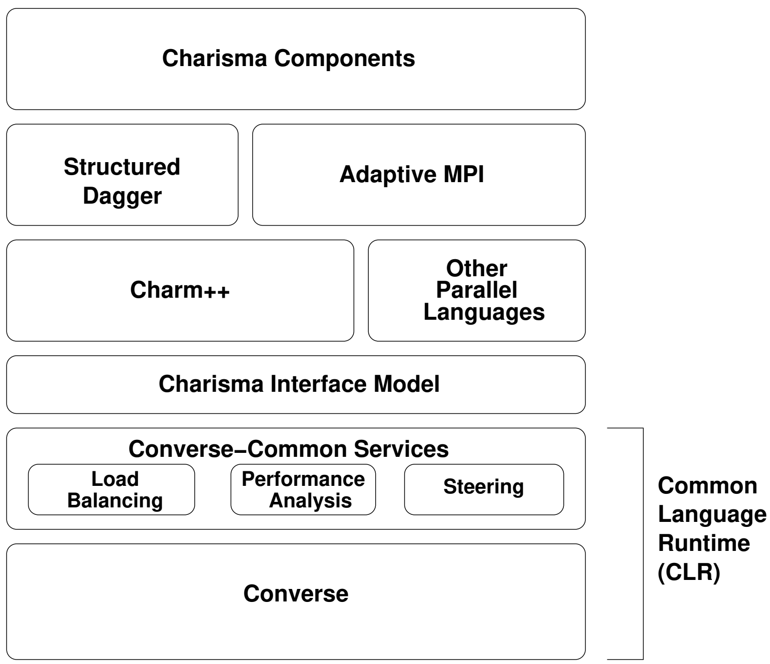


Figure 1.1: Overview of Charisma architecture

Our approach of "in-process components" eliminates the inefficiency resulting from separate application processes. Charisma does not require the data exchange to be serialized and control-transfer between components is close in efficiency to a procedure call. It maintains the independence of the individual components, while providing uniformity of data exchange.

An overview of Charisma architecture is shown in figure 1.1. At the core of Charisma is Converse, a message-driven interoperable parallel runtime system that allows parallel software components based on a variety of programming paradigms to co-exist in a single application. Along with common component services such as dynamic load balancing, computational steering, and performance profiling and analysis, Converse provides a common language runtime (CLR) for Charisma. The common interface model of Charisma is based on data-driven control transfer, and allows independent development of reusable components. Charm++ [44] adds encapsulation and object-virtualization to the message-driven programming model of Converse. However, intra-component control-flow expression is complicated for message-driven components written in Charm++. Control-flow expression is simplified by implementing a coordination language, Structured Dagger on top of Charm++. Another method to express control-flow is by using threaded message passing. Adaptive MPI implements threaded message-passing using familiar MPI syntax. Reusable Charisma components can be built using any of these languages. Adaptive MPI also provides a migration path for converting existing MPI codes to Charisma components. In the following sections, we summarize the building blocks that make up Charisma.

1.3.1 Common Language Runtime

As a result of ongoing research in parallel programming, a number of programming paradigms have been proposed and implemented. Some of them, such as message-passing, shared variables, and data-parallel have become popular, and have a large software base. A component architecture needs to be able to incorporate components written using these programming paradigms, as well as other lesser-known paradigms that are suitable for specific tasks. In addition, for effective utilization of resources, parallel components should concurrently interleave their execution. This is referred to as *Parallel Composition Principle* [25], which states:

> "For effective composition of parallel components, a compositional programming language should allow concurrent interleaving of component execution, with the order of execution constrained only by availability of data."

Since these programming paradigms differ in the amount of concurrency within a process and the way control is transferred among different entities of the programming model, their coexistence and interoperability among them is a non-trivial task. We have demonstrated that the message-driven programming paradigm is appropriate for implementation of these other paradigms efficiently, and facilitates coexistence and interoperability among these paradigms. For this purpose, we have developed a runtime system called Converse based on message-driven parallel programming paradigm. Converse employs a unified task scheduler for efficiently supporting different concurrency levels and "control regimes". Converse is described in detail in chapter 2.

In order to efficiently utilize the processor resources, especially in the presence of dynamic runtime conditions, the CLR needs to offer a common set of services to all the programming paradigms implemented using it. Some of the services Converse offers are automatic load balancing, both at the time of creation of components, and in the middle of a run; automatic and efficient checkpoint and restart service; a common performance tracing service; and a computational steering service. These services, while important and influential to the design of Charisma, are not central to this thesis, and have been described elsewhere [16, 40, 66, 65, 70].

1.3.2 A Common Interface Model

While the common language runtime along with a common set of services allows us to have multiple "in-process components" within an application, it does not specify how different components interact with each other. For this purpose, we need an interface model that allows components to access other components' functionality in a uniform manner. Since Converse is a message-driven runtime, traditional interface description languages do not perform well because they assume semantics of a blocking procedure call. Other shortcomings of traditional interface models have also been noted in literature [2]:

> "Current component interfaces are based on functional representation of services provided by a component. ... this model fails to describe many important features such as locality properties, resource usage patterns,..."

Enhancing the traditional interface languages to allow asynchronous remote procedure calls results in other problems, such as proliferation of interfaces. We have developed a different interface model based on separate input and output interfaces, which enables us to overcome these problems. In this interface model, each component describes the output data it publishes, and the input data it accepts on a set of named and typed ports. Computations in the components are attached to the set of input ports. This method of attaching computations to incoming data is performed by the component internally, and while being a close match with the underlying Converse runtime system, it does not introduce any foreign concepts to the programming paradigm of the component. For example, an input port for a component written using MPI may be accessible to the component as a ranked processor within a special communicator. A standard library of interface components needed for composition of parallel applications as well as an orchestration language – a high level scripting language that utilizes this interface model to make it easy to construct complete applications from components – is also developed. Charisma interface model is described in chapter 3.

1.3.3 Simplifying Component Development

Since the CLR is message-driven, it presents some challenges in efficient implementation of programming paradigms such as message passing that use blocking receives. Charm++, an object-based message-driven language provides encapsulation and object-virtualization necessary for large scale component software development, and being a close match to the underlying CLR, can be efficiently implemented. However, a Charm++ component is simply a set of computations attached to input ports, and expression of intra-component control-flow is complicated. Therefore, for simplifying development of message-driven software components using Charm++, one needs to simplify expression for intra-component control-flow. One may express the control-flow within a component using threads but it adds unacceptable overheads in the form of thread creation, synchronization and context switching costs. This is especially apparent in components with small grainsize. To express control-flow within components, without forsaking efficiency and the ability to interleave execution of different components within a single process, we have developed Structured Dagger, a coordination mechanism for message-driven computations. Structured dagger provides structured constructs for expressing dependencies between computations of a component. A translator translates structured dagger code into message-driven code that implements the control-flow. It is described in detail in chapter 4.

1.3.4 Migration Path

While it is possible to develop message-driven components that take full advantage of all the services offered by our component model, it is necessary to provide a migration path for existing parallel applications to be "componentized". As an illustrating example, we have developed a migration path for converting parallel MPI applications into Charisma components. For this purpose, we have to address two issues. First, an MPI application has to co-exist with other components as an in-process component. Second, they have to interact with other components using our common interface model. Adaptive MPI (AMPI), our thread-based implementation of MPI, originally designed for providing dynamic load balancing for MPI programs, serves the first purpose. Using AMPI, MPI applications are encapsulated into a thread-group with thread-private data. AMPI threads communicate with each other using the standard MPI messaging calls. We have added *cross-communicators* to AMPI, which allow an MPI application to communicate outside the component using the familiar syntax and semantics of MPI messaging. Cross-communicators allow us to provide a binding to Charisma interface model without introducing alien concepts to MPI. This serves the second purpose in migrating MPI codes to Charisma. AMPI is described in chapter 5.

The summary of the argument of this thesis is pictorially represented in figure 1.2.

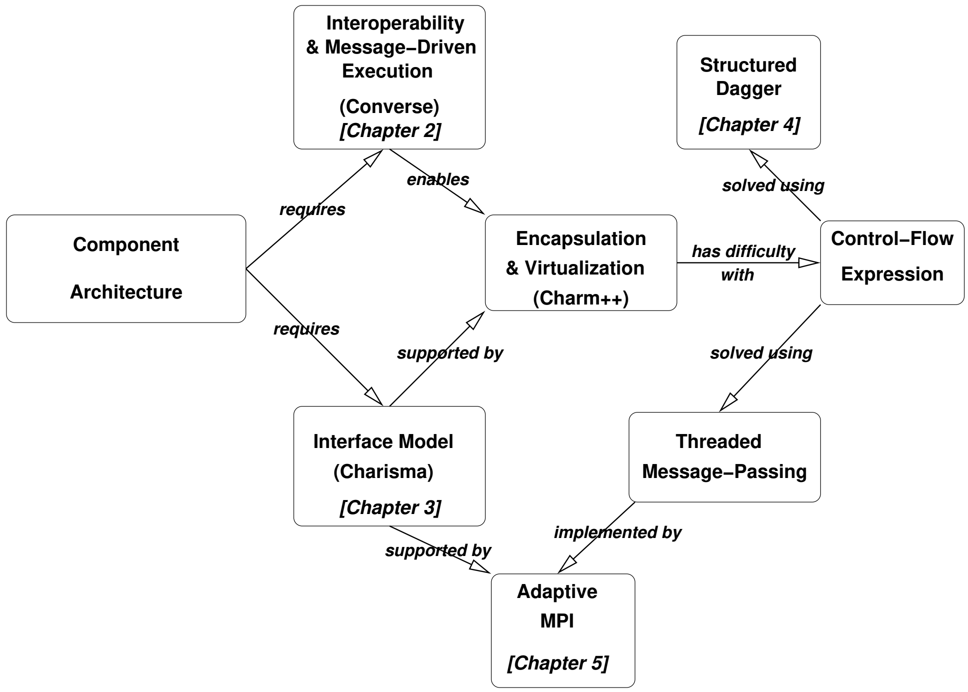


Figure 1.2: Pictorial summary of the thesis.

1.4 Contributions of thesis

Some of the work reported here has appeared earlier in the literature [39, 38, 47, 41, 12, 11]. The main contributions of this thesis are:

- Demonstration of the effectiveness of *message-driven parallel runtime system* in seamlessly supporting the coexistence of multiple software modules, possibly based on different programming paradigms.

- An *interface model* that encourages modular development of applications yet allows tighter integration of application components at run time.

- Mechanisms that allow *clear expression of control-flow* within software modules without sacrificing efficiency, in the presence of a message-driven runtime system.

- *Migration path* for existing parallel applications to simplify conversion to the new component architecture.

Chapter 2

Converse: A Message-Driven Runtime for Charisma

Research on parallel computing has produced a number of different parallel programming paradigms such as message-passing [71, 27], data-parallel [33, 34, 15], object-oriented [44, 18, 49], thread-based [29], macro-dataflow, functional languages, logic programming languages, and combinations of these. However, not all parallel algorithms can be efficiently implemented using a single parallel programming paradigm. It may be desirable to write different components of an application in different languages. Also, cross-project reuse of software components is possible only if pre-written components can be integrated into a single application without regard to the programming paradigms used for building those components. For this, we need to support interoperability among multiple paradigms.

This section describes *Converse*, an interoperable framework for combining components written using multiple languages and their runtime libraries into a single parallel program, so that software components that use different programming paradigms can be combined into a single application without loss of efficiency. Converse provides a rich set of primitives to facilitate development of new languages and notations, and supports new runtime libraries for these languages. This multi-paradigm framework has been verified to support traditional message-passing systems, thread-based languages, and message-driven parallel object-oriented languages, and is designed to be suitable for a wide variety of other languages.

2.1 Interoperability among parallel languages

Parallel languages and their implementations differ from each other in many aspects. The most important differences from the point of interoperating with components written using different programming paradigms are the number of concurrent tasks allowed within a process and the way control is transferred between these tasks.

2.1.1 Concurrency within a Process

The first aspect that is critical from the point of view of interoperability is how the language deals with concurrency within a single process. Concurrency within a process arises when the process has a choice among more than one actions at some point(s) in time. There are three categories of languages in this context:

No concurrency: Some parallel programming paradigms (such as traditional message-passing) do not allow concurrency within a process. Each process has a single thread of control, hence a process will "block" while it is waiting for a message that has not yet arrived. During this blocking, the semantics require that there should be no side effects, when the "receive" system-call returns, beyond the expected side effect of returning the message.

Concurrent objects: Concurrent object-oriented languages allow concurrency within a process. There may be many objects active on a process, any of which can be scheduled depending on the availability of a message corresponding to a method invocation. Such objects are called message-driven objects.

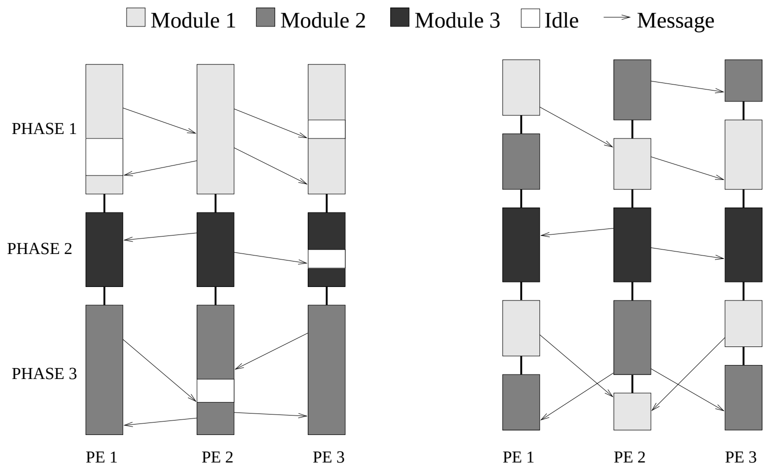
Multithreading: Another set of languages allows concurrency by threads— they permit multiple threads of control to be active simultaneously within a process, each with its own stack and program counter. The threads execute concurrently under the control of a thread scheduler.

Most languages can be seen to fall within one of these three categories or combinations of them, as far as internal concurrency is concerned. Other paradigms such as data parallel languages and functional languages can be implemented in one of the above categories. For example, HPF can be implemented using a statically scheduled SPMD style or using message-driven objects [54].

2.1.2 Control Regime

Another related aspect is the control regime for a language, which specifies how and when control transfers from one program module to another within a single process. Modules interact via *explicit* and *implicit* control regimes. In the explicit control regime, (as described in Figure 2.1(a)) the transfer of control from module to module is explicitly coded in the application program in the form of function calls. Moreover, at a given time, all processes are usually executing code in the same module. Thus all processors execute modules from different languages in different, non-overlapping phases. All processors transfer control to the next module only when the current module has completed all its work — there are usually no outstanding messages. This control regime is suitable for languages that have no concurrency within a process.

In the implicit control regime (Figure 2.1(b)), different parallel software components execute in an overlapped manner, so that entities in different modules can be simultaneously active on different processes. The transfer of control from module to module is implicit. Rather than being decided only by the application program, it may be decided dynamically by a scheduling policy in the runtime system. This regime allows for adaptivity in execution of application code with a view to providing maximal overlap of modules for reducing idle time. Thus, when a thread in one module blocks, code from another module can be executed during that otherwise idle time. The implicit control regime is suitable for languages with concurrent objects or multi-threaded languages.



(a) Explicit Control Regime (b) Implicit Control Regime

Figure 2.1: Control regimes for parallel programs

2.2 Design of Converse

After determining the necessity of handling the different models of concurrency and control regimes in single-threaded modules, message-driven objects, and thread-based modules, the following guidelines were used in the design of Converse:

Completeness of coverage: The interoperability framework should be able to efficiently support most approaches, languages and frameworks for parallel programming. More concretely, any language or library that can be portably implemented on MIMD computers should be able to run on top of Converse and interoperate with other languages.

Efficiency: There should not be any undue overheads for (a) remote operations such as messages, and (b) local scheduling such as the scheduling of ready threads, as compared to the cost of such operations in a native implementations.

Need based cost: The Converse framework must support a variety of features. However, each language or paradigm should incur only the cost for the features it uses. For example, an application that requires sophisticated dynamic load balancing might link in a more complex load balancing strategy with its concomitant overhead, while another application may link in a very simple and efficient load balancing strategy.

An important observation that influenced the design of Converse is the fact that threads and message-driven objects need a scheduler, and a single unified scheduler can be used to serve the needs of both. Apart from the central scheduler, the other components of Converse are: A machine interface (CMI), Thread objects, and load balancers, as shown in Figure 2.2. When initialized, a language runtime registers one or more message-processing functions (called *handlers*) with Converse. These language-specific handlers implement the specific actions they must take on receipt of messages from remote or local entities. The language handlers may send messages to remote handlers using the CMI, or enqueue messages in the scheduler's queue, to be delivered to local handlers in accordance with their priority. The Converse scheduler is based on a notion of schedulable entities, called "generalized messages".

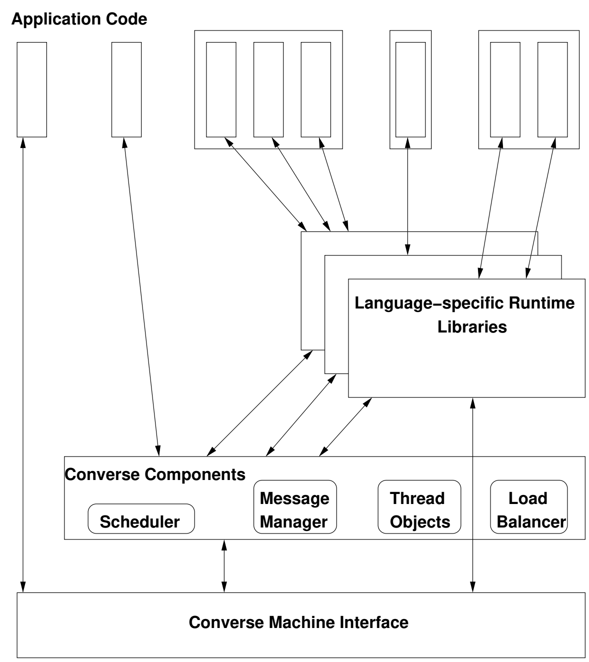


Figure 2.2: Software Architecture of Converse

2.2.1 Generalized Message Scheduling

In order to unify the scheduling of all concurrent entities, such as message-driven objects and threads, we generalize the notion of a message. A generalized message is an arbitrary block of memory, with the first few bytes specifying a function that will handle the message. The scheduler dispatches a generalized message by invoking its handler with the message pointer as a parameter. Any function that is used for handling messages must first be registered with the scheduler. When a handler is registered, the scheduler returns a handle to the function. When a message is sent, this handle must be stored in the message using macros provided by the scheduler.

There are two kinds of messages in the system waiting to be scheduled — messages that have come from the network, and those that are locally generated. The scheduler's job is to repeatedly deliver these messages to their respective handlers. Since network utilization issues demand timely processing of messages from the network interface, the scheduler first extracts as many messages as it can from the network, calling the handler for each of them. The handler for a particular message may be a user-written function, or a function in the runtime of a particular language. These handlers may enqueue the messages for scheduling (with an optional priority) if they desire such functionality, or may process them immediately. After emptying messages from the network, the scheduler dequeues one message from its queue and delivers it to its handler. This process continues until the Converse function for terminating the scheduler is called by the user program.

Converse supplies two additional variants of the scheduler for flexibility. The first allows the programmer to specify the number of messages to be delivered. The second, called the "poll mode", runs the scheduler loop until there are no messages left in either the network's queue or the scheduler's queue. For modules written in the explicit control regime, control stays within the user code all the time. However, for modules in the implicit control regime, control must shift back and forth between a system scheduler and user code. For these apparently incompatible regimes to coexist, it is necessary to expose the scheduler to the module developer, rather than keeping it buried inside the run-time system. A single-threaded module can explicitly relinquish control to the scheduler to allow execution of multi-threaded and message-driven components. This control transfer need not be exposed to the application, however. For example, an MPI implementation on Converse may internally call the scheduler in poll mode when any of the MPI functions are called, allowing other computations to complete while the MPI process is waiting for a message.

The pseudo-code for these variants of the scheduler is shown in figure 2.3. `CsdScheduler` implements the implicit control regime, `CmiGetSpecificMsg` implements programming paradigms with explicit control regime.

2.2.2 Converse Machine Interface

The Converse Machine Interface (CMI) layer defines a minimal interface between the machine independent part of the runtime such as the scheduler and the machine dependent part, which is different for different parallel computers. Portability layers such as PVM and MPI also provide such an interface. However, they represent overkill for the requirements of Converse. For example, MPI provides a "receive" call based on context, tag and source processor, and guarantees in-order delivery of messages. This is unnecessary for some applications. CMI is minimal, yet it is possible to provide an efficient MPI-style retrieval on top of it.

The CMI supports both synchronous and asynchronous variants of message send and broadcast calls. For retrieving messages that have arrived on the communication network, the CMI provides the call `CmiDeliverMsgs`, which invokes handlers for all messages that have been received from the network. For supporting single-threaded languages, which may require that no other activity take place while the program is blocked waiting for a specific message, the CMI provides a `CmiGetSpecificMsg` call, which waits for a message for a particular handler while buffering any messages meant for other handlers. The CMI provides a number of utility calls including timers, atomic terminal I/O, and calls to determine the logical processor number and the total number of processors.

Converse messaging primitives are geared toward implementing other programming languages, rather than being used directly by the user program. An illustrative example of such primitives is the set of novel *vector-send* primitives. Message passing primitives in languages often accept user data, concatenate a header onto that data, and then send the header and

```
void CsdScheduler()
{
  while ( 1 ) {
    CmiDeliverMsgs() ;
    if (exit called by handler)
      return;
    get a message from the scheduler queue ;
    call the message's handler ;
    if (exit called by handler)
      return;
  }
}

void CmiDeliverMsgs()
{
  while ( there are messages in the network ) {
    receive a message from the network ;
    call the message's handler ;
    if (exit called by handler)
      return;
  }
}

void CmiGetSpecificMsg(int spechdlr)
{
  while ( 1 ) {
    if ( there is message in the network ) {
      receive a message from the network ;
      call the message's handler ;
      if ( handler == spechdlr )
        return;
    }
    if ( scheduler queue not empty ) {
      get a message from the scheduler queue ;
      call the message's handler ;
      if (handler == spechdlr)
        return;
    }
  }
}
```

Figure 2.3: Pseudo-code for scheduler variants

data together to remote processors. For example, the MPI messaging routines accept a pointer to data, but they also specify a "communicator" object and an integer tag. The communicator, tag and the data are then transmitted together. The act of concatenating a header to message data often requires copying the header and data into a new buffer. This is inefficient. To help the implementers of such routines, Converse provides a set of *vector-send* primitives. These primitives accept an array of pointers to data. Each element in this array represents a piece of a message to be delivered. The pieces of data are concatenated and sent, but the concatenation is often done very efficiently, without the overhead of copying. For example, on machines where messages are packetized, the packets are simply gathered by scanning the array and packetizing each piece separately. On machines with shared memory, messaging is sometimes implemented by copying a message from the sender's memory to the receiver's memory. In this case, vector send is implemented by concatenating straight into the receiver's memory. All of these optimizations are done transparently to the Converse user by the implementation of the vector-send primitives, thus reducing programming complexity.

Converse Machine Model

Converse assumes the machine to be a collection of *nodes*, where each node consists of one or more *light-weight processes* sharing memory (some systems call these processes "kernel threads".) Each process operates with its own scheduler, and messages are usually directed to processes, not nodes. Each process houses an arbitrary number of Converse threads (see section 2.2.3.) This model covers all the memory hierarchies in existing machines, from distributed memory machines at one end to the fully-shared machine at the other, with clusters of SMPs in-between.

Most procedural programming languages supports global variables (global in scope, not shared across processors). Compilers for distributed-memory hardware typically give each processor a separate copy of the globals, but compilers for shared-memory hardware typically make only one copy for every shared memory node. Thus, attempts to use global variables tends to create portability problems. Converse solves this by adding explicit control over sharing of global variables. Converse provides macros to denote, initialize, and access thread-private, processor-private, and shared variables. Each Converse thread has its own copy of the thread-private variables. Each processor has a copy of the processor-private variables, and all the threads housed on that processor share the copy. Each node has a copy of the shared variables, which is available to all the entities in that node.

2.2.3 Threads

In many parallel programs, each process has a single thread of control: they have a single stack and a single set of registers. However many complex programs are difficult to express in a single threaded manner. This is particularly true for programs that involve asynchronous events, or when it is necessary to overlap computation and communication. In thread-based programs, there are multiple threads of control, with the scheduler switching between threads to enable maximum utilization of processor resources and to ensure progress.

A threads package typically consists of three components: A mechanism to suspend the execution of a running thread and resume the execution of a previously suspended thread; A scheduler that manages the transfer of control among the threads; and concurrency control mechanisms. Many thread packages and standards have been developed in the past few years [36, 59, 53]. However, the gluing together of scheduling, concurrency control and other features with the mechanisms to suspend and resume threads is problematic for the requirements of interoperability. E.g. the particular scheduling strategy provided by the threads package may not be appropriate for the problem at hand. Converse separates the capabilities of thread packages modularly [45]. In particular, it provides the essential mechanisms for suspending and resuming threads as a separate component, which can be used with different thread schedulers and synchronization mechanisms, depending on the requirements of the parallel language or application.

2.2.4 Dynamic load balancing

Modules in all languages affect the load on a processor, hence Converse supports load balancing across language modules. The need for load balancing arises in parallel programs in many contexts.

A particular situation of interest is when the program creates a piece of work or a task that can be executed on any processor. This is referred to as "seed-based" load balancing. The load balancer assigns the task to a processor depending on the load measures on other processors at that point in the program. A language runtime may hand over a "seed" for a task, in the form of a generalized message, to the load balancer on any processor. The load-balancing module moves such seeds from processor to processor until it eventually hands over the seed to its handler on some destination processor.

Another situation occurs in the presence of dynamic runtime conditions, where either the application itself induces load imbalance (such as in physical simulation applications that use adaptive mesh refinement) or when the application is run on non-dedicated platforms (such as clusters of workstations). Converse supports migration-based load balancing to handle such situations, where work (represented by the entities specific to different programming models) is migrated to lightly loaded processors from heavily loaded ones.

Dynamic load balancing based on object migration is an area of active research [16] and is orthogonal to our thesis objectives. However, the need to provide information to the load-balancing subsystem about locality among component interactions has influenced our interface model design (see section 3.3.)

2.2.5 Utilities

Multilingual interoperability is productive only when languages and paradigms can be quickly implemented to utilize components written using those paradigms. In order to make it convenient to implement new languages, several convenience modules are provided with Converse.

These represent the commonly needed abstractions to simplify the task of implementing runtime systems for parallel languages.

The *message manager* is a table object for storing messages according to a set of integer tags. It supports variable numbers of tags, and wild cards in the lookup process. It can also be used to store any data that must be indexed by integer tags. The *futures* library implements the futures abstraction [30] in a library-form. It provides a future "object", with methods to fill the object remotely, get its value, and block until the value has been filled. The *Converse Parameter Marshalling* system is a small C preprocessor and code generator that produces remote function-invocation code. One inserts the keyword `CpmInvokable` into a C source file in front of a function definition. The CPM preprocessor scans the C file, and generates code to invoke that function remotely. The CPM-generated code automatically packs up the arguments into a message, sends them to the destination, and invokes the specified function. The *POSIX threads* API has been implemented on top of Converse threads. These POSIX threads can interoperate with Converse threads, and the rest of the Converse system.

2.3 An Example Language Implementation

This section shows the implementation of CSM, a message-passing library. CSM was designed for illustration purposes. So, it is intentionally the simplest possible library that implements message-passing with threads. The basic design shown here can be used to implement any message-passing library, including MPI or PVM. The following function descriptions are from the CSM manual:

`void CsmTSend(int pe, int tag, char *buffer, int size)`

A message is sent to the given processor `pe` containing `size` bytes of data from `buffer`, and tagged with the given tag. The calling thread continues after depositing the message with the runtime system.

`int CsmTRecv(int tag, char *buffer, int size, int *rtag)`

Waits until a message with a matching tag is available, and copies it into the given buffer. A wild card value, `CsmWildCard`, may be used for the tag. In this case, any available message is considered a matching message. The tag with which the message was sent is stored in the location to which `rtag` points. The number of bytes in the message is returned.

Our implementation buffers messages on the destination processor. To implement this using Converse, two major data structures are needed. First, each processor needs a "message table" containing messages that were sent, but for which no `CsmTRecv` call has been issued yet. Second, each processor needs a "thread table" containing threads that are waiting for messages, indexed by the tags that they're waiting for. Given these data structures, the send and receive functions are implemented as follows.

`CsmTSend` creates a Converse message containing the user data and the tag. It configures the message to invoke the function `CsmTHandler`. `CsmTSend` then transmits a copy of this message to the destination processor. When the message arrives, the target processor calls `CsmTHandler`, passing it a pointer to the message (which contains the user data and tag). `CsmTHandler` takes the user data and tag, and inserts it into the local message table. It then checks the thread table to see if any thread was waiting for the message. If so, that thread is awakened.

When a thread calls `CsmTRecv`, it looks in the message table, and if a matching message is already there, it is extracted and returned. If not, `CsmTRecv` obtains its own thread ID, and inserts itself into the thread table. It then puts itself to sleep. When it wakes up, it knows it has been awakened by `CsmTHandler`. It retrieves the message from the message table, and returns it.

For the message and thread tables, we used an off-the-shelf table object provided by Converse (the "message manager"). Thus, our data structures were already available to us. The thread functions were provided, as was the messaging. We had to design the format of the CSM messages (header, then tag, then user data), write the subroutines shown above, and declare and initialize the tables. In all, this took about 100 lines (2 pages) of code. This is interesting, as the message-passing model we implemented is significantly different from the underlying message-driven model of Converse.

Notice that no explicit action was needed to keep CSM from interfering with other libraries also implemented on top of Converse. CSM messages, when they arrive, trigger changes to the CSM data structures. They have no other effect. If a library system does not explicitly monitor the CSM data structures, it will not be aware that a CSM message arrived. In general, two libraries implemented on top of Converse do not notice each other's existence unless explicit action is taken to create interaction. This is in contrast to such systems as MPI, where each independent module must take explicit action (e.g., the creation of new communicator objects, etc) to avoid interfering with other modules.

2.4 Performance

Converse has been implemented on IBM SP, SGI Origin 2000, CRAY T3E, Intel Paragon (ASCI Red), Convex Exemplar, and networks of Unix/Windows workstations connected by Ethernet/ATM, Myrinet, and Quadrics Elan.

The first set of performance experiments (Figure 2.4) involves simple message passing performance. This was measured using a round trip program that sends a large number of messages back and forth between two processors. On the receiving processor, every message was delivered to a user-level handler that responded by sending a return message. Using this, the average time for one individual message send, transmission, receipt and handling was computed for the following machines:

Linux-Myrinet 2-way 1 GHz Intel Pentium III nodes running Linux connected with Myrinet interconnect. Converse runs atop MPICH-GM as well as directly on top of GM.

Origin 2000 195MHz MIPS R10000 processor connected with proprietary interconnect. Converse is implemented on top of SGI MPI.

Sun-Ethernet 60MHz Sun Sparc node connected with 10baseT Ethernet. Converse uses UDP for communication.

Linux-Fast-Ethernet 4-way 500 MHz Pentium III nodes running Linux connected with 100baseT fast Ethernet. Converse uses UDP for communication.

IBM-SP 8-way 375 MHz Power3 nodes running AIX 4.3 connected with proprietary interconnect. Converse uses MPI for communication.

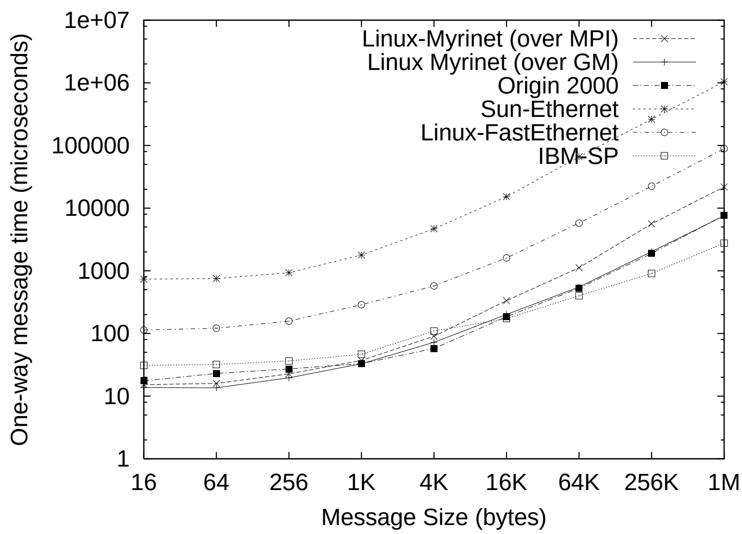


Figure 2.4: Converse message-passing performance

Overall, the performance is almost as good as that of the lowest level communication layer available to us on these machines. For example, the MPICH-GM library using Myrinet switches delivers messages of 256 bytes in 35 microseconds, whereas Converse messages need 37.14 microseconds.

In the second experiment, we incorporated queuing to investigate the overhead seen by languages that use scheduling. Each handler enqueues the received message in the scheduler's queue. The scheduler then picks a message from its queue and invokes its handler. Only languages that use the queue for scheduling objects pay this cost of scheduling. This experiment was done on Intel workstations connected by Myrinet switches to illustrate the

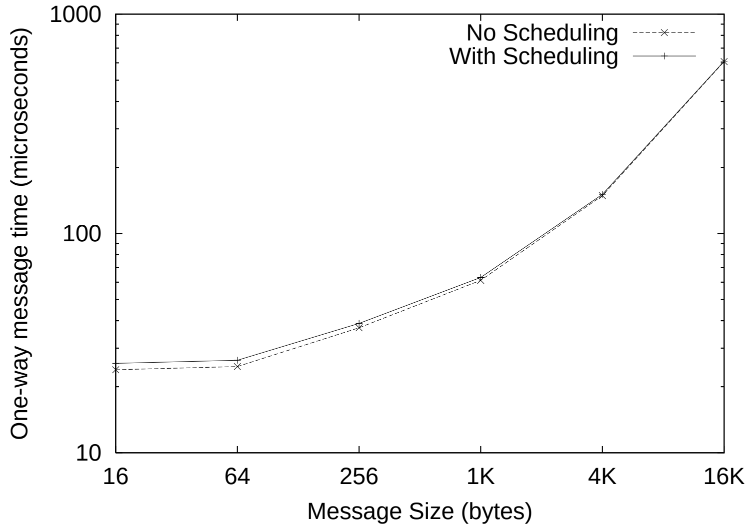


Figure 2.5: Converse scheduler overhead

magnitude of scheduling overhead (figure 2.5). The scheduling is seen to add about 1.5 to 2 microseconds.

2.5 Language Implementations

Several parallel programming languages and libraries have been implemented using Converse. The number of these languages and the ease with which we were able to implement them strongly demonstrates the utility of Converse. In this section, we describe several languages and libraries implemented on top of Converse.

We have implemented both MPI and PVM on top of Converse. This makes it possible for modules written in PVM or MPI to coexist within a single application. The MPI implementation [9] is based on MPICH [75]. The Converse port of MPICH (MICE) is very close in efficiency to the native port of MPICH on the machines we tested. In addition, MICE gains all the interoperability benefits of Converse. Our version of PVM is a from-scratch re-implementation of much of the PVM 3.3 C library. It is currently used in NAMD, a production quality molecular dynamics application (See section 2.6).

Charm [49] and Charm++ [50] were developed before Converse. They were later retargeted to Converse. Charm and Charm++ contained dynamic load balancing facilities. However, in an application with multiple language modules, load balancing should be done in the global context taking into account the entire load across all the language modules. Thus load-balancing facilities were moved into Converse, and Charm++ runtime was written to use them. We also developed a Java binding for the Charm++ constructs and entities such as remote objects with global name space, and asynchronous method invocation using Converse [42].

DP [54], a subset of High Performance FORTRAN (HPF) was implemented on top of Charm++ before the development of Converse. After Charm++ was retargeted to Converse, DP was automatically retargeted and is available for programming data parallel algorithms in a multilingual application. pC++ [15] is an object-parallel extension to C++. The method execution semantics of C++ objects is extended to include method invocation in parallel on a collection of objects. The pC++ implementation consists of a translator that converts pC++ constructs into ANSI C, and generates calls to the runtime system functions. The runtime system of pC++ offers a subset of Converse functionality. Implementing pC++ on top of Converse involved minor changes to the translator to insert calls to equivalent functionality of Converse.

Several experimental languages have been implemented on top of Converse. Import [46] is a simulation language based on MODSIM [8]. Import models a simulation system as a set of objects. The implementation relies upon the Converse messaging primitives, and its priority mechanisms. Speedups for our sample simulations have been excellent. Agents [78] is an experimental object-oriented language dedicated to exploring the idea of immutable, static networks of objects. The language supports remote method invocation, and thus, its implementation is much like the implementation of Charm++. The runtime system of the language took only a few hundred lines of code, though the compiler and optimizer were much more complex. *mdPerl* is a package for Perl, a popular scripting language. It allows writing message-driven parallel programs in Perl. The basic capability provided by mdPerl is to invoke Perl subroutines on remote processors. For typical Perl programs such as analyzing the log information of a web server, we have obtained a near linear speedup using mdPerl.

2.6 An Application of Multilingual Programming

NAMD [47] is a parallel molecular dynamics simulation program being developed in collaboration with the Theoretical Biophysics group at the University of Illinois. NAMD simulates the motions of biological molecules by repeatedly computing the forces exerted by individual atoms on one another, and integrating the motion due to these forces over time. The original version of the program, NAMD 1 [61], was built using a message-driven design. However, since it needed to use the DPMTA [67] library for long-range electrostatic force computation, it had to be implemented using PVM. This haphazard mix of SPMD and message-driven code reduced the readability of the program. NAMD 2 [38] was conceived as a rewrite of the core parallel code to increase scalability and modifiability. Our experience with NAMD 1 led us to conclude that a message-driven design was appropriate for the parallel core code, but that adding threads to the design would allow the integration loop to be expressed as a loop construct, with occasional thread suspension to wait for data. We also observed that much of the startup code was easier to write in an SPMD style. Converse supports all of these programming paradigms.

The case for multilingual programming is vividly made by our design for the integration logic that is the core of NAMD 2. The simulation space is divided into cubical regions called patches, each of which can be simulated in parallel, requiring only information from neighboring patches. Each patch has an associated object called a sequencer, which is responsible for integrating the equations of motion for the atoms owned by the patch. The sequencer contains the code that, for each time step, sends out atom positions, retrieves the forces calculated for those positions, and then computes the positions at the next time. Each Charm++ object initially creates a sequencer in its own Converse thread. First, the sequencer

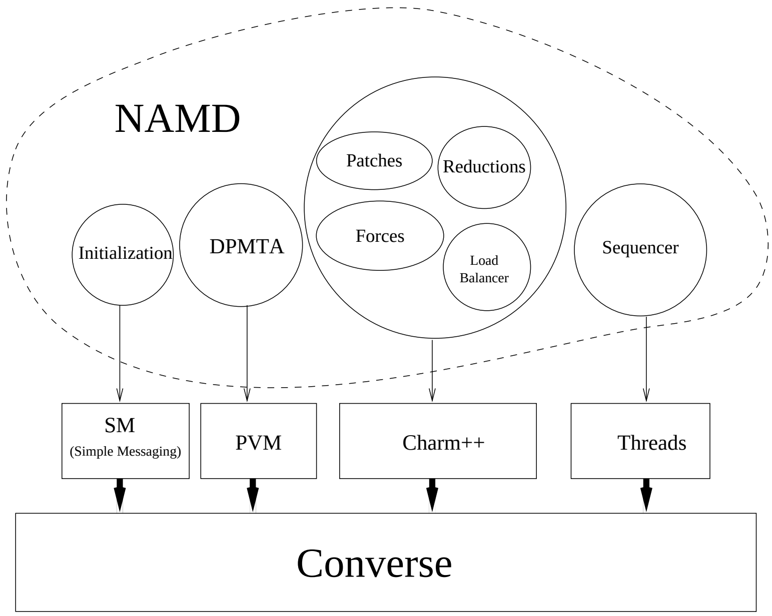


Figure 2.6: NAMD 2 Architecture

sends atom positions to compute objects, Charm++ objects that actually perform the force computations. Then the sequencer suspends its thread. The receipt of the force messages from the last compute object causes the sequencer thread to awaken, and the forces are used to update the positions for the patch. The author of a particular sequencer code writes the logic as a loop, and the only attention he must pay to the parallel nature of the code is to insert the thread suspend calls in the correct places. Thus, one can implement a new integration algorithm without having to understand details of the parallel code. NAMD 2 components and the programming paradigms used for them are shown is figure 2.6.

Increased sequential and parallel efficiency makes the program faster and much more scalable than NAMD 1, indicating that a multilingual Converse program pays little or no price for programming convenience. NAMD 2 supports all the features of NAMD 1, plus several significant features which were never part of the original program, providing evidence of the improved modifiability of its multilingual design.

2.7 Related Work

A number of runtime systems for implementing parallel languages have been described in literature. Each of these systems is geared toward portably implementing specific languages, and none of these systems have an explicit goal of supporting multilingual interoperability, which is the primary design goal of Converse. However, some of the mechanisms used by these runtime systems are similar to Converse.

2.7.1 Tulip

The Tulip effort [7] grew out of HPC++ work at Indiana University and is now deployed into many Department of Defense and Department of Energy research applications. Tulip supports remote memory copy operations such as Get and Put, remote method invocation, and efficient barrier synchronization.

Tulip uses the handler mechanism to associate computation with incoming messages. However, Tulip does not support more than one message handler in a parallel program. There is exactly one handler function (typically generated by the compiler) that handles all the messages arriving at a node. Tulip is not interrupt driven, so to avoid deadlocks and network congestion, one has to periodically call `Poll`. Typically, these poll calls are inserted by the compiler which uses Tulip as a back end. This has mandated the inclusion of a barrier primitive in Tulip, since vendor-provided barriers do not call Poll while waiting to synchronize. Apart from HPC++, no other language has been implemented using Tulip. Threads are not integral part of Tulip. (They are however, implemented in HPC++.)

2.7.2 Nexus

According to the Nexus [24] developers,

> Nexus is a portable library providing the multithreading, communication, and resource management facilities required to implement advanced languages,

libraries, and applications in heterogeneous parallel and distributed computing environments. Its interface provides multiple threads of control, dynamic processor acquisition, dynamic address space creation, a global memory model via interprocessor references, and asynchronous events. Its implementation supports multiple communication protocols and resource characterization mechanisms that allow automatic selection of optimal protocols.

Nexus is a cooperative effort between Argonne National Laboratory, the USC/Information Sciences Institute, the Aerospace Corporation, and the High Performance Computing Laboratory at Northern Illinois University. Nexus is probably the one system most closely related to Converse. Both Nexus and Converse provide the same basic facilities, active messages and threads. However, there are some significant differences in design philosophy between the two. The following list summarizes the differences between the two.

- In Nexus, if two processors share memory, they share all variables. There is no mechanism to allow an individual processor to have private data. This makes it difficult to implement functions like `malloc` efficiently. Converse makes it possible to choose whether a resource (like the malloc heap) is to be shared between processors or private. This choice can be made on a resource-by-resource basis.

- Converse threads are nonpreemptive, whereas Nexus threads are usually preemptive. Nonpreemptive threading, in combination with the processor-private data described earlier, almost completely eliminates locking and the need for thread-safe libraries. On the other hand, the preemptive threads in Nexus will be more useful in real-time or interactive systems.

- Nexus thread-blocking primitives (mutexes and condition variables) are at a higher level of abstraction than Converse thread-blocking primitives (suspend, awaken, thread identifiers). The ones in Converse are less expensive, which may be relevant to language implementers.

- Nexus has two features not present in Converse. First, Nexus can convert the data format of one machine to the data format of another, enabling the use of heterogeneous networks. Second, Nexus can dynamically add workstations to the set of processors it is using. These two features are clearly advantageous in certain cases. On the other hand, both have a cost, in that they complicate the Nexus API significantly.

- Converse includes convenience modules for dynamic load-balancing, futures, automatic parameter marshalling, tag-based message lookup, and many of the other functions a language implementer may want. Nexus is more minimalist, providing only what is strictly needed.

2.7.3 Active Messages and Variants

PM [73] is an active-message like communication mechanism on the Myrinet. Other libraries for communication over Myrinet include Myrinet API [14] (by Myricom), Fast Messages [62] and Active Messages [74] on Myrinet (Berkeley). Myrinet API is a multiuser API, but pays heavy cost for context-switching etc. and as a result the message latency is much larger than the other approaches. FM and AM-Myrinet are single user API's: Exactly one thread could be using the switch from start to completion. Also, AM requires modification to the OS kernel. PM supports multiuser channels using a user-level scheduling daemon process called S-core. It uses a modified ACK/NACK algorithm to preserve ordering among messages, and to determine when it is safe to switch context. Variable length messages of lengths up to a pagesize (4K) are allowed. For messages larger than that, one has to build packetization routines on top of PM.

Converse presents a higher level API, that is more tuned to implementing other languages than active messages and variants, which are geared toward implementing runtime systems for those languages. Converse implementations on clusters using Myrinet can use each of these APIs. Converse does not impose any restriction on the message sizes, and explicit packetization need not be implemented. In addition, Converse does not need message ordering. Therefore, the modified ACK/NACK is an overkill for our requirements.

2.7.4 Optimistic Active Messages

Optimistic Active Messages (OAM [76]) attempt to remove one major restriction imposed by Active Messages [74] on message handlers: The message handlers have to run to completion quickly. That is, if they block, it would result in a deadlock. If the message handlers execute for a long time, it would result in network congestion. Traditional RPC systems take care of this by executing each handler in a separate thread. This introduces a lot of overhead because of the context-switching time, as well as contention for access to resources by different threads. OAM takes an approach to bridge the two approaches: Every handler is expected to finish fast without blocking, and therefore executed in the scheduler's context (similar to active messages). However, when the handler asks to block, or runs for a long time, it generates a continuation and returns control to the scheduler.

Converse scheduler is re-entrant, therefore Converse can support blocking message handlers by starting another scheduler thread and by yielding to that thread if the handler wishes to block. OAM approach is to have a single scheduler thread and depend on the compiler to undo changes done to the state by message handler to re-schedule it at a later time. If one does not have compiler support, then message handlers have to be careful about changing the global state only after all the locks have been acquired. Converse threads (including the newly started scheduler thread) are non-preemptive, thus locking is often unnecessary if message handlers ensure that the global state is consistent when they block and yield to another scheduler thread.

2.7.5 Chant

Chant [29] is a parallel programming environment which extends the POSIX thread standards for lightweight thread packages. It does this by adding a new object called Chanter thread which supports the functionality for both point-to-point and remote service request communication paradigms. It uses polling-based point-to-point communication to avoid interrupts and allows for messages to be delivered directly to threads without an intermediate message buffer copy. The remote service request mechanism allows threads to register handles similar to Converse, and are executed by the respective thread when it invokes the scheduler in polled mode. Chant relies on two supporting libraries—an interprocess communication library and a lightweight threads library.

In Chant, both the sending and the receiving processor are aware of the message to be sent/received in advance. This feature is used to register the receive operation with the operating system(OS) so that the OS can copy the message directly to the proper memory location rather than perform an intermediate buffer copying operation. Pre-registration of message receives can be implemented by a programming language implementer using message-managers of Converse.

Chant threads are globally named. It identifies a thread using the combination of process group id, process rank within the group, thread rank within process. Threads within a process communicate using shared memory primitives whereas those in different processes use message passing. Chant threads are implemented over kernel-level threads, whereas Converse threads are user-level non-preemptive threads. There is no support for direct thread-to-thread communication in Converse. Rather, a more generic message-delivery mechanism is provided, which could be used to deliver messages to threads. The naming scheme for threads in Converse is local to a process. But the global naming scheme can be easily implemented on top of the generic mechanisms provided by Converse.

2.7.6 DMCS

Data Movement and Control Substrate (DMCS [19]) is an implementation of the API for communication and computation proposed by the PORTS consortium. DMCS consists of 3 subpackages:

Threads This provides a non-preemptive user-level threads package for thread creation and initialization. Two levels of priority can be assigned to a thread - low and high priority. Converse supports user-defined number of priority levels.

Communication This is implemented on top of a generic active message implementation on the SP2. Communication in DMCS is based on "Get" and "Put" operations on global pointers. In addition, asynchronous get and put operations are supported using acknowledgment variables. Global pointers, as well as other DMCS communication primitives can easily be implemented on Converse, using message handlers that get and put data from registered areas of memory upon arrival of a message.

Control This is layered on top of the threads and the communication subpackage. It consists of 2 modules: Remote Service Requests (RSR) and Load balancing. RSRs are similar to Converse invocation of remote handlers, except that the RSRs can be asked to be executed in a separate thread whose priority is specified at the time of creating an RSR. Load balancing support in DMCS is similar to Converse's "seed-based" load balancer, which allows creation of remote work on the lightly loaded processor. There is no parallel to the migration-based load balancing provided by Converse.

Chapter 3

Charisma Interface Model

Converse provides a common language runtime that allows efficient *coexistence* of software components within an application. However, software components in an application *interact* with each other by exchanging data and transferring control. An *interface model* defines the way these components interact with each other in an application. The ideal interface model for a parallel component architecture should have the following characteristics:

1. It should allow easy assembly of complete applications from reusable components. An interface description of the component along with the implicit understanding of the component's functionality should be all that is needed to use the component in an application. Thus an interface model should be able to separate the component definition from component execution.

2. It should allow individual components to be built completely independently, i.e. without the knowledge of each other's implementation or execution environment or the names of entities in other components.

3. Components should make little or no assumptions about the environment where it is used. For example, a component should not assume exclusive ownership of processors where it executes.

4. It should be possible to construct parallel components by flexibly grouping together sequential components.

5. Hierarchically composed parallel components should be able to interact with each other without being aware of each others' internal parallel structures.

6. It should not impose bottlenecks such as sequential creation, serialization etc on parallel components. In particular, it should allow parallel data exchange and control transfer among parallel components.

7. It should enable the underlying runtime system to efficiently execute the components with effective resource utilization.

8. It should be language independent and cross-platform.

In this chapter, we discuss these objectives in the context of Charm++, a message-driven object-based parallel language described in the next section. The Charm++ interface model satisfies some of the objectives listed above. In particular, Charm++ components do not assume exclusive ownership of the processor(s) where it executes, and hides the details of component construction by providing proxy interfaces. However, the Charm++ interface model, which is similar to traditional component architectures, uses a functional representation of component interfaces. Charm++ extends the object model by presenting the component functionality as methods of an object. Thus, a component interface description in Charm++ is similar to declaration of a C++ object. Components interact by explicit method calls using the interface description of each other. Extending the object model to specify component functionality has various limitations. In section 3.2, we describe these limitations. In section 3.3, we propose an interface model for Charisma that eliminates these limitations. We then describe our prototype implementation of this model.

3.1 Charm++

Charm++ [44] is a parallel message-driven object-oriented language. The basic unit of parallelism in Charm++ is a message-driven object (called a *chare*), which represents a mediumgrained computation. A Charm++ program consists of a collection of chares that interact with each other by calling each other's methods. Unlike a sequential C++ object, a chare in Charm++ has a globally unique identifier, which is used to invoke methods on it from any processor. Methods of a chare that can be invoked from objects on remote processors are called *entry methods*. Charm++ programs are written in C++ with a few library calls. In addition, each Charm++ object has a published interface, described in Charm++ interface description language (IDL). The translator for Charm++ IDL generates *proxy* interfaces (proxies) to chares. Proxies simplify remote method invocation by providing a syntax familiar to C++ programmers. The proxy object for a chare contains methods with signatures identical to entry methods of the chare, and are instantiated with a chare handle. The generated code for the method of a proxy object marshals the parameters passed to it in a single contiguous message, and sends the message to the processor that hosts the remote object associated with the proxy. The Converse scheduler on the remote processor then invokes the actual object's entry method after unmarshaling the parameters. This is illustrated in figures 3.1, 3.2, 3.3, and 3.4. The interface to a chare is described in figure 3.1. The Charm++ interface translator generates the proxy class in figure 3.2. The actual chare class is defined in figure 3.3. Figure 3.4 contains code to invoke a chare's entry methods using the proxy object.

```
chare MyChare {
  entry MyChare();
  entry void EntryMethod1(int);
  entry void EntryMethod2(void);
  entry void EntryMethod3(Message *);
};
```

Figure 3.1: Chare interface description

In addition to chares, Charm++ provides an abstraction for collections of chares, called *chare arrays*. Chare arrays have a global identifier for the collection of chares of the same class, and each individual chare is addressed within this collection with a unique index. An

```
// Generated by Charm++ interface translator
class CProxy_MyChare {
  private:
    // instance data
    CkChareID cid; // contains chare handle
  public:
    CProxy_MyChare() {
      cid = CkCreateChare(...);
    }
    void EntryMethod1 (int i) {
      MarshallMsg *m = CkCreateMessage(...);
      m->pack(i);
      CkSendMessage(cid, ..., m);
    }
    void EntryMethod2 (void) {
      MarshallMsg *m = CkCreateMessage(...);
      CkSendMessage(cid, ..., m);
    }
    void EntryMethod3 (Message *m) {
      CkSendMessage(cid, ..., m);
    }
};
```

Figure 3.2: Proxy class to `MyChare` generated from the interface description

```
class MyChare : public Chare {
  private:
    // object private data
  public:
    MyChare() { ... } // Constructor
    void EntryMethod1(int i) { ... } // a entry method
    void EntryMethod2(void) { ... } // another entry method
    void EntryMethod3(Message *m) { ... } // another entry method
};
```

Figure 3.3: Chare Definition

array element index can be, but is not limited to, a single integer, a pair, or a triplet. In general, any pattern of bits could be used to index an array element. Collective operations such as broadcast and reduction can take place over an entire array efficiently [55]. A chare array is mapped to available processors keeping load balance among processors (assuming unit load for each array element), and simplifies migration for dynamic load balancing. A

```
{
  // ....
  CProxy_MyChare * pc = new CProxy_MyChare();
  pc->EntryMethod1(345);
  pc->EntryMethod2();
  pc->EntryMethod3(new Message());
  // ....
}
```

Figure 3.4: Invoking a chare's method

chare array's interface is described in terms of the interface to the individual chare array element. The Charm++ interface translator generates proxy class definitions similar to that of ordinary chares, so that methods on individual array elements can be remotely invoked. In addition, the translator generates proxy methods that invoke corresponding methods on all the elements of an array (similar to broadcasting a message.) Figure 3.5 shows the interface definition of a one dimensional chare array. Its implementation is shown in figure 3.6. Figure 3.7 shows the two ways of invoking methods on arrays. `EntryMethod1` is invoked on an individual array element by explicitly naming it by its index, and `EntryMethod2` is invoked on all elements of the chare array.

```
array [1D] MyArray {
  entry MyArray();
  entry void EntryMethod1(int);
  entry void EntryMethod2(void);
};
```

Figure 3.5: Chare array interface description

A special type of chare array is an array indexed by processor number. This is called a *chare group*.[1] A chare group encapsulates chares exactly equal in number to the available processors, and each processor contains exactly one *branch* chare of the group. Elements of a chare group are nonmigratable, thus enabling several optimizations in chare group implementation. Chare groups can be used for low-level system tasks such as implementing

[1] This was called a Branch Office Chare, or Branched Chare in earlier versions of Charm++.

```
class MyArray : public ArrayElement1D {
  private:
    // object private data
  public:
    MyArray() { ... } // Constructor
    void EntryMethod1(int i) { ... } // a entry method
    void EntryMethod2(void) { ... } // another entry method
};
```

Figure 3.6: Chare array element Definition

```
{
  // ....
  int numElements = 25; // number of array elements

  // create the chare array
  CkArrayID aid = CProxy_MyArray::ckNew(numElements);

  // construct a proxy to the chare array
  CProxy_MyArray pa(aid);

  // invoke EntryMethod1 on array element 20
  pa[20].EntryMethod1(768);

  // invoke EntryMethod2 on all elements of the array
  pa.EntryMethod2();
  // ....
}
```

Figure 3.7: Invoking chare array's methods

collective communications, distributed tables [69] etc. In particular, they serve as parallel pathways for data exchange between Charm++ modules [43] as illustrated in section 3.6.

Charm++ can be considered to be an object-based counterpart of Converse, where handler functions are replaced by entry methods that execute in the context of an object. The object context provides encapsulation for entry methods, and maintains state across method invocation unlike Converse handlers. The objects also serve as a useful description of work units to the runtime system, especially the dynamic load balancing subsystem of Converse, since work can easily defined as computations performed by the entry methods of an object, and migration of work can be carried out at the object level. A Charm++ object does not assume ownership of the processor on which it resides, thus allowing the runtime system to concurrently interleave its execution with other components on the same processor. Also, Charm++ allows one to construct parallel components from sequential components with constructs such as chare arrays. Clients of the Charm++ components invoke component services through proxy objects, thus hiding the method of construction of the parallel component. Because of these qualities of Charm++ objects, we have chosen to build Charisma on top of Charm++ rather than directly on top of Converse. The existing interface model of Charm++ however, is an extension of the object model, and causes a number of limitations that prohibit it from possessing all the properties of the ideal interface model that we described earlier.

3.2 Limitations of Interfaces based on Object models

At its core, the Charm++ interface model is a sequential extension of the C++ interface model that allows invoking methods on remote objects. A chare is essentially a sequential component that may reside on a remote processor. Note that a chare may be implemented using more than one sequential C++ objects, but this is hidden from the user of the chare. The chare interface defines the access point for the chare and not its constituent sequential objects. A parallel component would typically consist of a number of such sequential components. A chare array presents an appropriate abstraction to implement a parallel component, since it provides a level of encapsulation over a collection of sequential components. However, the interface to chare arrays is presented in terms of interfaces to its constituent chares. Therefore, communication between such components would take place by components invoking methods on individual chares in other components, thus making them dependent on each others' internal parallel structures. One could hide the internal structure of components by providing wrapper objects for interfaces, as described next.

Solution 1 [Sequential Wrapper] : A straightforward extension of object-based interface models, such as the Charm++ interface description, for parallel components is to provide a sequential component wrapper for the parallel component, where functionality of a parallel component is presented as a sequential method invocation (Figure 3.8). This imposes serialization bottleneck on the component. For example, a parallel CSE application that interfaces with a linear system solver will have to serialize its data structures before invoking the solver. This is clearly not feasible for most large linear systems representations that need gigabytes of memory.

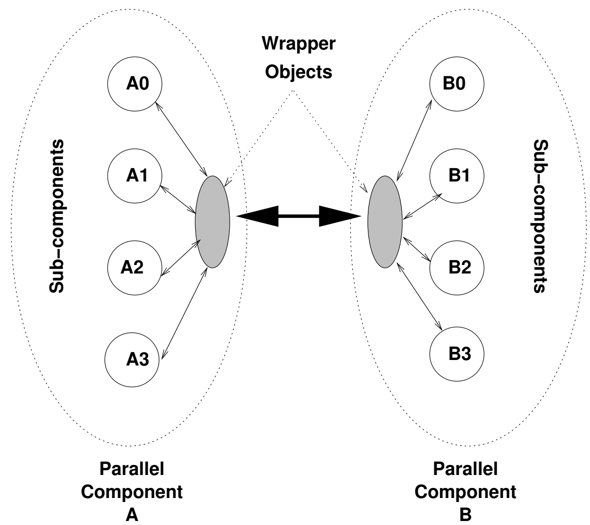


Figure 3.8: A sequential wrapper for parallel components

Solution 2 [Isomorphic Wrapper] : Another extension of the object-based interface models is to treat each parallel component as a collection of sequential components. In this model, the interaction between two parallel components takes place by having the corresponding sequential components invoke methods on each other (Figure 3.9). Thus, the interaction between components is defined in terms of interactions between sub-components of each component. While this model removes the serialization bottleneck, it imposes rigid restrictions on the structure of parallel components. For example, a parallel finite element solver will have to partition its mesh boundary into the same number of pieces as the neighboring block-structured CFD solver, while making sure that the corresponding pieces contain adjacent nodes.

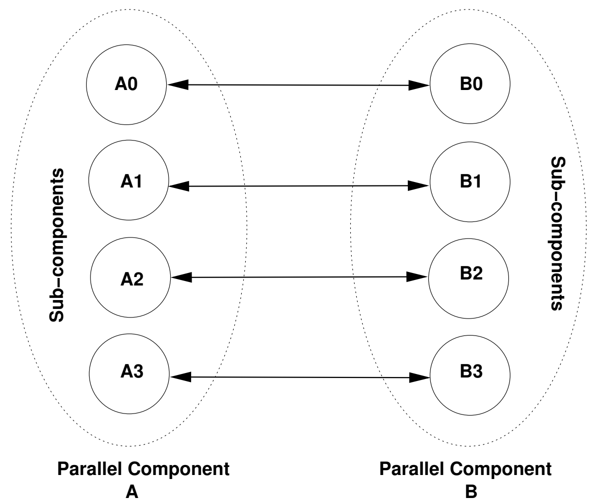


Figure 3.9: Rigid structure imposed on interaction between parallel components

Solution 3 [Processor-based Parallel Wrapper] : In order to avoid the serialization bottleneck in data and control transfer among components such as in figure 3.8 while not imposing rigidity in component interaction such as in figure 3.9, one can provide parallel wrappers around components as shown in figure 3.10. The parallel wrapper consists of a group of objects (e.g. `Pa0` and `Pa1` in figure 3.10), which are mapped to processors so that there is exactly one object per wrapper per processor. Objects belonging to a component always communicate with their local representative object of the parallel wrapper. They wait for data to be delivered to them by the local wrapper object, and upon computation, deliver the results also to the local wrapper object. Communication between components are thus mediated by the parallel wrapper objects of those components. The local parallel wrapper objects of two interacting components are bound together. If each component contains such parallel wrapper objects, the components themselves do not need to know about the connection topology of the peer components.

We have implemented this scheme of component interaction in NAMD (section 2.6) for mediating interaction between short-range electrostatics module based on Charm++ and long-range electrostatics module DPMTA. The short-range electrostatics module is implemented as a dynamically load balanced chare array in Charm++, where each array element represents a cubical portion of space (called a *Patch*) containing atoms. DPMTA (written in PVM) uses tree-structure and partitions its computations into pieces mapped one-to-one on processors. Since both these components use different partitioning schemes, atoms need to be re-partitioned according to their positions in space each time control transfers between these electrostatics modules. A parallel wrapper is implemented for the short-range electrostatics module using chare groups in Charm++. The chare array elements (patches) of the short-range electrostatics module deposit their atoms' positions with the local representative of the wrapper chare group (called *PatchManager*), which combines data from local patches and delivers it to the re-partitioning code in DPMTA via a library function call. When the parallel wrapper receives results of long-range electrostatics computations from DPMTA, it then re-partitions the received atoms and delivers them back to the local patches. While this method eliminates the serialization bottleneck, it results in a lot of "glue" code in the form of parallel wrappers. Also, since this method of component interaction is still based on method-calls (between wrapper objects), it makes the data exchange and control transfer between components hard-wired within the components, and in doing so, this method does not provide *control points* for flexible application composition and for effective resource management by the runtime system.

Lack of a control point at data exchange leads to reduced reusability of components. For example, suppose a physical system simulation component interacts with a sparse linear system solver component, and the data exchange between them is modeled as sending messages or as parameters to the method call. In that case, the simulation component needs to transform its matrices to the storage format accepted by the solver, prior to calling the solver methods. This transformation code is part of the simulation component. Suppose, a better solver becomes available, but it uses a different storage format for sparse matrices. The simulation component code needs to be changed to transform its matrices to the new format required by the solver. If the interface model provided a control point at data exchange, one

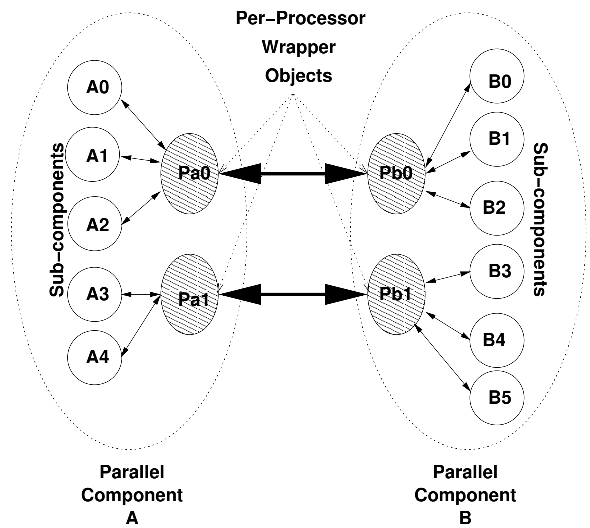


Figure 3.10: Processor-based parallel wrappers promote efficient interaction among parallel components

can use the simulation component without change, while inserting a transformer component in between the simulation and the new solver.

Lack of a control point for the runtime system at control transfer prevents the runtime system from effective resource utilization. For example, with blocking method invocation semantics of control transfer, the runtime system cannot schedule other useful computations belonging to a parallel component while it is waiting for results from remote method invocations. Asynchronous remote method invocation provides a control-point for the runtime system at control-transfer. It allows the runtime system to be flexible in scheduling other computations for maximizing resource utilization. However, when we extend functional interface representations to use asynchronous remote method invocations, the resultant components have to supply continuations explicitly to their connected components. These are referred to as "compositional callbacks".

Solution 4 [Compositional Callbacks] : When a component (caller) invokes services from another component (callee) using asynchronous remote method invocation, it has to supply the callee with its own unique ID, and the callee has to know which method of the caller to call to deposit the results. This is illustrated with a simple client-server transaction in figure 3.11. Note that the client has to know the server's interface (in particular the name of the method `service`). Also, the server has to know the client's interface. In addition, both have to decide upon and hardcode the types of data they deposit or accept.

```
Client::invokeService() {
  ServiceMessage *m = new ServiceMessage();
  // ...
  m->myID = thishandle;
  ProxyServer ps(serverID);
  ps.service(m);
}

Server::service(ServiceMessage *m) {
  // ... perform service
  ResultMessage *rm = new ResultMessage();
  // ... construct proxy to the client
  ProxyClient pc(m->myID);
  pc.deposit(rm);
}

Client::deposit(ResultMessage *m) {
  // ...
}
```

Figure 3.11: Asynchronous remote service invocation with return results

The mechanism of component interaction used in figure 3.11 is referred to as the "compositional callback" mechanism. It is useful in developing an application with pre-written server libraries. The server does not need to know the client's interface. The client must be a subclass of a generic client of the server. Compositional callback mechanism is equivalent to building an object communication graph (object network) at run-time. Such dynamic object network misses out on certain optimizations that can be performed on a static object network [77]. For example, if the runtime system were involved in establishing connections between communicating objects, it would place these objects closer together (typically on the same processor).

Another problem associated with the callback mechanism is that it leads to proliferation of interfaces, increasing programming complexity. For example, suppose a class called `Compute` needs to perform asynchronous reductions using a system component called `ReductionManager` and also participates in a gather-scatter collective operation using a system component called `GatherScatter`. It will act as a client of these system services. For `ReductionManager` and `GatherScatter` to recognize `Compute` as their client, the `Compute` class will have to implement separate interfaces that are recognized by `ReductionManager` and `GatherScatter` respectively. This is shown in figure 3.12. Thus, for each service that a component provides, this would result in two interfaces: one for the service, and another for the client of that service. If a component avails of multiple services, it will have to implement all the client interfaces for those services. In addition to the proliferation of interfaces, this model makes it difficult to have different concurrent instances of service invocations for the same service. For example, in Figure 3.12, if the class `Compute` needs to use the `Reduction` service at different places within the code, it needs to explicitly encode the continuation in its state before it invokes the reduction service. When the reduction results arrive via the `reductionResults` method, it has to explicitly deliver the results to the stored continuation.

```
class ReductionClient {
  virtual void reductionResults(ReductionData *msg) = 0;
}

class GatherScatterClient {
  virtual void gsResults(GSData *msg) = 0;
}

class Compute : public ReductionClient, public GatherScatterClient
{
    // ....
    void reductionResults(ReductionData *msg) { ... }
    void gsResults(GSData *msg) { ... }
}
```

Figure 3.12: Proliferation of interfaces

3.3 Charisma Interface Model

As outlined before, an ideal interface model would ensure that each component has no knowledge of the names of entities in other components to which it connects. Also, these components should not know about the methodology or programming paradigm used to construct other components. For maximum independence in building components, which leads to reusability of components, the interface should be a contract between a component and the component framework, rather than between two components.

Our interface model requires components to specify the data they use and produce. It takes the connection specification (glue) between components out of the component code. The connection specification code may even be written as a script that is translated and compiled into the application. This approach provides the application composer and the runtime system with a control-point to maximize reuse of components. Asynchronous remote method invocation semantics with message-driven execution (see section 2.2) is assumed for dispatching produced data to the component that uses them, thus supplying the runtime system with a control-point for effective resource utilization.

The interface of a component consists of two parts: a set of input ports, and a set of output ports. A component publishes the data it produces on its output ports. These data become visible (when scheduled by the runtime system) to the connected component's input port. The connection between an input port and an output port are specified outside of the object's code, using the Charisma interface language. Each object can be thought of as having its own scheduler which schedules method invocations based on the availability of data on any of its input ports, and possibly emitting data at its output ports. For message-driven object-based languages, such as Charm++, input ports of components have methods bound to them (see section 3.5), so that when data become available on the input port, a component method is enabled.[2] For languages such as MPI, an input port may be treated

[2]Enabling an object method is different from executing it. Execution occurs under the control of a scheduler, whereas a method is enabled upon availability of its input data. This separation of execution and

as a "pseudo-processor" that the component receives messages from (see section 5.4). Unlike other interface models, Charisma does not dictate the language binding for the port, leaving it to the language's runtime system. This makes the transition to Charisma easier for the component developer. It is up to the language developers to ensure that their implementation of ports remains interoperable with other languages.

The following examples illustrate key concepts behind the Charisma interface model. We have chosen the Charm++ implementation to illustrate the interface model. Also, we have used the Charisma interface description language (IDL) for Charm++. As described in section 3.5, one can provide the interface description to the runtime system using a C++ API instead of Charisma IDL. The Charisma interface language translator generates C++ code that invokes this API.

A simple producer-consumer application using Charisma interface model is shown in figure 3.13. Note that both the producer and the consumer know nothing about each other's methods. Yet, with very simple scripting glue, they can be combined into a single program. Thus we achieve the separation of application execution from application definition. Individual component codes can be developed independently, because they do not specify application execution. They merely specify their definitions. For example, the producer component does not ask the runtime system to deliver the data to the consumer component. It merely tells the runtime system that the data is available for delivery. In this model, the producer does not have to name the peer component, and thus can be developed independently, and is therefore more reusable. The connection script outside of the component, along with the message-driven execution semantics specifies the actual execution.

In figure 3.13, the data types of the connected ports of producer and consumer match exactly. In general, wherever transformations between data types is possible, the system will implicitly apply such transformation. For example, if the type of `producer::data` is `double`,

enabling objects methods is crucial to our interface model, as it provides a control-point for the Charisma runtime system to effectively utilize computational resources.

```
class producer {
  in Start(void);
  in ProduceNext(void);
  out PutData(int);
};
```

(a) Producer Interface

```
producer::Start(void) {
  data = 0;
  PutData.emit(0);
}
producer::ProduceNext(void) {
  data++;
  PutData.emit(data);
}
```

(b) Producer Implementation

```
class consumer {
  in GetData(int);
  out NeedNext(void);
};
```

(c) Consumer Interface

```
consumer::GetData(int d) {
  // do something with d
  NeedNext.emit();
}
```

(d) Consumer Implementation

```
producer p;
consumer c;

connect p.PutData to c.GetData;
connect c.NeedNext to p.ProduceNext;
connect system.Start to p.Start;
```

(e) Application Script

Figure 3.13: A Producer-Consumer Application

and the type of `consumer::data` is `int`, the system will automatically apply the required transformations, and will still allow them to be connected. However, if `producer::data` is `Rational` and `consumer::data` is `Complex`, then system will not allow the requested connection. Thus, the application composer can insert a transformer object (see figure 3.14) between the producer and the consumer, without having to rewrite any portions of producer or consumer.

A performance improvement hint "inline" (figure 3.14a) can be interpreted by the translator for the scripting language to execute the method associated with the input port immediately instead of putting it off for scheduling it later. This hint also guides the runtime system to place the transformer object on the same processor as the object that connects to its input.

```
class transformer {
  inline in input(Rational);
  out output(Complex);
};
```

(a) Transformer Interface

```
transformer::input(Rational d) {
  Complex c;
  c.re = d.num/d.den; c.im = 0;
  output.emit(c);
}
```

(b) Transformer Implementation

```
producer p;
consumer c;
transformer t;

connect p.PutData to t.input;
connect t.output to c.GetData;
connect c.NeedNext to p.ProduceNext;
connect system.Start to p.Start;
```

(c) Application Script

Figure 3.14: Transformer Component

The real power of this interface model comes from being able to define collections of such objects. For example, one could connect individual sub-image smoother components as a 2-D array (figure 3.15) to compose a parallel image smoother component (see figure 3.16).

The composite `ImageSmoother` component specifies connections for all the `InBorder` and `OutBorder` ports of its `SubImage` constituents. Note that by specifying connections for all its components' ports, and providing unconnected input and output ports with the same names and types as `SubImage`, `ImageSmoother` becomes *portwise-isomorphic* to `SubImage` and can be substituted for `SubImage` in any application. Code for `ImageSmoother` methods `InBorder` and `InSurface` is not shown here for lack of space. `InBorder` splits the input pixels into subarrays and emits them on `OutSurface` ports. `InSurface` buffers the pixels until all the border pixels are handed over to it from a particular direction. It then combines all the pixels into a single array, and emits them onto corresponding `OutBorder` port.

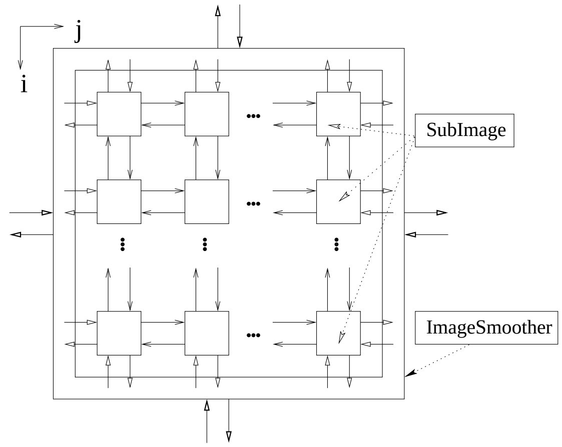


Figure 3.15: Construction of a 2-D array component from elements

Also, note that the `ImageSmoother` component can configure itself with parameters `N` and `M`. `N` and `M` determine the number of rows and columns in a 2-D array of `SubImage`s. They can be treated as attributes of the class `ImageSmoother`, which can be set through a script.

In this example, the number of ports in `SubImage` was fixed. User of this component was expected to feed and receive one array in each of the four directions. The above example demonstrates that one can substitute a parallel component in place of the sequential component by writing appropriate glue code to keep the original interface with fixed number of ports. One can design the interfaces so that the number of ports of a component can be configured by the application composer, as the next example shows. The configurable parameter supplied by the application script to the component acts as a guideline to the component for deciding its internal parallel structure.

Consider the problem of interfacing Fluids and Solids modules in a coupled simulation [63]. Each of the Fluids and Solids component is implemented as a parallel object. (Constituents of these modules, namely `FluidsChunk` and `SolidsChunk`, are not shown here for the sake of brevity.) A fluid-solid interface component `FSInter` specific to the application-domain is used to connect an arbitrary number of Fluids chunks to any number of Solids chunks by carrying out the appropriate interpolations. The core interface description of this

```
class SubImage {
  in[4] InBorder(Pixels *);
  out[4] OutBorder(Pixels *);
}
```

(a) SubImage Component Interface

```
enum {EAST=0, WEST=1, NORTH=2, SOUTH=3};
class ImageSmoother <int N, int M> {
  in[4] InBorder(Pixels *);
  out[4] OutBorder(Pixels *);
  in[2*N+2*M] InSurface(Pixels *);
  out[2*N+2*M] OutSurface(Pixels *);

  SubImage si[N][M];
  // Make the east and west elements connections to surface
  for(int i=0; i<N; i++) {
    connect si[i][0].InBorder[WEST] to this.OutSurface[i];
    connect si[i][0].OutBorder[WEST] to this.InSurface[i];
    connect si[i][M-1].InBorder[EAST] to this.OutSurface[i+N];
    connect si[i][M-1].OutBorder[EAST] to this.InSurface[i+N];
  }
  // Similarly connect north and south border elements to surface
  for(int j=0; j<M; i++) {
    // ...
  }
  // Now, make internal elements connect to each other
  for(int i=1; i<=(N-1); i++) {
    for(int j=1; j<=(M-1); j++) {
      connect si[i][j].InBorder[0] to si[i-1][j].OutBorder[2];
      // ...
    }
  }
}
```

(b) ImageSmoother Component Interface

Figure 3.16: A Parallel image smoother construction from sub-image smoother components

situation is shown in figure 3.17. Figure 3.17(a) shows the interface of the parallel `Fluids` component, and figure 3.17(b) shows the interface of the parallel `Solids` component. Each of these components have a configurable parameter that determines the number of ports presented by each component. The `FSInter` component (Figure 3.17(c)) presents two sets of ports, one for connecting to `Fluids` and the other to `Solids`. Figure 3.17(d) shows the skeleton application code that configures `Fluids` and `Solids` to have 32 and 64 input and output ports, respectively. The two sets of ports in `FSInter` are then configured accordingly to appropriately connect the `Fluids` component with 32 ports and `Solids` component with 64 ports. The `Fluids` and `Solids` components interacts solely with `FSInter`, which carries out the interpolation of data it receives from either either component, and passes it on to the other component at each timestep. Note that the configurable parameters only dictate the number of ports a component contains, and does not enforce the parallel structure on the component. For example, when configured to have 32 input and output ports, the `Fluids` component may internally partition its grid into 32×32 chunks, with its border chunks connected to `FSInter` with 32 ports as shown in figure 3.18. Alternatively, it may partition its grid in chunks equal in number to available processors, and can still split the grid boundary data into 32 parts that are published on 32 ports (Figure 3.19).

3.4 Dynamic Component Creation

Though most application compositions can be specified at compile (or link) time, for some applications (such as symbolic computations, branch-and-bound) it is necessary to dynamically specify connections, or to dynamically create components. Another situation where dynamic component creation is suitable commonly occurs in particle-based scientific applications. Particle sets are partitioned at runtime based on their spatial coordinates using algorithms such as recursive bisection. If each partition is represented by a sequential component that models a 3-D box containing particles, the component representing the entire particle set needs to be constructed from the 3-D boxes at runtime. This dynamic construction for a particle set is illustrated in figure 3.20 (shown in 2-D for simplicity).

For such applications, one has to use the component-connection API explicitly. This API

```
class Fluids<int N> {
  in[N] Input(FluidInput);
  out[N] Output(FluidOutput);
};
```

(a) Fluids Component Interface

```
class Solids<int M> {
  in[M] Input(SolidInput);
  out[M] Output(SolidOutput);
};
```

(b) Solids Component Interface

```
class FSInter<int F, int S> {
  in[F] FInput(FluidOutput);
  out[F] FOutput(FluidInput);
  in[S] SInput(SolidOutput);
  out[S] SOutput(SolidInput);
};
```

(c) FSInter Component Interface

```
Fluids f<32>;
Solids s<64>;
FSInter fs<32,64>;

for(int i=0;i<32;i++){
  connect f.Output[i] to fs.FInput[i];
  connect fs.FOutput[i] to f.Input[i];
}
for(int i=0;i<64;i++){
  connect s.Output[i] to fs.SInput[i];
  connect fs.SOutput[i] to s.Input[i];
}
```

(d) Application Composition

Figure 3.17: Fluids-Solids interface in a coupled simulation

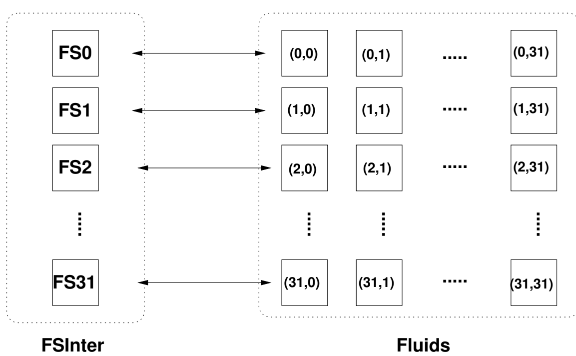


Figure 3.18: Fluids grid partitioned into 32×32 chunks

is available as an interface between the creator of the component and a "system" component. Thus, creator's output port connects to the system's input port and emits the class type to be created, and also specifies connection information. An example in figure 3.21 constructs a tree of objects dynamically.

In this example, the root node creates two children by emitting two `CreateDynamic`

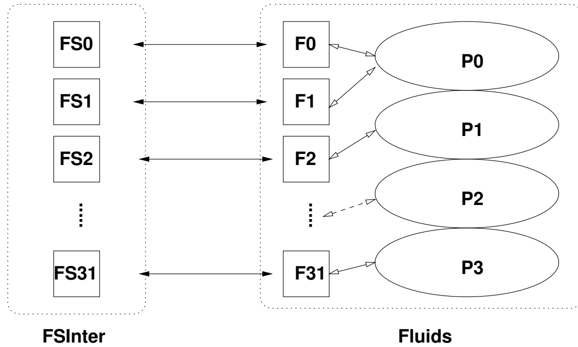


Figure 3.19: Fluids grid partitioned into 4 processors

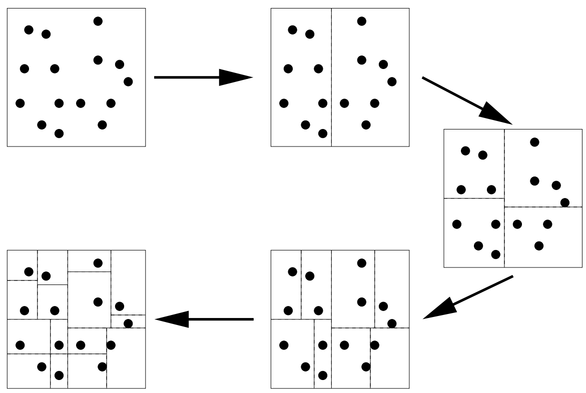


Figure 3.20: Construction of a Particle-Set component from Boxes dynamically using recursive bisection.

requests on its `Create` port, which is connected to the system's `CreateObject` input port. Before emitting `CreateDynamic`, it fills it with information about the object to be created. This includes the object's type, and its connections. The `Port` call takes two parameters, object ID and port name, and returns a class-local port ID. Note that the ID for the object that is to be created is unknown to the creator. For this purpose, a special value, 0, is recognized by Charisma API as the ID of the object to be created.

3.5 Prototype Implementation

We have a working prototype of our interface model implemented over Converse and Charm++. Currently, components can be written in Charm++ and Adaptive MPI, and can execute on all the machines that Converse supports. We describe this implementation with example

```
class node {
  in Start(void);
  out Child1(void);
  out Child2(void);
  out Create(CreateDynamic);
};
```

(a) Node Interface

```
node root;

connect root.Create to
        system.CreateObject;

connect system.Start to
        root.Start;
```

(b) Node Connections

```
node::Start(void)
{
  CreateDynamic c1;
  c1.classid = this.classid;
  c1.connect(Port(0, "Create"),
             Port(system, "CreateObject"));
  c1.connect(Port(this, "Child1"),
             Port(0, "Start"));
  Create.emit(c1);
  CreateDynamic c2;
  c2.classid = this.classid;
  c2.connect(Port(0, "Create"),
             Port(system, "CreateObject"));
  c2.connect(Port(this, "Child2"),
             Port(0, "Start"));
  Create.emit(c2);
  // ...
}
```

(c) Node Method

Figure 3.21: Dynamic Network Creation

components written using Charm++.

There are three important parts of our implementation: Component registration, Component creation, and Component connections. Before we describe each of these in detail, we will discuss what a Component is in terms of Charm++.

A Charisma component written using Charm++ is a Charm++ chare array. An array element in Charm++ is the most general abstraction for a message-driven object, since it can belong to arbitrary collections (arrays), can migrate in order to balance load, and has a globally unique name and index given to it at runtime. The index type used for this array element can be any user-defined type up to a maximum size (configured at compile-time). This is critical for writing reusable components, since a component may be used as a subcomponent of a component that could have any type of collection. For example, a 3-D Jacobi component which performs neighborhood averaging may be used in a component that uses 3-D block-decomposition, 2-D pencil-like decomposition, or as leaves (indexed with a bit-vector) of an oct-tree in adaptive mesh-refinement applications.

The port abstraction of Charisma does not impose any restrictions on the syntax for using ports. Instead, the syntax for the ports is dependent on language bindings. This makes it possible for language-runtime developers to provide abstractions that are natural for the component developer in that language. For example, ports implemented as templated classes may be natural to a Charm++ component developer, but alien to an MPI-FORTRAN90 programmer. One choice for the port abstraction for AMPI programmer would be to use cross-communicators (section 5.4), where publishing data on an output port would be equivalent to sending a message using that communicator.

For Charm++ implementation of Charisma, we have implemented output ports as sub-classes of `CkCallback` class from Charm++. `CkCallback` mechanism allows runtime binding of various types of method invocations. An input port in Charisma corresponds to an entry method of a Charm++ array element. Binding an output port `OutP` of a component `CompA` to an input port `InP` of component `CompB` involves getting the unique ID of `CompB` and binding the callback part of `OutP` to invoke method `InP` on `CompB`. The `emit` method of `OutP` is a wrapper to asynchronously fire a callback.

Runtime systems of different languages on top of Converse may provide different kinds of callbacks. For example, a multi-threaded language will associate a callback with its input ports that awakens a suspended thread. In message-passing languages, a callback object for an input port would make it appear as if the data to be accepted on that port comes as a message with a specific tag or from a specific processor. Even in Charm++, a component may use the Charisma API instead of the translator generated code to specify a different kind of callback for its input port. This leads to maximum flexibility in building components.

The interface language translator reads the definition of the component from an appropriately named file. For example, a component `Jacobi` would be defined in file `Jacobi.co`. It then generates a definition of class `CoJacobi` in file `Jacobi.co.h`. The `CoJacobi` class is used as a base class of the `Jacobi` component. The `CoJacobi` class contains output port definitions (with names and types specified in the `.co` file), and two constructors. One of the constructors is invoked by the initialization code of the application to ask the component to register itself and its ports with the runtime system without actually creating the component. The other constructor is invoked when the component is actually created. This constructor reads the component-specific part of connection information from the system and binds its ports.

The application script specifies the top level components to be created, and specifies connections between these top-level components using the same scripting language. However, an application script is translated using a special mode of the translator which treats it as an application rather than as a component. This involves producing a main program which actually starts the execution.

This generated main program starts with registering the top-level components with the Charisma runtime. These top-level components may consist of sub-components themselves, and the registration call for these components calls the registration method of the subcomponents and so on recursively. Thus the components themselves form a hierarchy. For example, if a simulation application consists of components `Rocflo`, `Rocsol`, and `Rocface`, calls to `Rocflo::Register`, `Rocsol::Register`, and `Rocface::Register` are generated. These components in turn register their subcomponents by calling `FloChunk::Register`, `SolChunk::Register`, and `FaceChunk::Register` for all the chunks it contains. Components that do not contain any further subcomponents register their ports. This hierarchical registration also aids in assigning globally unique names for components and their ports. For example, the main script of the producer-consumer application registers producer component with name `p` and consumer component with name `c`, whereas the producer registers its ports such as `PutData` without knowing the name it has been assigned by the main script. The Charisma runtime maintains the hierarchy of components and ports. For example, the `PutData` port of `p` is referred to as `p.PutData` outside of `p`, but simply `PutData` inside `p`. Thus, in the main application script outside `p`, a connection such as `p.PutData` to `c.GetData` can be made.

At the end of the registration phase, the Charisma runtime knows how many components exist in an application. The next phase is the connection phase. In the connection phase, the top level connections are specified to the runtime, and the generated `connect` methods on all the components are called, which in turn specify their own connections to the runtime. The runtime makes use of the hierarchical structure of the registration components to store connections. At the end of the connection specification phase, Charisma runtime knows about the connectivity of the components in an application. This is stored in the form of a graph, and upon the end of connection specification phase, the runtime passes this graph to a graph-partitioner (we use the freely available METIS [51] partitioning tool.) The graph partitioner produces an assignment of components to different processors. This assignment is in the form of a table that specifies a processor number for each component.

For the purpose of partitioning all the components are currently thought to be of equal computational load, and all the connections of equal weight. In future, we would associate some measure of load and communication parameters to each of the components and connections that can be taken into account while doing graph partitioning. Processor-assignment for components produced by Charisma is used only as a guideline for creating components. The dynamic load balancing strategies of Charisma take into account the connectivity of components in addition to the actual measurement of computational loads of individual components to balance the load periodically by migrating components at runtime.

Components are then created as Charm++ arrays. Even singleton components such as `producer` in the producer-consumer application are created as an array (of one element). This gives us uniformity to address all components, and simplifies the implementation. This is not a restriction imposed by Charisma. Indeed, Charisma promotes cross-paradigm components by leaving it to the component developers to use whatever abstraction they find suitable.

The components (array elements) themselves are created with a special parameter that enables them to look up their connections upon creation in the distributed database of connections maintained by the runtime system. Thus the first task each component performs upon creation is to inquire with the runtime system of the other end-points of its output ports. Between issuing the creation commands to components, and actual creation, the Charisma runtime makes sure that each processor has the connection graph, and thus can satisfy the connection-information request of the newly created component locally. This is currently achieved by storing the connection database in a readonly message that gets distributed to all the processors before any objects are actually created. In future, we would use the port ownership information provided to the runtime system by the components, and distribute the connection graph in the form of a distributed table with local caching instead of replicating the entire graph on all processors.

The output ports of each component contain a callback structure that gets initialized with the array identifier, index of a component within that array, and the entry method index corresponding to the input port of that component. The `emit` method on the output port then becomes a wrapper around the `send` method of the callback object.

3.6 Port Examples

By allowing the components to provide specialized implementations of input ports, Charisma provides efficient ways to integrate legacy codes, as well as to deal with advanced irregular applications, even those using the *multi-partitioning* approach (see chapter 5). In this section, we describe how a language runtime may enable its applications to be "componentized" by providing efficient port implementations.

Consider an example of a Finite Element Method Framework [10], which uses a sequential mesh partitioner (such as METIS [51]) to re-partition the finite element mesh distributed across a parallel machine. Computation on the finite element mesh are carried out by a component `FEMComp`, implemented as a chare array in Charm++, where each chare contains a chunk of the entire mesh. Each element of this chare array has one output port, where the element emits its portion of the mesh connectivity information. This connectivity information needs to be combined and when connectivity of the entire mesh is available, it needs to be supplied to the sequential METIS library routine, which re-partitions the mesh. The new partitioning needs to be conveyed to the `FEMComp` via its input ports.

As a most simplistic implementation of the re-partitioner component, one may write a sequential object wrapper `SeqPartitioner` around the METIS partitioner (Figure 3.22). `SeqPartitioner` can be implemented as a chare in Charm++. For the FEM application running with P partitions, `SeqPartitioner` has P input ports and P output ports. Each of `SeqPartitioner`'s input ports are connected to the corresponding output ports of `FEMComp`, and its output ports are connected to the corresponding input ports of `FEMComp`. When the mesh connectivity of a partition is published on the output port by any element of the `FEMComp` chare array, the connected input port simply forwards that information in the form of a Charm++ message to the `SeqPartitioner` chare. The `SeqPartitioner` object buffers incoming messages until it receives all P messages. After all the expected messages have been received, it combines the mesh connectivity information into a single array and supplies that to the METIS library function to partition the mesh. Upon partitioning, the `SeqPartitioner` splits the returned partitioning into FEM mesh chunks, and publishes a chunk on the corresponding output port, which reaches the appropriate constituent chare in `FEMComp` component through the connected input ports.

```
class SeqPartitioner : public Chare {
    Port *inputs;
    Port *outputs;
    int np, nrcvd;
    Partition** parts;
  public:
    SeqPartitioner(char *name, int p) {
      np = p;
      inputs = new Port[p];
      outputs = new Port[p];
      parts = new Partition*[p];
      // initialize input and output ports
      for(int i=0;i<p;i++) {
        inputs[i].init(i, thishandle,
                       GetMethodID("RecvPartition", "SeqPartitioner"));
        outputs[i].init(name, "outputs", i);
      }
      nrcvd = 0;
    }
    void RecvPartition(int p, Partition *part) {
      // buffer incoming message
      parts[p] = part;
      nrcvd++;
      // have all messages been received ?
      if(nrcvd==np) {
        Partition *comb = CombinePartitions(np, parts);
        // call METIS partitioner
        METIS_PartGraphRecursive(...);
        parts = SplitPartitions(comb, np, ..);
        for(int i = 0; i<np; i++)
          outputs[i].emit(parts[i]);
        nrcvd = 0;
      }
    }
}
```

Figure 3.22: Sequential partitioner wrapper for METIS

Note that the number of messages that are received by the `SeqPartitioner` components are equal to the number of chares in the `FEMComp` chare array. Typically, irregular applications such as FEM computations may use the multi-partitioning approach, where the number of chare array elements are much larger than the available number of processors. In such cases, the number of messages that SeqPartitioner has to process are much larger than the number of processors, and each processor may send a number of messages to the processor where SeqPartitioner resides. An obvious optimization in such cases is to combine all the messages originating from the same processor, and send the combined message to SeqPartitioner. One can implement this optimization in the input port abstraction provided by SeqPartitioner as shown in Figure 3.23. This implementation will not simply forward the mesh connectivity it receives to SeqPartitioner chare as was done previously, but would buffer it in object group ParPartitioner until all the output ports on that processor have emitted connectivity. Then it would concatenate all the data into a single message and send it to SeqPartitioner, which would split it into different parts and carry on as before. As another optimization, instead of simply concatenating all the connectivity information of mesh partitions on the same processor into a single message, the input port implementation may be modified to eliminate duplicate mesh connectivity information resulting from duplicate boundary nodes in adjoining regions. Note that these optimizations can be carried out by the input port implementation provided by the SeqPartitioner component without the connected FEMComp component being aware of them.

Indeed, one can use a parallel mesh partitioner such as ParMETIS [52] in place of the sequential partitioner. ParPartitioner, the component wrapper around ParMETIS could be implemented as a Charm++ object group, with one representative object per processor. The input ports provided by ParPartitioner would be similar to the mesh connectivity-combining ports described above. Once again this substitution could be made completely independently of the FEMComp component.

To summarize, Charisma interface model, which is based on the specification of data each component consumes and publishes, enables independent development of reusable components. An important difference between traditional interface models and Charisma is that Charisma interfaces are contracts between components and the runtime system, rather than between the components themselves. Charisma provides control points for the runtime system

```
class ParPartitioner : public Group {
    int nregistered, nrcvd;
    vector<Partition*> parts;
    CkChareID cid;
  public:
    // SeqPartitioner creates the ports as before
    // supplying them with the ID of this group
    ParPartitioner(CkChareID id) {
      cid = id; // id of the SeqPartitioner
      nregistered = 0;
      nrcvd = 0;
    }
    // input ports register with the local group representative
    void Register(void) {
      nregistered++;
    }
    // called only from local input ports
    void RecvPartition(int p, Partition *part) {
      // buffer incoming message
      parts[p] = part;
      nrcvd++;
      // have all local messages been received ?
      if(nrcvd==nregistered) {
        Partition *comb = CombinePartitions(nregistered, parts);
        // Send the combined partition to SeqPartitioner
        CProxy_SeqPartitioner psp(cid);
        psp.RecvPartition(CkMyPe(), comb);
        nrcvd = 0;
      }
    }
    // ...
}
```

Figure 3.23: Parallel processor-based partitioner wrapper for METIS

using asynchronous method invocations, allowing the runtime system to utilize system resources more effectively. Since Charisma uses a message-driven interoperable runtime system, Converse, at its core, one can use a variety of parallel programming paradigms for building reusable components. Message-driven object-based languages such as Charm++ provides the right building blocks for building efficient components because of its close match to the Charisma runtime. Charm++ also facilitates building reusable components because it supports encapsulation and object-virtualization. However, a significant limitation of Charm++ i that it is difficult to express intra-component control-flow in the message-driven style of Charm++. We have developed a notation, Structured Dagger, that simplifies expression of control-flow for message-driven objects, and is described in the next chapter.

Chapter 4

Intra-component Coordination

Converse provides constructs for attaching code blocks to availability of specific messages. These blocks are scheduled for execution by the run-time system when the specified messages arrive. This scheme allows coexistence of multiple components within a single operating system process, while minimizing the performance impact of communication latency. Converse schedules a ready component for execution while other components are waiting for data. In an object-oriented message-driven languages, such as Charm++, these blocks correspond to methods of parallel objects, called *entry-methods.* In Charisma, computations are attached to availability of data at the input ports. For Charisma components written using Charm++, input ports are mapped to Charm++ entry methods, so that asynchronous invocation of Charm++ objects' entry methods is equivalent to making data available on input ports by emitting data to the connected output ports.

Thus a Charisma component is a flat collection of computations attached to a set of access points as shown in figure 4.1. However, real world components may have complex control-flow structure as shown in figure 4.2. Individual computations not only depend on the availability of data for them but also upon the completion of previous computations. Further, components that implement iterative parallel computations may have loops in their control-flow structures, which cannot be clearly expressed in components written in the style of figure 4.1.

In addition, The order of execution of computations is determined by the order of data

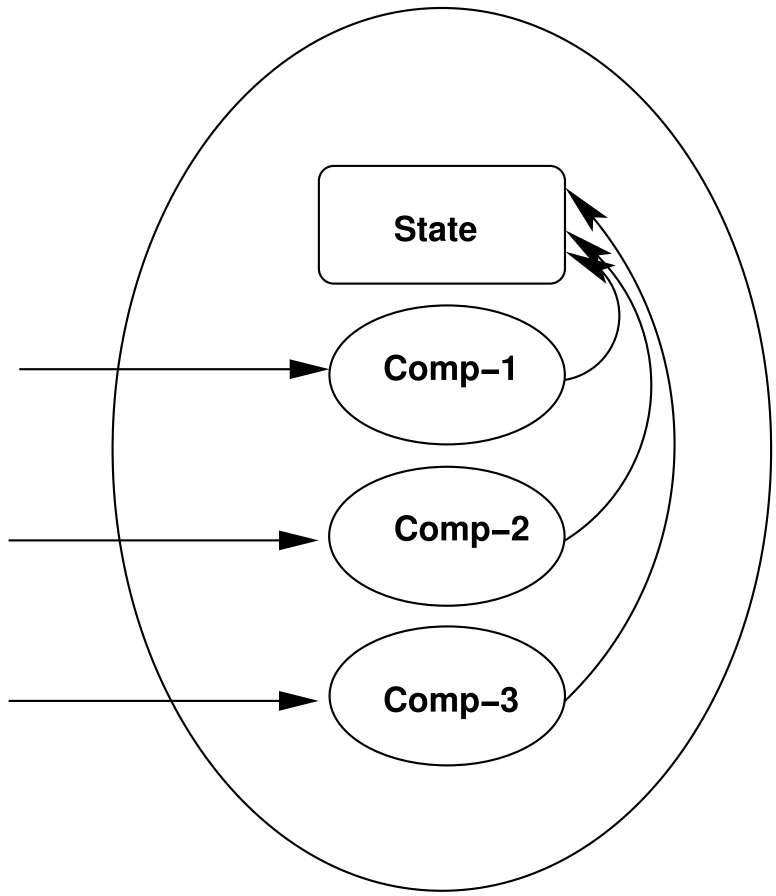


Figure 4.1: Charisma components lack explicit control-flow specification.

(bundled in Converse messages) available at the input ports and by priorities associated with these ports. Due to unpredictable delays in remote response times, the messages may arrive in any order and the programmer must deal with all possible message orderings. This increases the programming complexity significantly because the component code has to handle all possible message orderings, by buffering unexpected messages for later delivery, and by writing bookkeeping code to check and fetch from these buffers upon completion of sub-tasks. We call this code "component coordination".

The intra-component control flow can be expressed by issuing blocking receives for tagged messages in multithreaded components. This method is appropriate for explicit message-passing components that block for receiving specific tagged messages. In thread-based coordination, a blocking receive blocks only the calling thread, and not entire processor. An

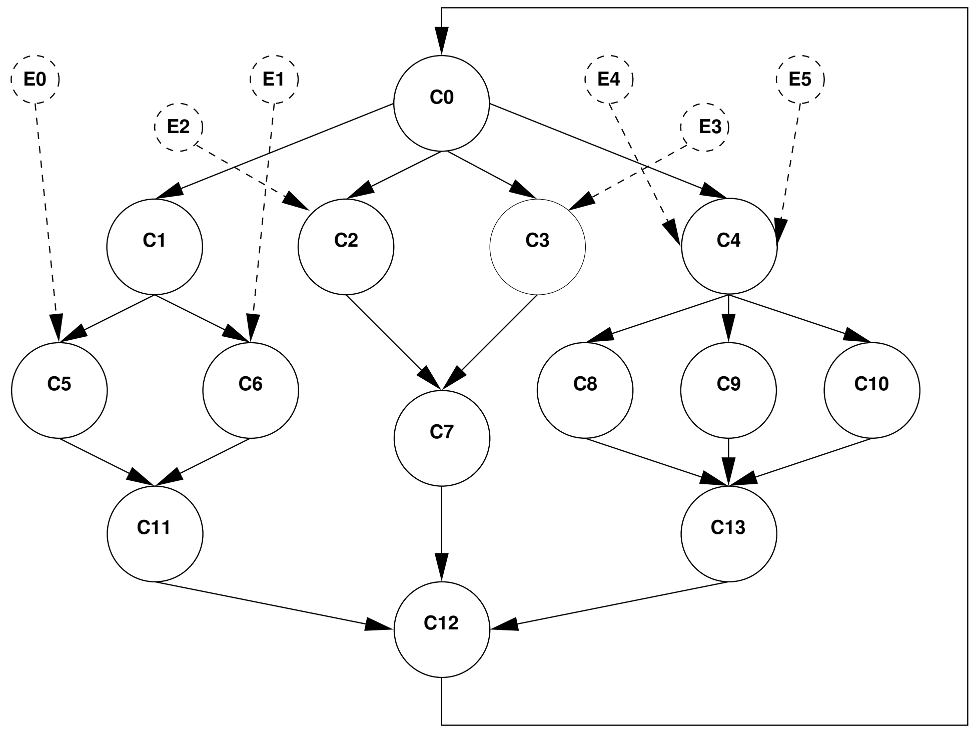


Figure 4.2: Real world components may have complex control flow structures. In this diagram, solid circles denote computations that are internal to the component. Dashed circles denote external computations. Solid lines are dependencies internal to the component, whereas dashed lines denote external dependencies among computations.

example of such orchestrator is the *sequencer* subroutine in NAMD that coordinates the actions of a *patch* object. For message-passing components that use single threaded messaging libraries such as MPI, a user-level thread is an alien concept. Thus exposing explicit thread yielding to the component developer will need major changes to made to the existing MPI-based components. However, implementation of blocking calls such as `MPI_Recv` or collective communication operations can internally suspend a thread and yield to other running threads on a processor. In the next chapter, we describe Adaptive MPI, our implementation of MPI over Converse that simplifies component coordination.

While threads simplify coordination of message-driven components, there are several disadvantages of threads. Threads have to pay a performance penalty in terms of the context-switching costs. Implicit references to threads' stacks make thread migration complicated. Also, it is difficult to precisely estimate the stacksize requirements for threads, leading to wastage of memory. At thread creation time, one has to reserve the maximum stack space that may be used by the thread in its lifetime. For example, in NAMD, the estimated stack size for sequencer threads was 128 KB each. For a molecular system of 37000 atoms divided into 343 patches, NAMD would consume about 65 MB of memory, causing it to exhaust all available memory on some machines.

Our method for simplifying component coordination avoids use of threads. Instead, our approach provides a language for expressing control-dependence graphs for components, and generating the coordination glue code from these graphs. Dagger [28] allows easy expression of such dependence graphs by introducing when-blocks and condition variables in a message-driven component. A when-block specifies dependences as a list of messages and condition variables with their associated reference numbers. A Dagger program tells the run-time system that it is at a stage to process a message by issuing an `expect` statement. A condition variable is used to signal the end of a when-block with a `ready` statement. Thus control-dependences among when-blocks belonging to the same message-driven object can be expressed using condition variables. However, the structure of Dagger programs does not clearly reflect the control flow within an object because a Dagger program is a flat collection of when-blocks.

We have developed a coordination language called Structured Dagger (Structured Dagger [39]) for this purpose. This language provides structured constructs that adequately express most commonly occurring control flow graphs viz. the series-parallel graphs.

The next section motivates simplification of component coordination with an example.

4.1 Motivating Example

Consider an algorithm for computing cutoff-based pairwise interactions between atoms in a molecular dynamics application, where interaction between atoms is considered only when they are within some cutoff distance of each other. The bounding box for the molecule is divided into a number of cubes (Patches) each containing some number of atoms. Since each patch contains different number of atoms and these atoms migrate between patches as simulation progresses, a dynamic load balancing scheme is used. In this scheme, the task of computing the pairwise interactions between atoms of all pairs of patches is divided among a number of *Compute Objects*. These compute objects are assigned at runtime to different processors. The initialization message for each compute object contains the indices of the patches. The patches themselves are distributed across processors. Mapping information of patches to processors is maintained by a replicated object called *PatchManager*.

```
class compute_object : public Chare {
 private:
  int       count;
  Patch     *first, *second;
  CkChareID chareid;
 public:
  compute_object(MSG *msg) {
    count = 2; MyChareID(&chareid);
    PatchManager->Get(msg->first_index,recv_first, &chareid,NOWAIT);
    PatchManager->Get(msg->second_index,recv_second, &chareid,NOWAIT);
  }
  void recv_first(PATCH_MSG *msg) {
    first = msg->patch;
    filter(first);
    if (--count == 0 ) computeInteractions(first,second);
  }
  void recv_second(PATCH_MSG *msg) {
    second = msg->patch;
    filter(second);
    if (--count == 0) computeInteractions(first,second);
  }
}
```

Figure 4.3: Charm++ Implementation of "Compute-Object" in Molecular dynamics

Figure 4.3 illustrates the Charm++ implementation of the flow of control in compute object as illustrated in figure 4.4. Each compute object requests information about both patches assigned to it from the PatchManager. PatchManager then contacts the appropriate processors and delivers the patch information to the requesting compute object. The compute

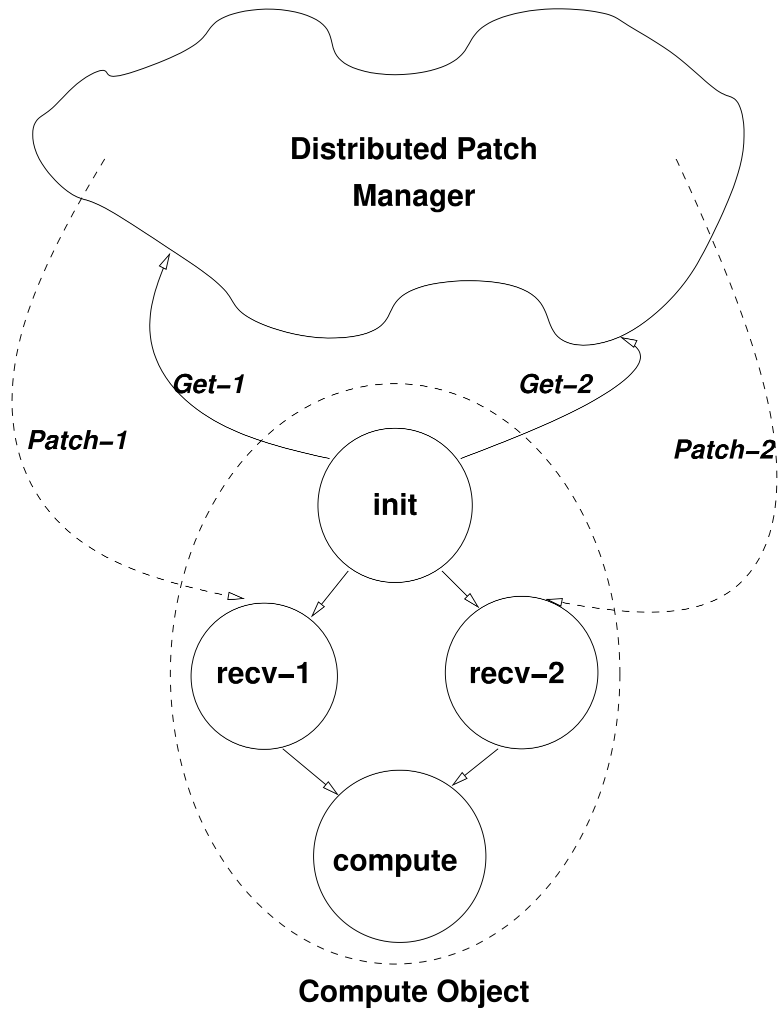


Figure 4.4: Flow of control in compute-object.

object, after receiving information about each patch, determines which atoms in a patch do not interact with atoms in other patch since they are apart by more than the cut-off distance. This is done in the `filter` method. Filtering could be done after both patches arrive. However, in order to increase overlap between computations and communications, we do it immediately after any patch arrives. Since the patches can arrive at the requesting compute object in any order, the compute object has to buffer the arrived patches, and maintain state information using counters or flags. This example has been chosen for simplicity in order to demonstrate the necessity of counters and buffers. In general, a parallel algorithm may have more interactions leading to the use of many counters, flags, and message buffers, which complicates program development significantly.

4.2 Threaded Coordination

```
void compute_thread(int idx1, int idx2)
{
  getPatch(idx1);
  getPatch(idx2);
  threadId[0] = createThread(recvFirst);
  threadId[1] = createThread(recvSecond);
  threadJoin(2, threadId);
  computeInteractions(first, second);
}
void recvFirst(void)
{
  recvPatch(first, FIRST_TAG);
  filter(first);
}
void recvSecond(void)
{
  recvPatch(second, SECOND_TAG);
  filter(second);
}
```

Figure 4.5: Multi-threaded Implementation of "Compute-Object" in Molecular dynamics

Contrast the compute-object example in Figure 4.3 with a thread-based implementation of the same scheme in Figure 4.5. Functions `getFirst`, and `getSecond` send messages asynchronously to the PatchManager, requesting that the specified patches be sent to them, and return immediately. Since these messages with patches could arrive in any order, two threads `recvFirst`, and `recvSecond` are created. These threads block waiting for messages to arrive. After each message arrives, each thread performs the filtering operation. The main thread waits for these two threads to complete, and then computes the pairwise interactions. The threaded implementation eliminates the programming complexity of buffering the messages and maintaining the counters. However, this implementation has added costs of thread creation, context switching, and synchronization.

4.3 Structured Dagger

In order to reduce the complexity of program development, as demonstrated in Figure 4.3 without adding any overheads such a thread creation and context switching (Figure 4.5), a coordination language called Structured Dagger has been developed on top of Charm++. Structured Dagger language is defined by augmenting Charm++ with structured entry-methods, which specify pieces of computations (when-blocks) and dependences among computations and messages. A when-block is guarded by dependences that must be satisfied before it can be scheduled for execution. These dependences include arrival of messages or completion of other constructs. Before describing the Structured Dagger language in detail, let us consider the molecular dynamics example once again, and show how it can be coded in Structured Dagger.

```
class compute_object
sdagentry compute_object(MSG *msg){
  atomic {
    PatchManager->Get(msg->idx1,...);
    PatchManager->Get(msg->idx2,...);
  }
  overlap {
    when recv_first(Patch *first)
      atomic { filter(first); }
    when recv_second(Patch *second)
      atomic { filter(second); }
  }
  atomic {
    computeInteractions(first, second);
  }
}
```

Figure 4.6: Structured Dagger Implementation of "Compute-Object" in Molecular dynamics

Figure 4.6 shows the compute object written using Structured Dagger. Whenever the entries `recv_first` or `recv_second` receive messages, the `filter` method is called. After both patches arrive, the `computeInteractions` function is called. Structured Dagger takes care of the bookkeeping functions such as incrementing counters, flags and buffering the messages. Therefore, the resulting code is more readable (consequently easy to program and modify), and retains the performance benefits of the message-driven model.

4.3.1 Structured Constructs

Constructs in Structured Dagger include specification of dependence of computation on messages (when-blocks), atomic, ordering, and loop constructs.

When-Blocks When-blocks specify dependences between computation and message arrival at an entry-method. In general, a when-block may specify its dependence on more than one entry-method. When all constituent entry-methods receive messages, computation corresponding to the when-block may be triggered. Simplified syntax of a when-block is shown in Figure 4.7.

```
when entry1 (MsgType1 *m1),
     entry2 (MsgType2 *m2),
     entryN (MsgTypen *mN) {
  <Structured Dagger Constructs >
}
```

Figure 4.7: Simplified syntax of a when-block

Note that the computations are not contained within entry methods. Indeed the entry methods are generated by the Structured Dagger translator to perform dependence checking and buffering the received messages. All computations in a structured entry-method are performed inside "atomic" blocks described next.

Atomic Construct The atomic construct is a wrapper around C++ statements and specifies that no Structured Dagger constructs appear inside it. Since no structured construct exists inside an atomic construct, it does not contain code dependent upon arrival of remote messages and is therefore executed atomically.

Ordering Constructs Receiving a message for an entry-method is not sufficient to trigger a computation. The computation must be in a state where it is ready to process the message. Even if all the entry-methods specified in a when-block have received messages, the computation specified in the when-block is not triggered because other constructs occurring previously in the program order may not have completed. The program order is specified in Structured Dagger using the ordering constructs, *seq* and *overlap*. The seq construct is written as `{component-construct-list}` and ensures that each of the constructs in the list is enabled only after its predecessor completes. The seq construct completes when the last of its component constructs reaches completion.

The overlap construct enables all its component constructs concurrently and can execute these constructs in any order. Actual execution of these component constructs may be dependent on arrival of messages that trigger them. An overlap construct reaches its completion only after each of its component constructs has completed. Note that the component constructs are not executed in parallel even if computational resources are available. In particular, if the component constructs of an overlap consist only of atomic code blocks, these computations are guaranteed to execute atomically one after the other. This is different from concurrency in the context of preemptive multithreading.

Conditional and Looping Constructs In many situations, one may need to conditionally enable the Structured Dagger constructs, or to iterate over a set of constructs. For this purpose, Structured Dagger provides *if*, *for*, *while*, and *forall* constructs. The syntax of the first three constructs is the same as the equivalent statements in C++. However, semantically they are different from their C++ counterparts because they may include when-blocks as their component constructs and thus, may suspend and transfer control to the Converse scheduler. A forall construct enables its component constructs for the entire iteration space as opposed to the while and for constructs, which enable their component constructs for each element of the iteration space in strict sequence. Messages intended for different iterations of the looping constructs are marked by "reference numbers" using function `SetRefNum(msg)`. The when-block construct is augmented with the expected reference number specification for each message.

With these structured constructs, one can easily express complex control-flow graphs, such as the one depicted in figure 4.2. The corresponding Structured Dagger code is shown in figure 4.8. The *atomic* wrappers for code-blocks (C_n) are not shown for brevity.

Reference Numbers When-blocks combined with the ordering constructs are adequate for specifying computations where concurrent phases of the same computation do not have to co-exist. However, in many practical problems, such as Jacobi Relaxation in numerical methods, many phases of the same computation may be running concurrently. Since Charm++ does not enforce in-order delivery of messages, messages intended for different phases of computations may get mixed if they arrive out-of-order and as a result, the results of the computation can be unpredictable. A scenario where the computation can go wrong is illustrated in Figure 4.9. Processors i and j are exchanging messages and doing some local computation. The message sent by processor i in the second iteration is delayed. When the processor i receives a message from j in the third iteration, it performs the local computation and sends the message belonging to the fourth iteration. Processor j receives the message which belongs to the fourth iteration before the one belonging to the third iteration.

In order to keep different phases of the computations separate without cluttering the name spaces of entry-methods and message types, Structured Dagger provides *reference numbers* attached to messages to distinguish between messages belonging to different phases of computation. A when-block optionally specifies reference numbers for the messages triggering its constituent entry-methods. Messages that belong to the same phase of the computation are given specific reference numbers by the user. Structured Dagger matches the messages with those reference numbers to activate a when-block. Reference numbers of messages are accessed and set by the function calls provided by the system: `GetRefNumber(msg)`, and

```
sdagentry control_flow(void){
    while ( loop-condition ) {
        seq {
            C0;
            overlap {
                seq {
                    C1;
                    overlap {
                        when E0 { C5; }
                        when E1 { C6; }
                    }
                    C11;
                }
                seq {
                    overlap {
                        when E2 { C2; }
                        when E3 { C3; }
                    }
                    C7;
                }
                seq {
                    when E4, E5 { C4; }
                    overlap {
                        C8;
                        C9;
                        C10;
                    }
                    C13;
                }
            }
            C12;
        }
    }
}
```

Figure 4.8: Structured Dagger Implementation of the control-flow depicted in figure 4.2

`SetRefNumber(msg)`.

Example Program We present an example Structured Dagger program that implements the Harlow-Welch scheme in Computational Fluid Dynamics [28]. In this method, the three dimensional computational space is first divided along X and Y axes into rectangular re-

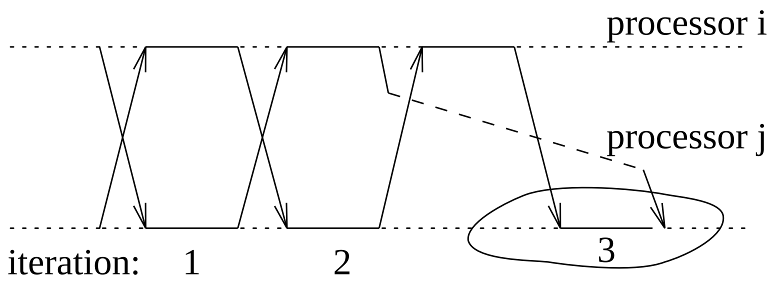


Figure 4.9: Out of order messages

```
class Harlow_Welch
sdagentry Harlow_Welch(MSGINIT *msg) {
    atomic {
      initialize();
      for(i=0; i<NZ; i++)
         convdone[i] = FALSE;
    }
    forall[i](0:NZ-1,1) {
      while (!convdone[i]) {
        atomic {
          for (dir=0; dir<4; dir++) {
            m[i][dir] = copy_boundary(i,dir);
            SendMsgBranch(entry_no[dir],m[i][dir],nbr[i][dir]);
          }
        }
        when[i] North(Bdry *n),
                South(Bdry *s),
                East(Bdry *e),
                West(Bdry *w) {
          atomic {
            update(i, n, s, e, w);
            reduction(my_conv, i, Converge, &mycid);
          }
        }
        when[i] Converge(Conv *c) {
          atomic { convdone[i] = c->done; }
        }
      }
      atomic { print_results(); }
    }
}
```

Figure 4.10: Harlow-Welch Program

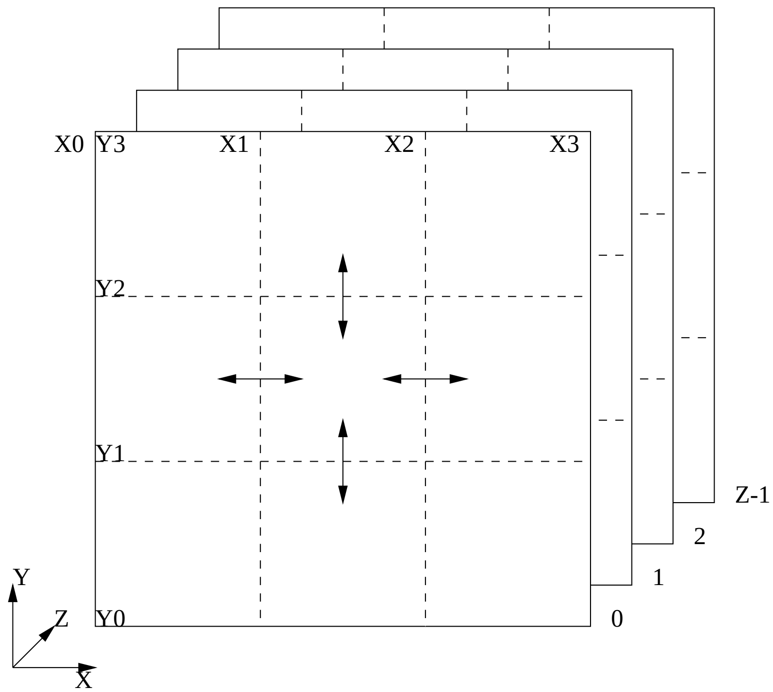


Figure 4.11: Spatial Decomposition in Harlow-Welch Scheme

gions (Figure 4.11). In the implementation shown in figure 4.10, the region ($X_i..X_{i+1}$, $Y_i..Y_{i+1}$,0..$Z-1$) is encapsulated in a chare object of class `Harlow_Welch`. These objects are distributed across the processors of the parallel machine. This encapsulated region is then implicitly divided along the Z axis into N_Z parallel planes. Each iteration in this scheme consists of exchanging the boundary values of the previous iteration with up to four immediate neighbors in the 2-D grid. Each object then updates its own region based on the values from the previous iteration as well as the boundary values obtained from the neighbors. This is followed by a global reduction to determine error and to check whether the scheme has converged. (The reduction is carried out asynchronously by a separate branch office object, called *reduction manager*, and is not shown here. After the reduction is complete, the reduction manager sends messages to the participating objects at specified entry-methods.) This is done concurrently for all the planes and each of the planes could converge independently of each other. The `forall` construct in figure 4.10 implements concurrent convergence across the N_Z planes, whereas the `while` construct implements convergence of each plane. The plane index `i` is used as a reference number for messages participating in convergence for that plane.

4.3.2 Implementation

Structured Dagger is implemented on top of Charm++ as a translator and a run-time library. The Structured Dagger translator transforms the program to an equivalent Charm++ program. The translation consists of splitting a structured entry-method into a number of Charm++ entry-methods and private methods, inserting counters and flags to specify dependences between different component constructs of the structured entry-method. For each construct, the translator generates object-private methods for enabling the construct and for the completion of the construct. A construct is enabled when all its predecessors in the program order have been satisfied, and it is completed when all its component constructs have finished.

The runtime library maintains a collection of message queues and a list of pending when-blocks within every object that contains structured entries. The message queues are indexed by the entry-method numbers. Whenever any when-block is enabled, it checks if the messages intended for its component entries have already arrived in the message queue. If all of these are available, it enables its constituent constructs and if possible executes them. In particular, atomic constructs do not have dependence on message arrival, therefore the code generated for when-block executes code in those constructs. When a message directed at an entry method arrives, the generated code for that entry method inserts the message in an appropriate queue. It then checks to see if any when-blocks have been waiting for that entry to be triggered. If a when-block is waiting and has all its dependences satisfied with the arrival of this message, its component constructs are enabled, and if possible, executed. If this message has arrived out of order (i.e. when no when-block was waiting for it), the entry method buffers the message and sets appropriate flags indicating its availability. By doing a careful analysis of the dependence structure, the translator avoids periodic checking for all

Program	Charm++	Structured Dagger	Multi-threaded
Reverse Order	0.13	0.20	0.39
Random Order	0.15	0.20	0.43

Table 4.1: Comparison of Structured Dagger, Charm++, and Threads. All timings are in seconds.

enabled when-blocks and only when-blocks that may be waiting for a particular entry-method are enabled.

For assessing the performance impact of our translation scheme, we ran two simple programs on a single node of SGI Origin2000. The first program created two objects (threads), supplying one of them with a seed message, which then started sending messages with reference numbers in the reverse order as that expected by the other object (thread). The other object (thread), upon receiving a message simply bounced it back to the first object (thread). The second program also created two objects (threads), but supplied the first object (thread) with a random order of reference numbers. The first object (thread) sent messages to the second object (thread) with reference numbers in the supplied order, while the other object (thread) received them in sequential order.

We compared the performance of our Structured Dagger programs with Charm++ programs and also with multi-threaded programs written using thread-objects in Converse. The results for 10000 round-trip messages (each of size 4 bytes) are in Table 1. As can be seen from these results, Structured Dagger program does not add significant overhead to the native Charm++ code while reducing the program complexity. The cost of context-switching in a multi-threaded program is significant when compared with Structured Dagger and Charm++ in the absence of any computation between the coordination steps.

Structured Dagger eliminates stack-size estimation required for multi-threaded components. Stack-size estimation is a non-trivial task for most real world applications, especially when system library functions are invoked. One has to conservatively estimate the stack size in order to avoid runtime errors of stack overflow. This leads to wastage of memory space. Structured Dagger methods are invoked from the message-driven scheduler, and thus execute on the default process stack, which is typically allocated in chunks by the operating system on demand, and thus does not require conservative estimation, thus leading to reduced memory requirements.

In order to make it possible for NAMD to simulate large molecular systems on machines with limited resources, we implemented the sequencer as a message-driven object using Structured Dagger. This new version of NAMD reduced the memory requirement significantly (consuming about 3.7 MB instead of 65 MB required for the threaded version, and it was possible to use NAMD on machines with limited memory without any performance degradation.

4.3.3 Inadequacy of Structured Dagger

Structured Dagger is adequate to express the series-parallel control flow graphs that occur in most parallel objects in real-world applications. However, in some cases, the control-flow graphs of objects do not conform to this restriction. Figure 4.12 shows one such control flow graph. The annotated circles represent code-blocks to be executed. The arrows show dependencies between code-blocks. Note that only the dependencies within an object containing these code-blocks are shown. Each of these code-blocks may have additional external dependencies such as message arrival from remote processors. If one tries to code this graph as a Structured Dagger program, this introduces spurious dependencies and can slow the program down. Figure 4.13 shows two possible implementations in Structured Dagger. In implementation 4.13(a), C_4 has to wait for C_3 to finish execution and in implementation 4.13(b), C_3 unnecessarily waits for C_2 to finish execution. In order to express such control flow graphs correctly, a lower-level mechanism is needed. Dagger [28], a notation that makes use of unstructured constructs such as when-blocks and condition-variables can be used to express such control-flow graphs.

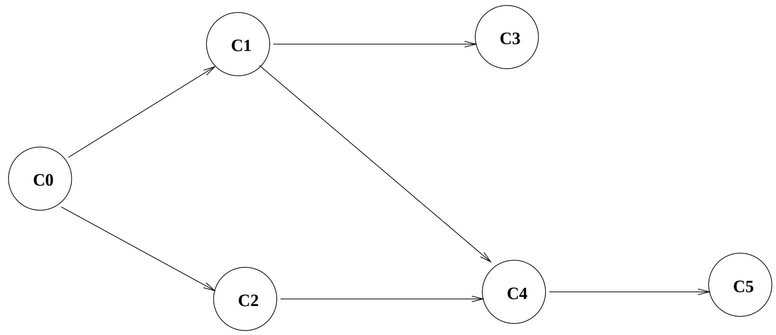


Figure 4.12: A Non-Series-Parallel Control-Flow Graph

```
seq {
   C0;
   overlap {
     seq { C1; C3; }
     C2;
   }
   C4;
   C5;
}
```

(a) An Structured Dagger implementation of Figure 4.12

```
seq {
   C0;
   overlap {
     C1;
     C2;
   }
   overlap {
     C3;
     seq { C4; C5; }
   }
}
```

(b) Another Structured Dagger implementation of Figure 4.12

Figure 4.13: Possible Structured Dagger implementations for the Non-Series-Parallel Graph (Fig. 4.12)

4.3.4 Related Work

CC++ [18] is an object-parallel language that bears some similarities to Structured Dagger. CC++ is a thread-based system. A computation consists of one or more processor objects each with its own address space. Objects within these processor objects can be accessed by remote objects using global pointers. Within individual processor objects, new threads can be spawned using the structured constructs *par*, and *parfor*, and the unstructured construct *spawn*, which creates a new parallel thread. Multiple threads created by these statements may be executed by different processors, or interleaved on the same processor, and they may share variables.

The *par* and *parfor* constructs of CC++ are analogous to the *overlap*, and *forall* constructs in Structured Dagger. However, they are different in a fundamental sense: two statements in a *par* construct may actually be executed in parallel by two different processors, whereas two constructs in an *overlap* statement are always executed by the same processor. Also they can interleave only in a disciplined fashion: only entire when-blocks can be interleaved, based on the arrival of messages, and not the individual C++ statements.

The most important difference between Structured Dagger and CC++ (and other systems such as Chant [29]) has to do with threads. Using threads creates a flexibility, but at a cost: thread context switches are more expensive than message-driven invocations of methods in Charm++ or Structured Dagger. Also, threads waste memory: creating hundreds or thousands of threads, each with its own stack, may not be possible, whereas a large number of parallel objects can easily be created without reaching memory limits.

ABC++ [4] is a thread-based object-parallel language. Both synchronous and asynchronous remote method invocations are allowed. There is a single thread associated with each parallel object. This thread receives messages corresponding to method invocations and decides when and whether to invoke methods. Primitives are provided to selectively enable execution of individual methods. Unlike Structured Dagger, no direct expression of control flow across method invocations is possible.

The enable set construct [20] addresses the issue of synchronization within Actors [1]. Using this, one may specify which messages may be processed in the new state. Any other messages that are received by an actor are buffered until the current enable set includes them. The ordering constructs in Structured Dagger achieve this in a cleaner manner. Also, there is no analogue of a when-block, viz. a computation block, that can be executed only when a specific group of messages have arrived.

While Structured Dagger presents significant performance advantages because it does not use threads while simplifying expression of control flow within a component, for some cases, such as when converting legacy codes to run on top of the message-driven Converse system, it is sometimes easier to use threads. Threaded Simple Messaging (tSM), Threaded message-passing objects (TeMPO), and Parallel Array of Threads (PATH) languages implemented on top of Converse allow one to construct interoperable threaded components. However, for converting legacy codes such as those written in MPI, a better solution is possible as a migration path that uses Converse's user level threads, and the MPI syntax for message-passing. We describe Adaptive MPI, our threaded implementation of MPI on top of Converse in the next chapter.

Chapter 5

Migration Path: Adaptive MPI

Research on parallel computing has produced a variety of programming paradigms, and many of them have a large software base. A large amount of parallel software in use today is based on the message-passing paradigm as embodied by MPI. Support for migration of such legacy codes to any new programming system is critical for the success of that system. In this chapter, we describe how legacy MPI codes can be converted to become parallel Charisma components.

Traditional MPI runtime systems assume a single-threaded process on every processor, which will block the entire processor on a blocking receive call. By modelling MPI processes with Converse's user-level threads, parallel components written using MPI can be made to co-exist with other components in a Charisma-based parallel application. Adaptive MPI (*AMPI*), our implementation of MPI on top of Converse, uses Converse's user-level threads to run the component code. AMPI design is similar to our earlier efforts "Threaded Simple Messaging" (tSM), "Parallel Array of Threads" (PATH), and "Threaded Message Passing Objects" (TeMPO). In particular, the PATH library on Converse aimed at providing Chant-like [29] abstraction of a group of threads. One critical difference between these earlier efforts and AMPI is that AMPI was built from scratch with the main objective of providing easier migration path for legacy codes. Therefore, message-passing primitives of AMPI have the same syntax and semantics as MPI, which makes it simpler to adapt existing components written in MPI to run on our component architecture.

There are two issues that need to be addressed in a multi-threaded implementation of MPI. First is the overhead associated with thread context-switching and synchronizations. We have demonstrated that Converse's user-level threads have very low overheads. Even these low overheads can often be compensated for by the cache-performance benefits of a technique called "overdecomposition" or "multipartitioning". The second issue arises while converting existing MPI components to AMPI. Since the original single threaded MPI component codes need to co-exist with other instances of the component running the same code on the same processor, one has to make sure that the MPI component does not reference any writable global variables.

We discuss these issues in detail next.

5.1 Overhead of Threaded Components

Threads that block waiting for messages transfer control to other runnable threads on the same processor (yielding the processor) via the Converse scheduler, causing a thread context-switch. Converse threads are user-level, which already have very low context-switch overhead compared to alternatives such as kernel-level threads. However, the percentage overhead introduced by the use of threads depends on the grainsize of a component. If the grainsize of a component (in this context, the average time spent by the component between successive coordination steps or blocking communication calls) is large, one can indeed tolerate the overhead introduced by threads. On today's processors, the typical context switching time for a user-level thread is less than half a microsecond. For a scientific application with near neighbor communication, each partition would typically have six message-receives, out of which, three will block. Thus, in each timestep, there will be on an average three context-switches for each partition. If the computation time for each partition is large, say a few milliseconds, the context switching overhead would be less than 0.1%, which can be tolerated especially since it gives us vital capabilities of dynamic load balancing. In addition, "overdecomposing" a message-passing component (i.e. having many more "chunks" of a parallel component than the number of available physical processors) often allows the component to compensate for the thread overhead since the threaded message-passing component is more latency tolerant. When a chunk waits for data, the runtime system can schedule other ready chunks for execution, thus effectively overlapping communication with computation. Multi-partitioning can also exhibit better cache utilization in application components that are not optimized for caches.

To evaluate the overhead introduced by threaded components with "overdecomposition", we carried out an experiment using a Finite Element Method application that does structural simulation on an FEM mesh with 300K elements. We ran this application on 8 processor Origin2000 (250 MHz MIPS R10000) with different number of partitions of the same mesh mapped to each processor. Results are presented in figure 5.1. It shows that increasing number of chunks is beneficial up to 16 chunks per physical processor. This increase in performance is caused by better cache behavior of smaller partitions, and overlap of computation and communication (latency tolerance). Though these numbers may vary depending on the application, we often see similar behavior for many applications that deal with large data sets and have near-neighbor communication.

As another test for evaluating the efficiency of the overdecomposition, we studied a Conjugate Gradient Solver[1]. This component is a partial differential equation solver which uses a sparse, matrix-free form of the conjugate gradient method to solve the Poisson problem on a regular 2D grid. The mesh size for this problem is 1000×1000; equivalent to a million-row matrix. Each thread is responsible for computing the solution on a rectangular region of the mesh. Since the solution residual for a grid point depends on the solutions for its nearest four neighbors, each processor maintains a one-element-thick ghost region. In each step, messages are exchanged to fill these ghost regions, and there are two short global

[1] The Conjugate Gradient Solver was written by Orion Lawlor as part of a class project at the University of Illinois.

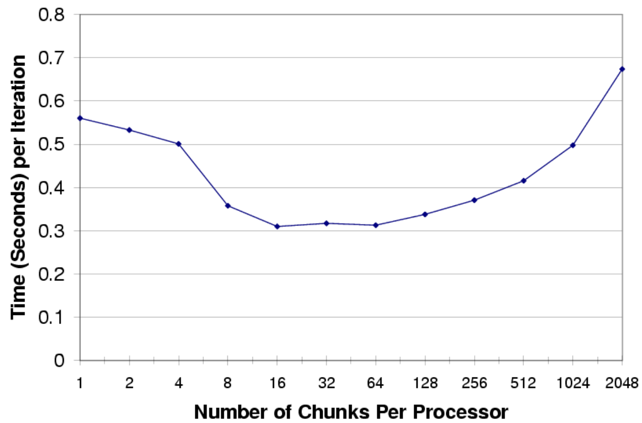


Figure 5.1: "Overhead" of threaded components

reductions. Like many scientific codes, this application is normally memory bandwidth bound. Figure 5.2 shows the time per step of the solver on a single physical processor, while varying the number of virtual processors between 1 and 4096. Because AMPI's virtual processors are implemented as user-level threads, there is very little overhead in managing the threads. On our Pentium IV system, with a relatively small cache but very fast RDRAM memory, simulating 100 virtual processors led to only a slight (10%) slowdown. However, for the Athlon and Pentium III Xenon, with their large caches and slower memory systems, simulating 100 virtual processors was actually slightly *faster* than using the single physical processor normally. Thus the single-processor virtualization efficiency is very high.

5.2 Obstacles to Migration

While it is clear that threads simplify coordination of message-driven parallel components, they present obstacles to important across-the-module services offered by our common language runtime system, Converse. One particularly important service is migration-based

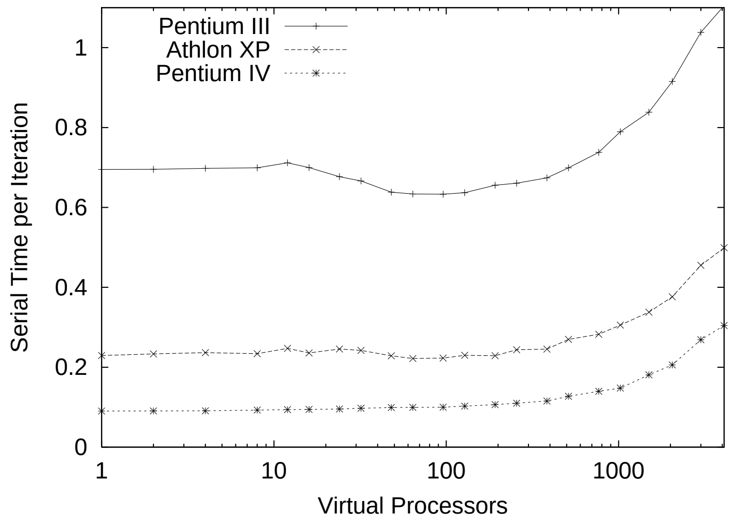


Figure 5.2: Performance of Conjugate-Gradient solver on a single processor.

dynamic load balancing. Converse load balancing framework keeps track of computational loads and communication patterns of each component, and periodically migrates components from overloaded processors to underloaded ones. Threads impede migration because of references to thread stacks. Local variables in subroutines are stored on the thread stack, and other variables could contain references to the stack variables. Memory for thread stacks is allocated dynamically, and may occupy different virtual addresses when migrated to another processor. Therefore, references to thread stacks may become invalid when a thread migrates from one address space to another. This situation is illustrated in figure 5.3.

The code running inside a thread (see figure 5.4) calls a function and passes a local integer array (`iarray`) as a parameter to that function. When the thread migrates while still inside a called function, the reference to the local array becomes invalid upon migration. In the absence of compiler support, which may detect such stack references, and change them appropriately when a thread migrates, migrating threads is difficult. For thread migration to work, one has to make sure that the address spaces occupied by the thread stack remains unchanged on any processor where a thread can migrate.

Our preliminary implementation of migratable threads was based on a stack-copy mech-

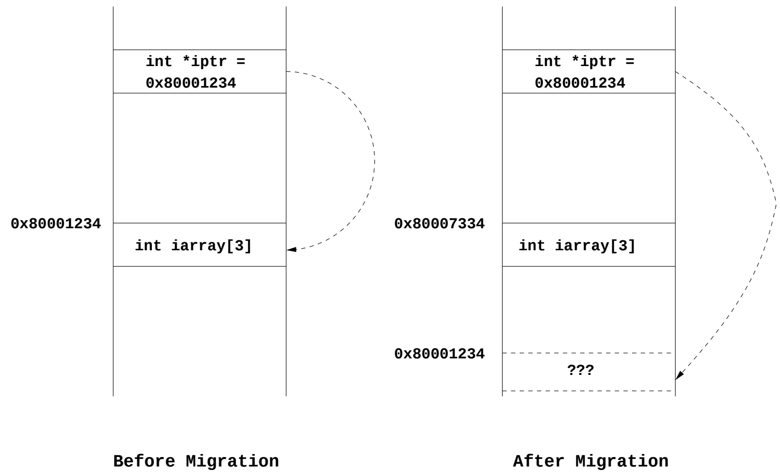


Figure 5.3: Threads impede migration, since references to stack variables may become invalid upon migration if stacks are dynamically allocated.

```
void foo(void)
{
  int iarray[3];
  // initialize iarray
  bar(iarray);
}

void bar(int *iptr)
{
  ... // use *iptr
  migrate();
  ... // use *iptr
}
```

Figure 5.4: An example of references to stack variables

anism, where contents of the thread-stack were copied into and out of the process stack at every context-switch between two threads. Thus, all threads execute with the same stack and refer to valid addresses even after migration (assuming that the main process stack on all processors begins at the same virtual address which is true for most parallel machines when using the same compiler and operating environment.) This has two drawbacks. First, it is inefficient because of the copy overhead on every context-switch (figure 5.5), and second, one thread cannot access another thread's local variables, thus making it mandatory to copy it to some shared location. The context-switching time of stack-copying threads varies with the amount of stack space in use at the time of context switch. At context-switching time, the system determines the size of the buffer required to set aside the filled stack, allocates memory, copies the stack out to the buffer, and copies the buffered stack of another thread in. In order to ensure efficiency with this mechanism, it is recommended to keep the stack size as low as possible at the time of a context-switch.

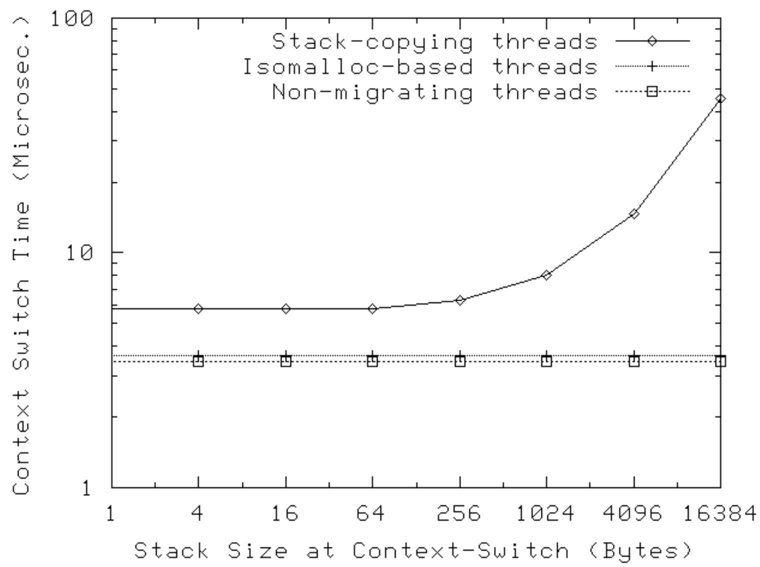


Figure 5.5: Comparison of context-switching times of stack-copying and isomalloc-based migrating threads with non-migrating threads. This experiment was performed on NCSA Origin2000, with 250 MHz MIPS R10000 processor.

Our current implementation of migratable threads uses a new scalable variant of the *isomalloc* functionality of PM^2 [3]. In this implementation, each thread's stack is allocated such that it spans the same reserved virtual addresses across all the processors. This is achieved by splitting the unused virtual address space among physical processors. Figure 5.6 shows a typical layout of the address space of a Unix process. The heap and the stack expand dynamically. However, by limiting both to an upper bound, one gets a large amount of unused virtual address space. The boundaries of the unused virtual address space are marked by making the boundary pages inaccessible to the process. When a thread is created, its stack is allocated from a portion of the virtual address space assigned to the creating processor. This ensures that no thread encroaches upon addresses spanned by others' stacks on any processor.

Allocation and deallocation within the assigned portion of virtual address space is done using the `mmap` and `munmap` functionality of Unix. Thus, the size of the process page table is still determined by the amount of data actually in use. Since we use isomalloc for fixed size thread stacks only, we can eliminate several overheads associated with PM^2 implementation of isomalloc. This results in low thread-creation overheads. Context-switching overheads for isomalloc-based threads are as low as non-migrating threads, irrespective of the stack-size. However, it is still more efficient to keep the stack size down at the time of migration to reduce the thread migration overhead.

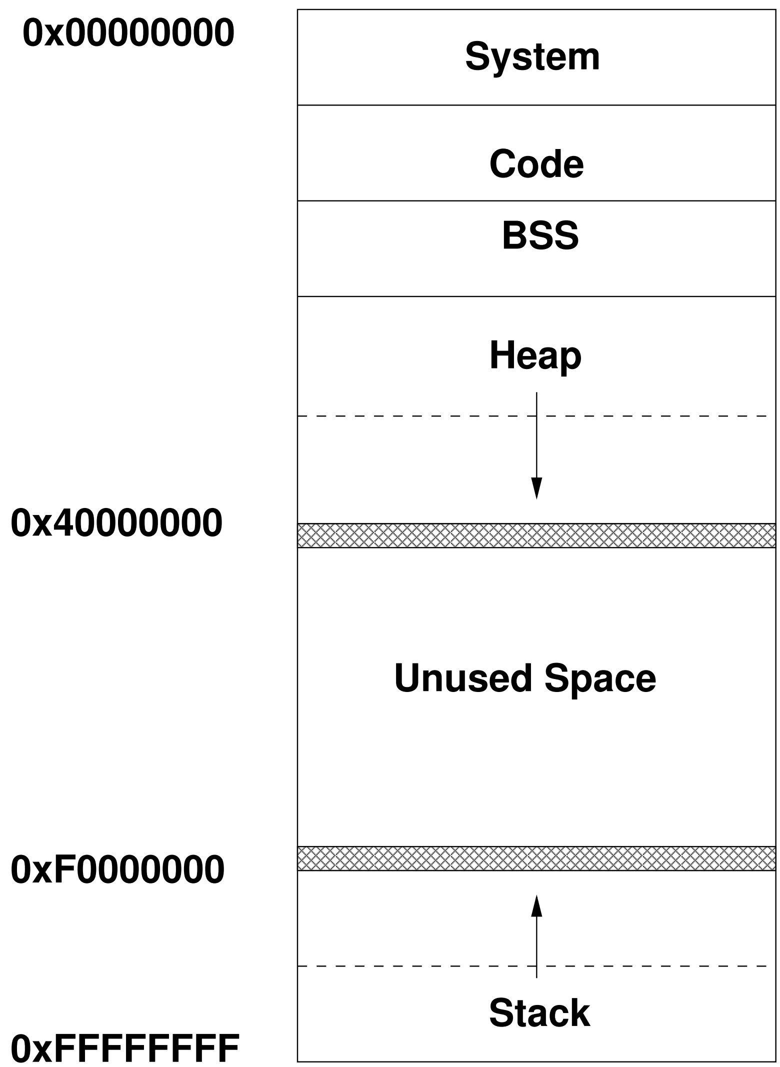


Figure 5.6: Address space layout of a typical Unix process.

5.3 Conversion to AMPI

In order to convert existing MPI application components to AMPI, one has to make sure that variables global in scope (such as common blocks, global variables etc) are not defined before and used after a blocking call, such as `MPI_Recv`. The reason for this restriction is straightforward. If a global variable is defined before a blocking call, it may be modified by another user-level thread when the defining thread blocks and another thread is scheduled. Thus, after returning from a blocked MPI call, the original thread will see a different value of that variable. In order to use AMPI, such variables should be localized (copied to variables local to the subroutines, which are on stack) or privatized (made accessible only through thread-private variables).

One typically finds three kind of global variables in MPI programs. The first type are variables that are initialized at startup (say by reading them from a configuration file), and never modified. This type of variables need not be privatized for each thread, since each thread sets and expects the same values. The second type are temporary variables that, though they have global storage scope, are defined and used only within the scope of a small set of subroutines. If there are no blocking MPI calls made while the variables are live, this kind of variable need not be privatized because AMPI threads are non-preemptive. Finally, there are truly global variables, which have different values for each thread but long lifetimes. These global variables must be privatized. Careful inspection of the program may reveal such variables.

However, sometimes such careful inspection may not be possible. In that case, we have devised a method to systematically put all the global variables in a private area allocated dynamically or on thread's stack. The idea is to make a user-defined type, and make all the global variables members of that type. In the main program, we allocate a variable of that type, and then pass a pointer to that variable to every subroutine that makes use of global variables. Access to these global variables in such subroutines should be made through this pointer. This is equivalent to converting the original procedural component an object-based component, where the object state encapsulates the previously global data, and all the procedures that operated on these global data become that object's instance methods.

```
MODULE shareddata
  INTEGER :: myrank
  DOUBLE PRECISION :: xyz(100)
END MODULE

PROGRAM MAIN
  USE shareddata
  include 'mpif.h'
  INTEGER :: i, ierr
  CALL MPI_Init(ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
  DO i = 1, 100
    xyz(i) =  i + myrank
  END DO
  CALL subA
  CALL MPI_Finalize(ierr)
END PROGRAM

SUBROUTINE subA
  USE shareddata
  INTEGER :: i
  DO i = 1, 100
    xyz(i) = xyz(i) + 1.0
  END DO
  CALL MPI_Send(....)
c blocking call: potential context-switch
  CALL MPI_Recv(....)
END SUBROUTINE
```

Figure 5.7: Original MPI program

This conversion process is illustrated in figures 5.7, 5.8, and 5.9. Figure 5.7 shows the original MPI code. The main program and subroutine `subA` share an array `xyz` and a variable `myrank` as global variables. We first aggregate the shared global variables into a user-defined type called `chunk` as shown in figure 5.8. We change the main program to be a subroutine `MPI_Main`. Then we modify the `MPI_Main` subroutine to dynamically allocate a thread-private variable of the `chunk` type and change the references to them. Subroutine `subA` is then modified to take this variable as argument. Code in figure 5.9 shows the converted AMPI program.

```
MODULE shareddata
  TYPE chunk
    INTEGER :: myrank
    DOUBLE PRECISION :: xyz(100)
  END TYPE
END MODULE
```

Figure 5.8: Conversion to AMPI: Shared data is aggregated in a user-defined type.

```
SUBROUTINE MPI_Main
  USE shareddata
  USE AMPI
  INTEGER :: i, ierr
  TYPE(chunk), pointer :: c
  CALL MPI_Init(ierr)
  ALLOCATE(c)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, c%myrank, ierr)
  DO i = 1, 100
    c%xyz(i) =  i + c%myrank
  END DO
  CALL subA(c)
  CALL MPI_Finalize(ierr)
END SUBROUTINE

SUBROUTINE subA(c)
  USE shareddata
  TYPE(chunk) :: c
  INTEGER :: i
  DO i = 1, 100
    c%xyz(i) = c%xyz(i) + 1.0
  END DO
END SUBROUTINE
```

Figure 5.9: Conversion to AMPI: References to shared data are made through thread-private variables.

This conversion process, though mechanical, is cumbersome, and can indeed be automated. We have developed *AMPIzer*[2] [56], a simple prototype source-to-source translator, based on the Polaris [13] compiler front end. AMPIzer can recognize all global variables in FORTRAN90 or FORTRAN77 programs and automatically converts the program to be a threaded AMPI component.

Large scientific applications are typically built from combining multiple MPI modules together. These MPI modules themselves are typically derived from complete MPI applications. Interaction between these modules occurs using function calls into each other, and exhibit the same limitations as interfaces derived from the object models as described in chapter 3. AMPI can, in principle, use the Charisma binding for Charm++ by providing parallel Charm++ wrappers for the AMPI threaded component. However, we need to provide an interface model for AMPI that is consistent with MPI's paradigm, so that adapting legacy MPI codes to Charisma becomes a less-daunting task. AMPI implementation of the Charisma interface model is described in the next section.

5.4 Charisma interfaces in AMPI

Large scientific applications such as coupled simulations are composed from independently written MPI modules. AMPI makes it easier to combine such independently written MPI modules, because each individual module is run inside its own thread group, with it's own `MPI_COMM_WORLD`. Encapsulation of the global data by converting them into thread-private data, and the namespace separation effected by separate communicators allow these MPI modules co-exist within a single application. These module execute concurrently, thus overlapping idle times in one module with useful work in others.

In order to allow these modules to interact with each other, AMPI introduces the notion of cross-communicator point-to-point communication. Thus, any virtual processor in one module may send messages to and receive messages from other virtual processors in other

[2]Karthikeyan Mahesh developed AMPIzer working with me.

modules using the same syntax and semantics as the MPI point-to-point communication subroutines. A concrete example may make this clearer. Suppose an application consists of two modules, A and B, as shown in figure 5.10. Each of these modules first need to register themselves, so that AMPI knows how to invoke each of these modules, and also allocates a communicator for them. This is done by providing a subroutine called MPI_Setup as shown in figure 5.11. The AMPI registration call MPI_Register returns an index for the module, which can be used to look up the "world" communicator (MPI_COMM_WORLD) for that module. These communicators are stored in an indexed communicator array MPI_COMM_UNIVERSE. Thus if a virtual processor from module A needs to communicate with virtual processor 14 in module B, it can send a message to it using an MPI call such as MPI_Send, but specifying MPI_COMM_UNIVERSE[B_Idx] as communicator.

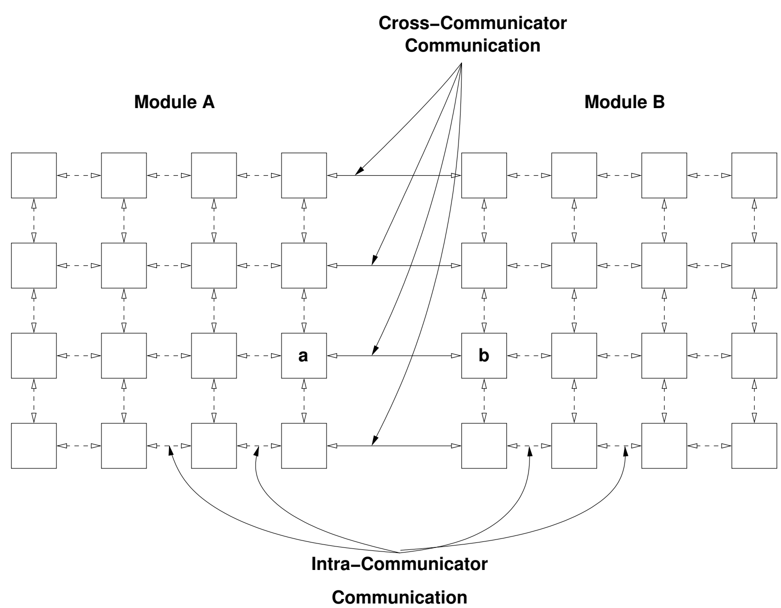


Figure 5.10: Intermodule communication using AMPI cross-communicators.

```
subroutine MPI_Setup
  A_Idx = MPI_Register("A", A_Main)
  B_Idx = MPI_Register("B", B_Main)
end subroutine
```

Figure 5.11: Registration of multiple AMPI modules.

While this technique is suitable for components that are designed to be complementary only to each other, it does not result in truly reusable components, since components have to possess explicit knowledge of other component's decomposition. For example, in figure 5.10, virtual processor a of module A has to know the rank of virtual processor b of module B. If the decomposition of module B changes, say by splitting each original chunk into more chunks, module A will have to change its code to reflect that. This dependence on the other modules is contrary to the Charisma philosophy. However, a small modification to the way components are registered results in the Charisma-style interfaces for AMPI.

In the Charisma model, AMPI components register themselves with the runtime system, and as before, they get a unique `MPI_COMM_WORLD`. However, they also get two additional communicators, to which they can register their input and output ports. These communicators are called `MPI_COMM_INPUT`, and `MPI_COMM_OUTPUT`. At the registration stage, each component specifies its input and output ports to the runtime system using the call `MPI_ADD_PORT`. The parameters to this call are: the name of the port, and the MPI data-type it expects. This call returns the port index, that can later be used to publish data in case of output ports, or to wait for it in case of input ports. The application composer, after each component has been registered, specifies the connection between ports using Charisma port-binding calls. For an application composer written using AMPI, `MPI_BIND` call is made available for this purpose.

When an AMPI component needs to publish data on an output port, it sends a message using `MPI_Send` to an appropriate "pseudo-processor" (port index returned by `MPI_ADD_PORT`) in the communicator `MPI_COMM_OUTPUT`. Similarly, when it requires data on an input port, it makes the `MPI_Recv` call on the appropriate port index with the communicator `MPI_COMM_INPUT`. Since the connections between components are made outside of the component in this model, any AMPI component can be developed completely independently, without knowing about other components.

With colleagues at the Center for Simulation of Advanced Rockets (CSAR), we have converted some large MPI applications using this approach. The techniques used, efforts involved, and preliminary performance data are given in the next section.

5.5 AMPI Performance

We have compared AMPI with the original message-driven multi-partitioning approach to evaluate overheads associated with each of them using a Computational Fluid Dynamics (CFD) kernel that performs Jacobi relaxation on large grids (where each partition contains 1000 grid points.) We ran this application on a single 250 MHz MIPS R10000 processor of the Origin2000 machine at National Center for Supercomputing Applications (NCSA), with different number of chunks, scaling the mesh to keep the chunk-size constant. Two different decompositions, 1-D and 3-D, were used. These decompositions vary in number of context-switches (blocking receives) per chunk. While the 1-D chunks have 2 blocking receive calls per chunk per iteration, the 3-D chunks have 6 blocking receive calls per chunk per iteration. However, in both cases, on the average only half of these calls actually block waiting for data, resulting in 1 and 3 context switches per chunk per iteration respectively. As can be seen from figure 5.12, the optimization due to availability of local variables across blocking calls, as well as larger subroutines in the AMPI version neutralizes thread context-switching overheads for a reasonable number of chunks per processor. Thus, thread-based Adaptive MPI can be effectively used for component coordination without incurring any significant overheads.

Encouraged by these results, we converted some large MPI applications using AMPI as part of the Center for Simulation of Advanced Rockets (CSAR). At CSAR, we are developing a detailed multi-physics rocket simulation and virtual prototyping tool [31]. GEN1, a first generation integrated rocket simulation code is composed of three coupled modules: *Rocflo* (an explicit fluid dynamics code), *Rocsolid* (an implicit structural simulation code), and *Rocface* (a parallel interface between *Rocflo* and *Rocsolid*) [63]. *Rocflo* and *Rocsolid* were

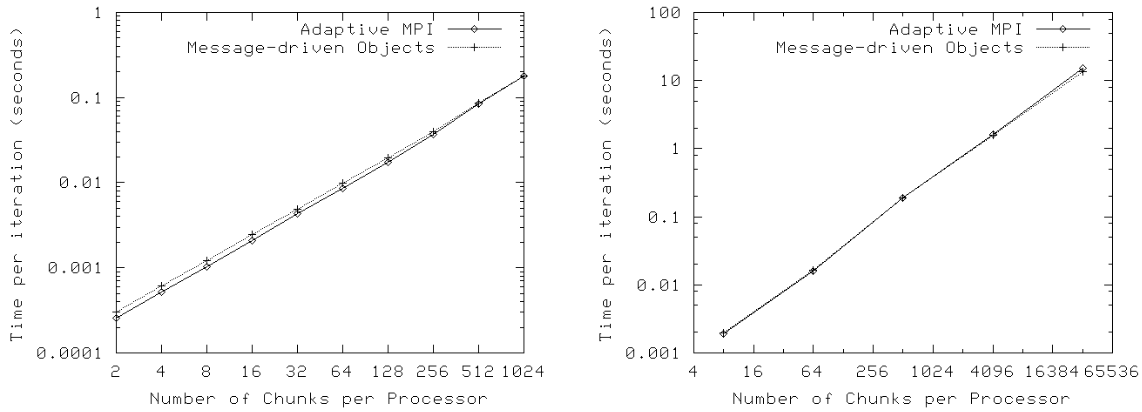


Figure 5.12: The throughput (number of iterations per second) for a Jacobi relaxation application. (Left) with 1-D decomposition. (Right) with 3-D decomposition.

written using FORTRAN 90 (about 10000 and 12000 lines respectively), and use MPI as parallel programming environment. *Rocflo* and *Rocsolid* were the the first application codes to be converted to AMPI[3]. This conversion, using the techniques described in the last section, resulted in very few changes to original code (in fact, the changed codes can be linked with MPI, without any changes), and did not take much time for us, even as we were unfamiliar with the codes (about a week for one person for each of these codes). Conversion of Rocface was even quicker[4]. This quick manual conversion was possible because *Rocface* was a modularly written code written in C++ with very few global variables.

The overhead of using AMPI instead of MPI is shown (tables 5.1 and 5.2) to be minimal, even with the original decomposition of one partition per processor. We expect the performance of AMPI to be better when multiple partitions are mapped per processor, as depicted in Figure 5.1. Also, the ability of AMPI to respond to dynamic load variations outweighs these overheads.

Figure 5.13 shows the performance of the integrated rocket simulation code implemented with AMPI-based components, and its comparison with the original MPI-based code. Setup

[3]Eric deSturler and Jay Hoeflinger, our colleagues at CSAR, converted *Rocflo* to AMPI, while I converted *Rocsolid* along with a research assistant.

[4]Working with Jim Jiao, the original developer of *Rocface*, I converted it to AMPI in 45 minutes.

No. of Processors	*Rocflo* MPI(sec.)	*Rocflo* AMPI(sec.)
1	1637.55	1679.91
2	957.94	916.73
4	450.13	437.64
8	234.90	278.93
16	142.49	126.59
32	61.21	63.82

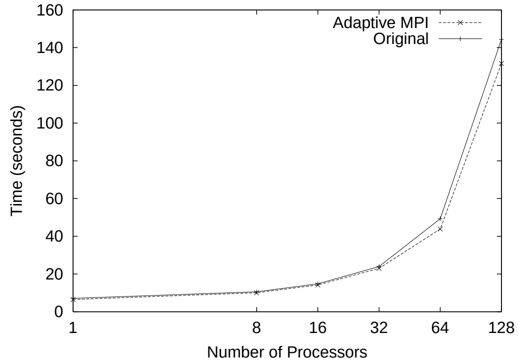
Table 5.1: Comparison of MPI and AMPI versions of *Rocflo*.

No. of Processors	*Rocsolid* MPI(sec.)	*Rocsolid* AMPI(sec.)
1	67.19	63.42
8	69.81	71.09
32	70.70	69.99
64	73.94	75.47

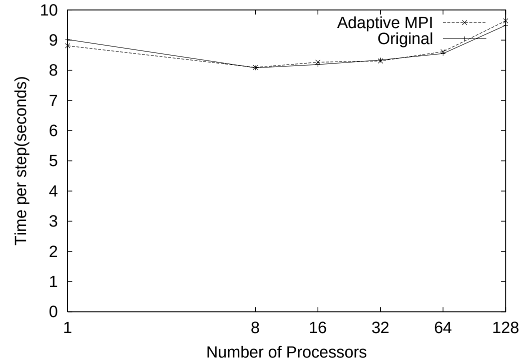
Table 5.2: Comparison of MPI and AMPI versions of *Rocsolid*. Note that this is a *scaled* problem.

stage is carried out once at startup (and exhibits a serial bottleneck at the file-system.) Each timestep of the integrated code consists of fluids update, solids update, and a predictor-corrector step. Timings of the AMPI component-based code are comparable with the original MPI-based codes, with the maximum 3% overhead of AMPI. These experiments were carried out on the NCSA Origin2000 (250MHz MIPS R10000 processors). The AMPI implementation was deliberately chosen to run on the version of Converse that used MPI as its underlying communication library. Also, for this comparison, only one virtual processor of the AMPI version of the rocket simulation code was mapped to each physical processor. Note that this is scaled problem, where the problem size grows with the number of processors.

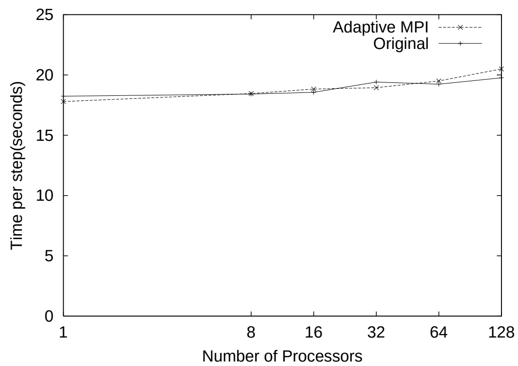
Workstation clusters are often built incrementally, with individual machines that typically reflect technology improvements over the time it took to build a cluster. An example of such a cluster is the “Turing” cluster at the Department of Computer Science at University of Illinois. At the time this experiment was performed, it consisted of 208 dual-processor machines, ranging in architecture from 400 MHz Pentium II to 1 GHz Pentium III, connected with the Myrinet network. Synchronizations and near-neighbor communications make the



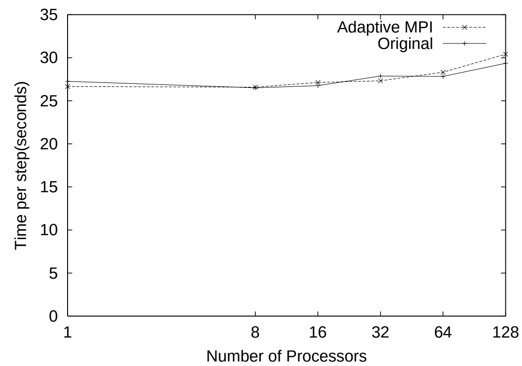
(a) Setup



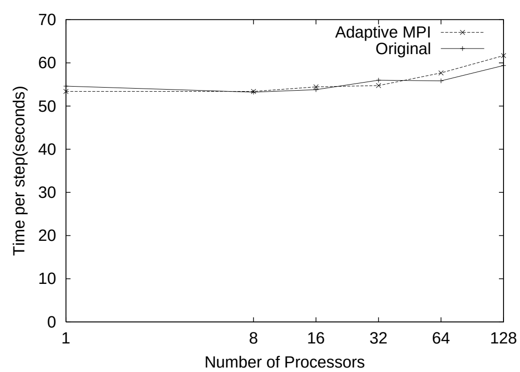
(b) Fluids Update



(c) Solids Update



(d) Predictor-Corrector



(e) Total

Figure 5.13: Comparison of AMPI with native MPI.

faster processors wait for the slower processors to finish work. Therefore, for all except the most trivial parallel applications, the slowest processor dictates the time taken on such het-

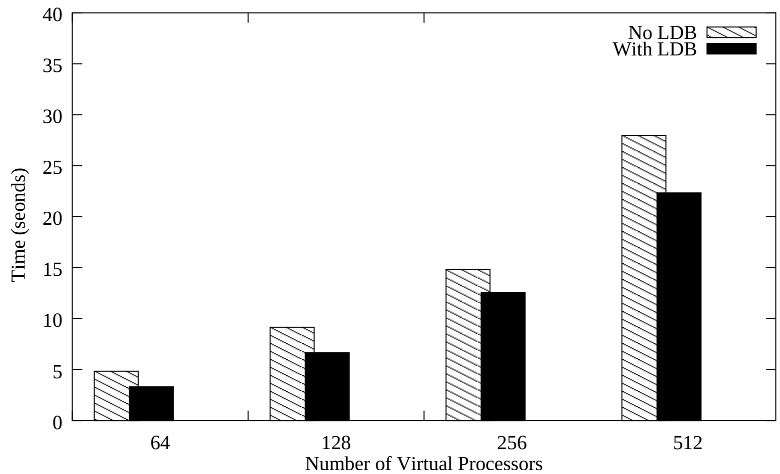


Figure 5.14: AMPI adaptivity on a heterogeneous cluster.

erogeneous clusters. Because of the virtualization strategy employed by AMPI, the runtime system has the freedom to balance the load, taking into account the differences in processor speeds. This is demonstrated in figure 5.14. We ran a Jacobi relaxation application written using AMPI on 62 processors of the Turing cluster. The set of processors available to us had the following composition: 7 Pentium II (400 MHz), 17 Pentium II (450 MHz), 36 Pentium III (550 MHz) and 2 Pentium III (1 GHz). We varied the number of virtual processors from 64 to 512, scaling the mesh so that the individual partition-size was kept unchanged, and compared the timings for each timestep with and without using the processor speeds information. Indeed, we see a significant improvement in performance when processor speeds are taken into account. As the number of virtual processors increase, AMPI has more flexibility in mapping them to available physical processors to balance the load, resulting in greater performance improvements.

Figure 5.15 shows the performance improvements for the various components of the GEN1 codes when running on the heterogeneous Turing cluster.

The dynamic load balancing capabilities of AMPI result in improved utilization of dynamic computing platforms such as workstation clusters, where availability of computational resources may change with time. Figure 5.16 shows the conjugate gradient solver described above initially running on 16 processors in a workstation cluster. As 16 new processors

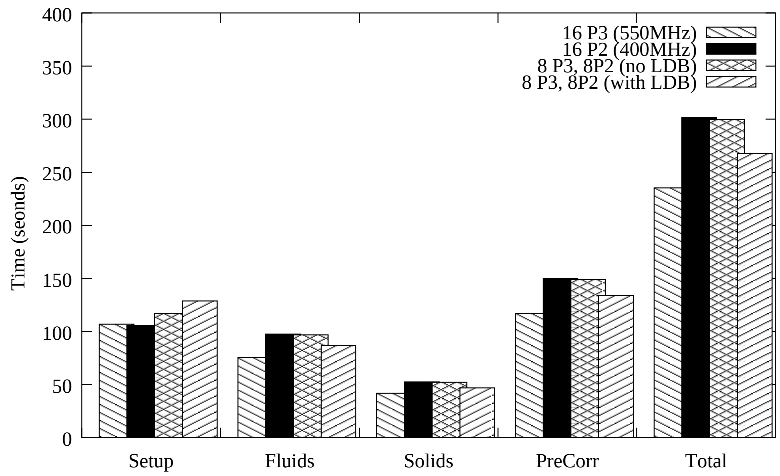


Figure 5.15: CSAR integrated rocket simulation code on heterogeneous cluster.

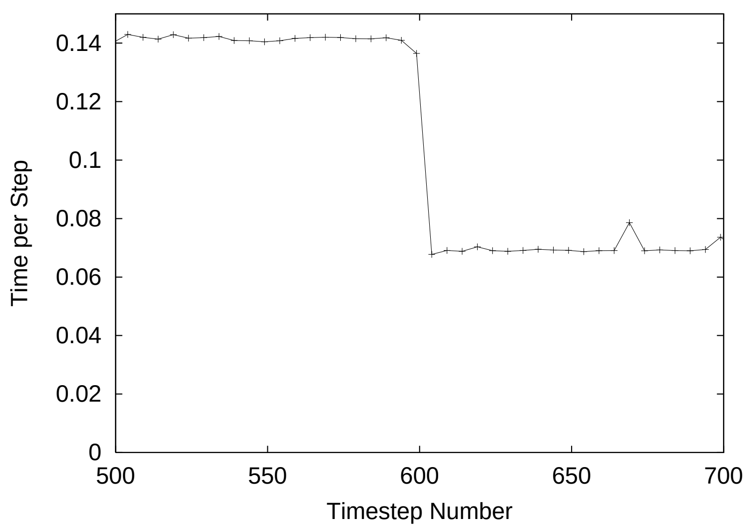


Figure 5.16: AMPI components respond to changing machine availability at runtime.

become available after timestep 600, AMPI redistributes load to all available 32 processors, and the time required for each timestep of the conjugate gradient solver reduces to about half (as expected) of the earlier time. Such flexible allocation of processor resources is possible using an adaptive job scheduler developed by Sameer Kumar and Jay Desouza [48]. This experiment was carried out on an IA-32 based "Platinum" cluster (Pentium III 1 GHz, Myrinet) at NCSA[5].

[5]This experiment was carried out by Sameer Kumar with the Conjugate Gradient Solver code provided by Orion Lawlor.

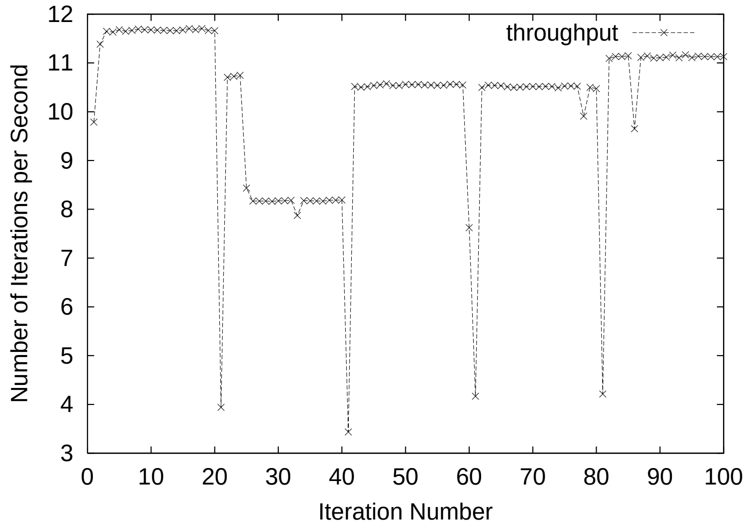


Figure 5.17: Dynamic load balancing in irregular and dynamic AMPI components.

Dynamic and/or irregular applications cause load imbalance at runtime even on homogeneous computing platforms with constant availability. Examples of such problems include scientific applications that use techniques such as adaptive mesh refinement. The measurement-based load balancing framework in Charisma deals with such dynamic applications by redistributing workload among available processors in order to balance the load as shown in figure 5.17. This figure shows the performance of a neighborhood averaging component in AMPI based on Jacobi-relaxation on eight processors of SGI Origin 2000 at NCSA. The grid is partitioned into 64 partitions, that are mapped to eight processors by the Charisma runtime. In the 25^{th} iteration, one of the partitions is refined eight times, thus increasing the computational load of that partition in proportion. Even though the computation of the entire application increases only by 11%, the throughput of the application is reduced by more than 30% due to this load imbalance, since the neighboring partitions are idle waiting for the overloaded partition. The load balancing framework of Charisma is activated every 20 iterations, which detects this load imbalance and moves some partitions from the heavily loaded processors to the lightly loaded processors, bringing the throughput to the desired levels.

Chapter 6

Conclusion and Future Work

Efficient and scalable integration of independently developed parallel software components into a single application requires a component architecture that supports "in-process" components. In this thesis, we described Charisma, a component architecture for parallel in-process components. Charisma has Converse, an interoperable parallel runtime system based on message-driven execution, at its core. Converse provides data-driven control transfer among components, thus supporting multi-paradigm interoperability and efficient parallel composition of modules. We have demonstrated utility of Converse by building threaded, message-driven, and message-passing languages on top of Converse, thus enabling components written using these paradigms to coexist within a single application as in-process components.

Message-driven execution, along with encapsulation and object virtualization provided by a message-driven object language, Charm++, allows us to build efficient software components that are interoperable. The common interface model of Charisma component architecture allows these components to interact with each other. We observed that traditional interface models based on the object model exhibit several weaknesses, especially for in-process components, where efficiency is of paramount importance. We developed an interface model for Charisma that promotes truly independent development of components. Charisma component interfaces are contracts between the components and the runtime system, rather than between components. Rather than invoking services provided by other components, a

Charisma component supplies the runtime system with the data it requires for computations and the data it publishes as a result. These are called input and output ports of a Charisma components. Connections between input ports of one component and the output ports of another are specified (either using Charisma API or the Charisma scripting language) outside of both the components. This, coupled with data-driven control transfer in Converse makes it possible to build reusable components that can be flexibly and efficiently used for application composition.

Although message-driven programming facilitates efficiency and modularity for independent component development, we noted a limitation of message-driven programming systems that obfuscates expression of control-flow within a software component, increasing programming complexity. We have developed a notation called Structured Dagger based on guarded computations (when-blocks), that allows clear expression of control-flow within message-driven components, without incurring additional overheads such as those observed with threaded components.

We have provided support for components based on legacy parallel message-passing codes in Charisma. For this purpose, we have developed Adaptive MPI, an implementation of MPI using Converse's user-level threads. In this thesis, we have described the mechanisms used by AMPI, and evaluated its performance. We have discussed how legacy MPI codes can be converted to AMPI, and can be made into reusable components by providing Charisma interfaces in the message-passing paradigm.

6.1 Future Work: Adaptive MPI

The AMPI implementation currently has over 70 commonly used functions from the MPI 1.1 standard. It needs be made fully standard compliant. Communicator-related functionality could be implemented with array sections in Charm++. Topology-related functions of MPI could then be implemented more efficiently on top of this new implementation of communicators.

We have demonstrated that for real applications, AMPI overhead is compensated for by several advantages of AMPI over MPI. However, we believe that the AMPI overhead itself can be further reduced, especially in collective communications. In general, further work is needed for optimizing collective communications in the presence of object virtualization. Work is currently undergoing for building Converse-level optimized communication routines, and we expect it to be beneficial to AMPI.

On 32-bit processors, the unused address space that is used for isomalloc'ed threads can be limited sometimes depending upon the heap size and stack usage. While this problem becomes irrelevant on the new 64-bit processors, support for clusters based on Intel IA-32 chips, as well as the new ASCI class BG/L machine from IBM (that uses a 32-bit processor) is crucial. For this purpose, one can implement isomalloc'ed threads by limiting their migratability. This leads to fewer divisions of the unused address space, and therefore more availability of this address space on each processor.

6.2 Future Work: Charisma

Charisma is the newest addition to the Converse-based suite of parallel programming paradigms. There are a number of ways Charisma could be extended and become more mature.

6.2.1 Runtime Optimizations

Several new runtime optimizations become feasible with Charisma interface model because the composition and connectivity information is explicitly available to the runtime system. In the future, we plan to study and evaluate such optimizations. A few of them are illustrated here.

Consider an example in molecular dynamics (section 2.6), where each pair of patches has a compute object associated with it, which is responsible for computing pairwise cutoff-based interactions among atoms in those patches. If the patch neighborhood information is specified using our interface model (by constructing a 3-D grid of patches with the interface language), these patches could be placed automatically by the runtime system on the available set of processors by taking locality of communication into account. Further, specification of connections between patches and compute object would allow the runtime system to place the compute objects closer to the patches they connect. This, combined with the information about communication volume available from the Converse load balancing framework, would allow the runtime system to place a compute object on the same processor as the patch that sends more atoms to it. We propose to provide such runtime optimization techniques by incorporating the connectivity information in the Converse load balancing framework.

Connection specification also enables the runtime system to optimize data exchange between components that belong to the same address space. This can be achieved by allowing the components to publish their internal data buffers to the output ports. In the normal course, the published data will be sent as a message directed at the input port of the connected component. If the connected component belongs to the same address space, the runtime system may pass the same buffer to the input port of the connected component in the same process, when the corresponding input port declares itself to be *readonly*.

6.2.2 Dynamic Loading

In the current prototype implementation of Charisma, the application composition is performed at link-time. However, this may be restrictive for applications that need component substitution at run-time. This limitation of Charisma can be removed by supporting dynamic loading of components at run time. Each component binary may be in the form of a shared object ("so" in Unix, "DLL" on Windows). These components may be registered in a component repository that the application composer knows about. Component interfaces for registration, creation, and port-bindings could then be invoked by loading the component at runtime, and invoking appropriate functions in the loaded shared object by specifying their names. Also, we believe that dynamic loading will cleanly solve the namespace conflicts between components. This is crucial for increased reusability of components.

6.2.3 Substitution

With the support for dynamic loading of components, one could also standardize unloading of components. A component may have standardized interfaces for un-binding its ports, saving their state in persistent storage, and unloading them. When a component un-binds its ports, Charisma should substitute them by dummy ports that buffer all the incoming data, instead of discarding them. This would make sure that other components connected to the substituted components do not have to be aware of a connected component being substituted. Also, each component should support a new interface for starting from a saved state. Compatibility of components available for substitution may be determined by compatibility of the saved state. We expect that this will be useful for upgrading to newer component versions available in addition to substituting new functionality.

6.2.4 Reflection

For the application composer, a component should provide descriptions of the services that it provides, in addition to the interfaces available for gluing together components. This is an important factor in component reuse and distribution. While external documentation may be sufficient in some cases, we believe that the best place for component documentation is in the component itself. This makes the component truly self-contained. Thus, when the component registers itself with the component repository, it will also register the interfaces for querying the documentation. This will make it possible for the application composer to make queries to the repository such as "Show me components that implement conjugate gradient solver".

6.2.5 Performance Characteristics

Component reuse depends not only on the functionality they implement, but also the performance characteristics they exhibit. For example, the same functionality of solvers may be implemented in a variety of ways, with differing performance profiles over a range of parameters. Some may be more efficient for small-sized matrices, while some may be faster for large matrices. Some may be optimized for small number of processors, while some may be designed to be scalable to large number of processors. Some may perform better on shared memory machines than in distributed memory machines. In the Faucets project [48], we are working on characterizing performance profiles for applications. Similar performance profiling could be done for components, and this profile could be stored and continuously updated in the component repositories. This will enable the application composer to choose from among components implementing the same functionality based on their performance profiles.

6.2.6 Component Frameworks

A component architecture's success is measured in terms of the number of applications built using it, and the number of reusable components available for application composition. We believe that the ease of development of components and the efficiency of integration of components are crucial for a component architecture to be successful. One way to achieve this is by building component frameworks on top of the component architecture. We have several ongoing efforts in building frameworks for scientific applications, such as the Finite Element Method (FEM) framework [10], a multi-block framework, an Adaptive Mesh Refinement (AMR) framework [60] etc. These frameworks have been written on top of Charm++. We have demonstrated that application development is simplified using these frameworks. These frameworks will be retargeted on top of Charisma. Thus application components built using these frameworks will become available as reusable Charisma components.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] Gul Agha. Compositional development from reusable components requires connectors for managing both protocols and resources. In *Workshop on Compositional Software Architectures*, Monterey, California, January 1998.

[3] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the pm2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico, Held in conjunction with the 13th Intl Parallel Processing Symp. (IPPS/SPDP 1999), IEEE/ACM. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[4] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. Ch. Eigler, and G.R. Gao. ABC++: Concurrency by Inheritence in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[5] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, pages 115–124, Redondo Beach, California, August 1999.

[6] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders,

and M. Tholburn. Pooma: A high performance distributed simulation environment for scientific applications. In *Proceedings Supercomputing '95*, 1995.

[7] P. Beckman and D. Gannon. Tulip: A portable run-time system for object-parallel systems. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[8] R. F. Belanger. MODSIM II - A Modular, Object-Oriented Language. In *Proceedings of the Winter Simulation Conference*, pages 118–122, 1990.

[9] Milind Bhandarkar and L. V. Kale. MICE: A Prototype MPI Implementation in Converse Environment. In *Proceedings of the second MPI Developers Conference*, pages 26–31, South Bend, Indiana, July 1996.

[10] Milind Bhandarkar and L. V. Kalé. A parallel framework for explicit fem. In *Proceedings of the International Conference on High Performance Computing*, Bangalore, India, December 2000.

[11] Milind Bhandarkar and L. V. Kale. An Interface Model for Parallel Components. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC), Cumberland Falls, KY*, August 2001.

[12] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. Technical Report 00-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, September 2000.

[13] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In *Proceedings of 7th International Workshop on Languages and Compilers for Parallel Computing*,

number 892 in Lecture Notes in Computer Science, pages 141–154, Ithaca, NY, USA, August 1994. Springer-Verlag.

[14] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet—A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[15] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

[16] Robert Brunner. *Versatile Automatic Load Balancing With Migratable Objects*. PhD thesis, University of Illinois at Urbana-Champaign, January 2000.

[17] K. G. Budge and J. S. Peery. Experiences developing alegra: A c++ coupled physics framework. In M.E Henderson, C. R. Anderson, and S. L. Lyons, editors, *Object oriented methods for interoperable scientific and engineering computing, proceedings of the 1998 SIAM workshop*, October 1998.

[18] K.M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-oriented Programming Notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 281–313. MIT Press, 1993. ISBN 0-272-01139-5.

[19] Nikos Chrisochoides, Induprakas Kodukula, and Keshav Pingali. Data movement and control substrate for parallel scientific computing. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 256–268, 1997.

[20] C.Tomlinson and V.Singh. Inheritance and synchronization with enabled-sets. In *ACM OOPSLA*, pages 103–112, 1989.

[21] Noah Elliott, Scott Kohn, and Brent Smolinski. Language interoperability mechanisms for high-performance scientific components. In *International Symposium on Comput-*

ing in Object-Oriented Parallel Environments (ISCOPE 99), San Francisco, CA, USA, September-October 1999.

[22] Common Component Architecture Forum. Cca forum home page. See `http://www.cca-forum.org/`.

[23] I. Foster and C. Kesselman (Eds). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[24] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[25] Ian Foster. Compositional parallel programming languages. *ACM Transactions of Programming Languages and Systems*, 18(4):454–476, 1996.

[26] L. A. Freitag, W. D. Gropp, P.D. Hovland, L. C. McInnes, and B. F. Smith. Infrastructure and interfaces for large-scale, numerical software. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June-July 1999.

[27] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[28] Attila Gursoy. *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance*. PhD thesis, University of Illinois at Urbana-Champaign, June 1994. Also, Technical Report UIUCDCS-R-94-1852.

[29] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings of Supercomputing 1994*, Washington D.C., November 1994.

[30] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[31] M. T. Heath and W. A. Dick. Virtual rocketry: Rocket science meets computer science. *IEEE Comptational Science and Engineering*, 5(1):16–26, 1998.

[32] William D. Henshaw, D. L. Brown, and Daniel J. Quinlan. Overture: An object-oriented framework for solving partial differential equations on overlapping grids. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 58–67. SIAM, 1998.

[33] High Performance Fortran Forum. *High Performance Fortran Language Specification (Draft)*, 1.0 edition, January 1993.

[34] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine independent parallel programming in fortran-d. In J. Salz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier Science Publishers B.V., 1992.

[35] Richard Hornung and Scott Kohn. The use of object-oriented design patterns in the SAMRAI structured AMR framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998.

[36] Draft Standard for Information Technology—Portable Operating Systems Interface (Posix), September 1994.

[37] Xiangmin Jiao. Roccom home page. See `http://www.cse.uiuc.edu/ jiao/roccom.html`.

[38] L. V. Kalé, M. Bhandarkar, R. Brunner, N. Krawetz, J. Phillips, and A. Shinozaki. Namd: A case study in multilingual parallel programming. In Zhiyuan Li, Pen-Chung Yew, Siddharta Chatterjee, Chua-Huang Huang, P. Sadayappan, and David Sehr, editors, *Languages and Compilers for Parallel Computing*, number 1366 in Lecture Notes in Computer Science, pages 367–381. Springer-Verlag, 1998.

[39] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[40] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

[41] L. V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[42] L. V. Kalé, Milind Bhandarkar, and Terry Wilmarth. Design and implementation of parallel java with a global object space. In *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, pages 235–244, Las Vegas, Nevada, July 1997.

[43] L. V. Kalé and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proc. 27th Conference on Parallel Processing for Scientific Computing*, pages 738–743, February 1995.

[44] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[45] L. V. Kale and Joshua Yelon. Threads for Interoperable Parallel Programming. In *Proc. 9th Conference on Languages and Compilers for Parallel Computers*, San Jose, California, August 1996.

[46] L. V. Kale, Joshua M. Yelon, and T. Knauff. Parallel import report. Technical Report 95-16, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1995.

[47] Laxmikant Kalé, Robert Skeel, Milind Bhandarkar, Robert Brunner, Attila Gursoy, Neal Krawetz, James Phillips, Aritomo Shinozaki, Krishnan Varadarajan, and Klaus Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[48] Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[49] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 17–25, August 1990.

[50] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[51] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. TR 95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55414, May 1995.

[52] George Karypis and Vipin Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. In *Proceedings of Supercomputing '96*, Pittsburg, PA, November 1996.

[53] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington Department of Computer Science and Engineering, May 1993.

[54] E. Kornkven and L. V. Kalé. Efficient implementation of high performance fortran via adaptive scheduling – an overview. In *Proceedings of the International Workshop on Parallel Processing*, Bangalore, India, December 1994.

[55] Orion Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of the International Symposium on Computing in Object-oriented Parallel Environments*, June 2001.

[56] Karthikeyan Mahesh. Ampizer: An mpi-ampi translator. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champiagn, 2001.

[57] Mathematics and Argonne National Laboratory Computer Science Division. Alice web page. See `http://www.mcs.anl.gov/alice`.

[58] Richard Monson-Haefel. *Enterprise Javabeans*. O'Reilly and Associates, 2000.

[59] F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *USENIX Winter Conference*, pages 29–41, 1993.

[60] Puneet Narula. An adaptive mesh refinement (AMR) library using charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[61] Mark Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant Kale, Robert D. Skeel, and Klaus Schulten. NAMD—a parallel, object-oriented molecular dynamics program. *Intl. J. Supercomput. Applics. High Performance Computing*, 10(4):251–268, Winter 1996.

[62] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois fast messages (FM) for myrinet. In *Proceedings of Supercomputing 1995*, dec 1995.

[63] I. D. Parsons, P. V. S. Alavilli, A. Namazifard, J. Hales, A. Acharya, F. Najjar, D. Tafti, and X. Jiao. Loosely coupled simulation of solid rocket moters. In *Fifth National Congress on Computational Mechanics*, Boulder, Colorado, August 1999.

[64] Alan Pope. *The Corba Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1998.

[65] Parthasarathy Ramachandran and L. V. Kalé. Mulitlingual debugging support for data-driven and thread-based parallel languages. Technical Report 99-04, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1999. To appear in the Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC '99).

[66] Parthasarathy Ramachandran and L. V. Kalé. Web-based interaction and monitoring for parallel programs. Technical Report 99-05, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[67] W. Rankin and J. Board. A portable distributed implementation of the parallel multipole tree algorithm. *IEEE Symposium on High Performance Distributed Computing*, 1995. [Duke University Technical Report 95-002].

[68] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[69] A. Sinha and L.V. Kalé. Information Sharing Mechanisms in Parallel Programs. In H.J. Siegel, editor, *Proceedings of the 8th International Parallel Processing Sym posium*, pages 461–468, April 1994.

[70] A.B. Sinha. *Performance Analysis of Object Based and Message Driven Programs*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, January 1995.

[71] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2, 4:315–339, December 1990.

[72] L. M. Taylor. Sierra - a software framework for developing massively parallel, adaptive, multi-physics, finite element codes. In *Presentation at the International conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada, USA, June 1999.

[73] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An operating system coordinated high performance communication library. In *HPCN Europe*, pages 708–717, 1997.

[74] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[75] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*, August 1995.

[76] Deborah A. Wallach, Wilson Hsieh, Kirk Johnson, M. Frans Kaashoek, and William Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proceedings of the Fifth Symposium on Principles and Practices of Parallel Programming*, pages 217–226, July 1995.

[77] Joshua Yelon. *Static Networks Of Objects As A Tool For Parallel Programming*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1999.

[78] Joshua Yelon and L. V. Kalé. Agents: An undistorted representation of problem structure. In *Lecture Notes in Computer Science*, volume 1033, pages 551–565. Springer-Verlag, August 1995.

Vita

Milind A. Bhandarkar was born in the state of Maharashtra, India. He obtained his high school diploma from the Board of Secondary and Higher Secondary Education, Maharashtra in 1985. He graduated from Birla Institute of Technology and Science, Pilani, India with M.Sc.(Tech) in Computer Science in 1989, and obtained M. Tech. in Computer Science and Engineering from Indian Institute of Technology, Kanpur, India in 1991.

He worked at Center for Development of Advanced Computing, Pune, India as Member Technical Staff in the Parallel Applications Group from 1991 to 1993, before joining State University of New York at Buffalo, NY for his PhD. He transferred to Department of Computer Science at the University of Illinois at Urbana-Champaign, IL in the Fall of 1994, and has been with the Parallel Programming Laboratory (PPL) led by Prof. L. V. Kale since then.

As a member of PPL, he has worked on many aspects of parallel computing, including runtime systems, coordination languages, and component architecture that is presented in his doctoral thesis. From 1996 to 1998, he was a member of Theoretical Biophysics Group (TBG) at the Beckman Institute for Advanced Science and Technology at the University of Illinois. At TBG, he was one of the primary designers of a parallel molecular dynamics simulation application, NAMD. From 1998, he has been a Research Programmer, with the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois. While at CSAR, he has worked on implementing several engineering simulation applications, and development of frameworks for engineering applications. After completion of his PhD, Milind will be joining Siebel Systems, Inc. as Software Engineer.