The Charm Parallel Programming Language and System: Part II - The Runtime System*

B. Ramkumar* **A. B. Sinha**** **V. A. Saletore***** **L. V. Kalé****

*Dept. of Electrical and Computer Eng.
University of Iowa
Iowa City, Iowa 52242
ramkumar@eng.uiowa.edu

**Dept. of Computer Science
University of Illinois
Urbana, Illinois 61801
{sinha,kale}@cs.uiuc.edu

***Dept. of Computer Science
Oregon State University
Corvallis, Oregon 97331
saletore@mist.ogst.edu

Abstract

Charm is a parallel programming system that permits users to write portable parallel programs on MIMD multiprocessors without losing efficiency. It supports an explicitly parallel language which helps control the complexity of parallel program design by imposing a separation of concerns between the user program and the system. It also provides target machine independent abstractions for information sharing which are implemented differently on different types of processors. In part I of this paper [16], we described the language support provided by Charm and the rationale behind its design.

Charm has been implemented on a variety of parallel machines including shared memory machines like the Encore Multimax and the Sequent Symmetry, message passing architectures like the Intel iPSC/2, Intel i860 and the NCUBE 2, and a network of Unix workstations. The *Chare kernel* is the run-time system that supports the portable execution of Charm on several MIMD architectures. We discuss the implementation and performance of the *Chare kernel* on three architectures: shared memory, message passing, and a network of workstations.

Index terms: Message-driven execution, MIMD machines, Parallel programming, Portable parallel software, Task granularity.

*This research was supported in part by the National Science Foundation grants CCR-90-07195 and CCR-91-06608. Dr. Ramkumar's work is supported in part by the National Science Foundation grant NSF-CCR-9308108.

1 Introduction

The increasing availability of parallel machines has raised expectations of faster execution times, better response times and improved productivity. This has largely remained unfulfilled, beyond a narrow class of applications, for two important reasons. The first reason is the complexity of the parallel software development process. Parallel algorithm design involves consideration of issues not present in sequential algorithm design, notably deadlock avoidance, synchronization, mutual exclusion, load balancing, scheduling, idling due to latency, etc. These make parallel programming considerably more complicated than sequential programming. The second reason is the wide range of performance characteristics and programming interfaces exhibited by the different parallel computers available today. In Part I of this paper, we introduced Charm as a language that helps the user control the complexity of parallel programming by satisfying five requirements:

- *High level:* Charm provides high-level parallel programming support, including portability across MIMD architectures, and automatic mapping and scheduling.
- *General purpose:* In addition to being high-level, Charm is designed to be a minimal abstraction over different machine architectures, so that it will be useful both for end users writing parallel applications and for implementors designing parallel languages.
- *Efficiency:* Features, such as latency tolerance through message driven execution, allow Charm to provide an efficient method for programming parallel machines.
- *Expressiveness:* Charm allows dynamic creation of tasks and provides specific modes of information sharing. These increase the expressiveness of Charm.
- *Modularity:* Charm provides an efficient and convenient way of modularly writing parallel programs, which is often not possible using other contemporary languages.

The features of this language have been chosen so as to lend themselves to efficient implementations on the different architectures. Thus, a uniform language interface is presented to a parallel programmer – the parallel program need only be written once and it will run *unchanged* and efficiently on all the target architectures on which Charm has been implemented. In this paper, we

describe the implementation and performance of *Charm's* run-time system — the *Chare kernel*. The Chare kernel is a runtime system that supports *portable*, object based, and message driven parallel programming across MIMD architectures [1].

We discuss issues involved in realizing the features of Charm described in part I [16] of this paper on three classes of target architectures: shared memory machines, message passing machines and networks of workstations. Charm currently runs on shared memory machines like the Sequent Symmetry, the Encore Multimax and the Alliant FX/8, message passing machines like the Intel iPSC/2 and i860, Intel Paragon, IBM SP-1, NCUBE-II, and Thinking Machines CM-5. It has also been implemented on networks of UNIX workstations, including a network of Sun Sparc, IBM RS6000 and HP Series 700 workstations. It is currently under implementation on message passing machines such as the Cray T3D, and NUMA machines such as KSR-1 and Convex Exemplar.

The rest of this paper is structured as follows. In Section 2, we present a brief overview of Charm. In Section 3, we present an overview of the design objectives and the structure of the system. In Section 4, we present the implementation of the system core. This includes how the system is initialized and how we support message driven execution model on each of three architectures: shared memory, nonshared memory, and networks of workstations. Section 5 addresses the machine interfaces and specific issues relating to a network of workstations implementation and how they affect performance. In Section 7, we describe how various other language features and linkable strategies, such as dynamic load balancing and queuing, are supported using the core. In Section 8, we outline an approach for designing efficient Charm programs, and illustrate it using some of the large parallel applications that have been developed to date. In Section 9, we describe the performance of the system and high-light some of the important characteristics of Charm programs. We then conclude the paper in Section 10 with a discussion of our experience.

[1] An object-oriented version of Charm, based on *C++* has also been developed [14]. Like Charm, it too uses the Chare kernel for its run-time support.

```
chare ChareName {

   /* Persistent Data (i.e.  Local Variables) */

   entry EP1 :  (message MSGTYPE1 *msg)
      { /* C code block */ }
   ...
   [private|public] Function1(...)
      { /* C code block */ }
}
```

Figure 1: The syntax of a chare.

2 A brief overview of the language

The basic unit of computation in Charm is a *chare* (which is a form of a concurrent object). Figure 1 shows the syntax of a chare. A chare's *definition* consists of an encapsulated data area and entry functions that can access the data area. A chare *instance* can be created dynamically using the *CreateChare* system call. Each chare instance has a unique address. Entry functions in a particular chare instance can be asynchronously invoked by addressing a message to the desired entry function of the chare using the *SendMsg* system call. A chare is a concurrent object [2] and is somewhat like an *actor* [1] although there are some significant differences[16].

The objects in Charm are message-driven. In practical terms, this means that there is no receive call in the language, nor is there any blocking call that depends on processing on a remote processor. When the C-code block in an object is activated, it runs to completion without blocking or interruption. A chare may be resident on only one processor at a time. In addition, for any given chare, at most one of its entry points may be in execution at any point in time.

Charm provides a second type of process called a *branch office chare* (BOC). A copy (branch) of the chare executes on each processor; each branch has a separate data area owned by it. In addition to receiving messages at entry-points on individual branches like chares, BOCs also provide *public functions*. On any processor P, these functions can be called by any chare resident on P. A BOC has some similarities to the *concurrent aggregate* construct [7] independently proposed by Chien and Dally which was designed for fine-grained parallel machines like the J-machine [9] (discussed in part I of this paper). Branch office chares provide a convenient abstraction for the implementation

of various distributed strategies, such as load balancing and support for distributed data structures.

Charm does not permit general-purpose shared variables. Parallel programs often share data in only a few *distinct* and *specific* modes; the 'completely general' shared variable is rarely needed. Further, the completely general shared variable is difficult to implement efficiently on nonshared memory machines. Instead, Charm provides five different kinds of specifically shared variables: *read only*, *write once*, *accumulator*, *monotonic*, and *distributed tables*.

A detailed description of the language features of Charm can be found in part I [16] of this paper. We provide the above brief description only to set the necessary context for the ensuing discussion.

3 Design objectives

In designing the runtime support system for Charm we had three objectives:

- *Quick portability and extensibility*: The Charm language is portable, i.e., user programs written in Charm run unchanged on all machines on which Charm is supported. One of the main considerations in the design for the runtime system was that it be easily portable, too. Essentially, this means that the machine-specific interface be minimal so that porting Charm to a new system is easily done by defining the interface for the new machine.

- *Small core*: Another objective of our design was to keep the 'core' of the runtime system small, and implement language and system features in terms of the core. We identified that the core of the runtime consisted of chares and branch office chares. The various load balancing and queuing strategies and language features, such as quiescence detection and information sharing abstractions are implemented using the features in the core.

- *Efficiency*: The most important objective of our design was to have an efficient implementation. In a few cases, the other two design objectives were compromised in order to have the most efficient implementation for a particular machine configuration.

Figure 2 shows the overall design structure of the Chare Kernel. At the central system core are those portions of the kernel that initialize the system, and implement chares and branch office

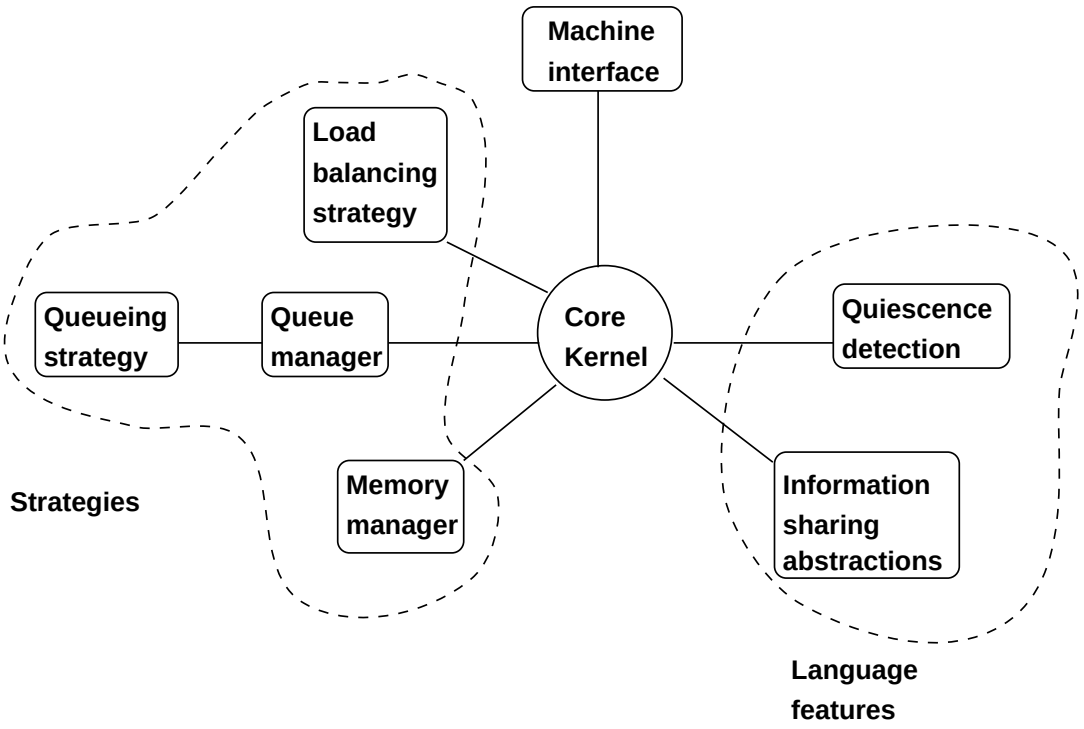


Figure 2: Overall design structure of the Chare Kernel

chares. Various language features and linkable strategies are implemented using the system core. The machine interface is a small generic layer that implements functionality required by the core in terms of the specific machine primitives.

4 The System Kernel

The execution of a Charm program can be broadly classified into two phases: initialization and the message-driven loop. The kernel associates types with messages to improve processing efficiency. The user creates a *new-chare* message when making a $CreateChare()$ request, a *for-chare* message when sending a message to an existing chare using $SendMsg()$, a *new-boc* message when making a $CreateBOC()$ request, and a *for-boc* message when sending a message to an existing branch office using $SendMsgBranch()$. In addition, the kernel supports additional message types for system generated messages. In this section, we describe what happens during initialization and how the system implements message driven execution.

4.1 Initialization

Every Charm program is required to have a *main* chare. This chare is required to have an entry point called *CharmInit*. The execution of the user program begins at the *CharmInit* entry point. Before the system executes the *CharmInit* entry point, it performs system initialization. This consists of initializing the memory manager, allocating internal data structures for queue management, and creating system branch office chares for load balancing and other system functions.

Inside the *CharmInit* entry point, the user typically specifies the creation of different information sharing abstractions, the creation of branch office chares, and the creation of one or more chares, thereby initiating the execution of the Charm program. The requests for the creation of branch office chares and information sharing abstractions are guaranteed to be serviced *before* the the application program begins execution. However, the requests to create chares result in the creation of *seed* messages, which are buffered by the kernel until the initialization of branch office chares and information sharing abstractions is completed. Thus, a chare that executes after *CharmInit* can assume that every specifically shared variable and branch office chare that was created in *CharmInit* has already been installed on every processor in the system. In the initialization phase, the task of the system is to ensure that the dependences and creation-sequences among objects as dictated by the code in *CharmInit* is enforced on all the processors.

The creation of a BOC is requested by using the $CreateBOC()$ primitive. This results in the creation of a *new-boc* message that is broadcast to all processors. Since several of these messages are typically created (many of the system features, including the information sharing abstractions and load balancing are implemented as BOCs), these messages are enqueued in a FIFO queue on each processor during the user-initialization phase.

On shared memory machines, one Unix process (or thread, depending on the available OS support) is then created per processor *after* the user-initialization phase is complete. Each process begins node-specific initialization which includes picking up these messages and creating the BOCs requested. This completes the system-initialization phase on shared memory machines.

On message passing machines, initialization has to allow for the possibility of messages arriving out of order. Since some *new-chare* messages, as well as messages from other branches of BOCs,

may arrive before all the initialization messages (i.e., the messages resulting from the execution of *CharmInit*) are received and processed, they need to be buffered until the processing of all the initialization messages is completed. The system supports a special *init-count* message that notifies each node of the total number of initialization messages that it is *expected* to receive. As initialization messages arrive, they are received and processed; all other messages are buffered. The system maintains two counts — the number of expected initialization messages and the number of received (hence processed) initialization messages. The former is set when the *init-count* message is received. The system initialization phase is complete when the *init-count* message has been received *and* the number of expected and received initialization messages are equal.

4.2 Supporting message driven execution

Once initialization is complete, the system enters the the *pick-and-process* loop (see Figure 3). The *pick-and-process* loop implements message-driven execution. Conceptually, at this point, the runtime support system can be viewed as a work pool "manager". It manages a pool of messages representing seeds for new chares (referred to as *new-chare* messages) or messages to existing chares (referred to as *for-chare* messages). Each message is destined for a named entry point in the program code. Processing a message involves executing the code associated with the entry point sequentially without interruption. Once a message is processed, control returns to the kernel. Thus, the kernel is in a *pick-and-process* loop, constantly picking messages from the work pool and processing them one by one. The kernel exits from this loop only when global termination is detected. The order in which messages are picked up for processing is determined by the queue manager and the queuing strategy chosen by the user. This may be a simple LIFO or FIFO order, or may be based on priorities that the user is permitted to assign to messages.

On *shared memory* machines, the work pool is implemented using a collection of shared queues. Messages that have a known destination (e.g. *for-chare* or *for-boc* messages) are placed in a processor-specific queue. There is one such queue for each processor in use. *New-chare* messages that do not have the destination determined by the user are placed in a single common queue. Although a single shared queue can potentially increase contention, the small number of processors typically available on shared memory systems bounds this problem. Moreover, this minor disadvantage is

Pick-And-Process() `/* for shared memory machines */`

```
Initialize-System-Bocs();
Initialization();
while (not System-Done)
     TimerChecks(); /* Boc function call */
     PeriodicChecks(); /* Boc function call */
     PickNextMessage(&message); /* Boc function call to queue manager*/
     if (message == NULL)
          CheckForQuiescence; /* Boc function call */
     else ProcessMessage(message);
```

Pick-And-Process() `/* for nonshared memory machines */`

```
Initialize-System-Bocs();
Initialization();
while (not System-Done)
     message = NULL;
     while (message == NULL)
          TimerChecks(); /* Boc function call */
          PeriodicChecks() /* Boc function call */
          PumpMessages(); /* receive and enqueue all available messages */
          PickNextMessage(&message); /* Boc function call */
     ConditionallyUnpack(&message);
     ProcessMessage(message);
```

ConditionallyUnpack(message) `/* for nonshared memory */`

```
if (message->packid != NULL_PACK_ID)
     (UnpackTable[message->packid])(&message);
```

Figure 3: The "pick-and-process" loop of the Chare kernel.

offset by the significant improvements achieved in load-distribution using a single queue.

To minimize contention on shared memory systems further, on any processor P, the *pick-and-process* loop attempts to pick messages from the common shared queue only when there is no message in P's processor-specific queue. This has been empirically found to provide negligible contention for a wide variety of programs [26].

This strategy also helps with memory utilization and load balancing. Processing new chare messages before for-chare messages would lead to an accumulation of for-chare messages proportional to the creation tree of chares, which would lead to exponential memory utilization in many application classes, such as divide and conquer. Also new chare messages represent pieces of work that can be mapped arbitrarily onto any processor, whereas other messages must be processed on the processor

on which the chare they are directed to exists. The dynamic load balancing strategy, to be effective, would need to have a good supply of freely mappable work, and hence, selecting the new chare messages only after no other messages are available is desirable.

Note that the queue manager implements the strategy concerning the number of queues in which to divide the set of available messages and the sequence in which to process them. A different queue manager implementing a different strategy than above can easily be selected (or implemented) by the user if it is desirable for the application.

On *nonshared memory* machines, the work pool implementation is different. Unlike shared memory machines, it is necessary to explicitly send and receive messages exchanged between processors. Incoming messages need to be periodically received by every node and then inserted into the local work pool. Clearly, no sharing of queues is possible. Each processor has its own local queue. This necessitates periodic load balancing to distribute messages across different local queues. We discuss how load balancing is accomplished in Section 7.2.

Each iteration of the *pick-and-process* loop does the following on each node P of a distributed memory machine: All messages sent by other processors that have already arrived at P are received by the kernel. User-messages are inserted into the local queue. System messages that relate to load-balancing, updates of information sharing abstractions (See Section 7.1), and quiescence detection (See Section 7.4) are processed immediately upon receipt at the destination processor. Once this is done, the next available message is then picked up from the local queue for processing. If the work pool is empty, the node tries to 'pump' messages repeatedly from the network until a message in the work pool is available for processing.

4.3 How is a message processed?

A message picked from the work pool can be one of the following: a *new-chare* message, a *for-chare* message, or a *for-Boc* message. All these messages have entry-points associated with them. In addition, the *for-chare* and *for-boc* messages also carry the ID of the chare or BOC they are destined for. The *new-chare* messages, when processed, result in the creation of a new chare instance.

```
ProcessMessage(message)
    switch (message->type)
    case new-chare:
        DataArea = Allocate-Chare-Data-Area(message->datasize);
        (EntryPointTable[message->InitEP])(message, DataArea);
    case for-chare:
        DataArea = Get-Chare-Data-Area(message);
        (EntryPointTable[message->ForEP])(message, DataArea);
    case for-boc:
        DataArea = BocDataTable[message->BocNum];
        (BocEPTable[message->ForEP])(message, DataArea);
    case terminate:
        CkPrintStatistics();
        System-Done = TRUE;
```

Figure 4: The *ProcessMessage* function of the Chare kernel.

In addition, one special message type is generated by the system during system termination: a *terminate* message which initiates the global termination protocol. This message is generated as a result of an explicit user terminate request (called *CkExit()* in Charm). Other system message types exist for various system strategies, such as load balancing.

A translator translates the language described in part I of this paper [16] into C. Every entry point and the code associated with it is translated into a C-function, and a unique id is assigned to the entry point. Further, the translator generates code to create two tables of function pointers to hold the pointers to the entry point functions for chare entry points and BOC entry points, respectively. These tables are indexed by the entry point ID. The ids and tables are generated at runtime to allow for the separate compilation of multiple modules.

Upon encountering a *new-chare* message, the kernel first creates the data area associated with the newly created chare or BOC. In the case of *for-chare* and *for-boc* messages, the destination chare or BOC associated with the message is used to extract the data-area allocated for the destination object.

Processing a message M now entails **(a)** extracting the function pointer F from the appropriate function table, using the entry point associated with M, and **(b)** invoking the dereferenced function (see Figure 4) using two arguments: a pointer to the destination object's persistent data, and the message to be delivered to it. Once processed, the message M is deemed to have been consumed

by the entry point.

Once a chare C is created on a processor P, it remains *anchored* to P for its lifetime in the current implementation described in this paper [2]. C is then uniquely identified by the pair <*P, C's data-area-ptr*>. These fields constitute the globally unique ID assigned to C. So, once C is created, all messages for C are delivered to, and processed on, P.

On shared memory architectures, anchoring of chares automatically provides mutual exclusion on chare and branch office data areas. (On nonshared memory machines, since sharing of data is not possible, this is not a concern.) Note that the language requires that no two entry points of the same chare or a branch office instance should be allowed to execute concurrently. On nonshared memory machines, the anchoring has an added benefit in that it is possible for the kernel to direct *for-chare* and *for-boc* messages efficiently to the appropriate processors.

5 Machine interface

The machine interface necessary for implementing Charm is small and uses the common low-level primitives of the specific architecture. For example, on a shared memory machine, the machine interface that needs to be defined consists of timers and locks. On a nonshared memory machine, the interface that needs to be defined consists of timers, functions to probe and receive messages from the network, and functions to send and broadcast messages. Our experience has been that this interface is small and the functionality required is supported by most parallel vendors. The only special case was the interface for the network of workstations.

A network of workstations can be regarded as a collection of processors [8] that do not share an address space. From this viewpoint, the nonshared memory implementation works unchanged on networks of workstations, and indeed the workstation version includes the same modules for the core kernel and strategies. However, the machine layer is more complex than that for parallel machines, such as Intel Paragon, because it needs to implement efficient message passing. We chose to implement a reliable communication layer using UDP socket connections. It was also necessary to implement fragmentation and reassembly of messages to accomodate large messages. A sliding

[2] Chare migration is supported, but is outside the scope of this paper. For details see [11].

window protocol was implemented to reduce the number of acknowledgement packets transmitted by the destination. This protocol uses a timeout mechanism to retransmit lost UDP datagrams to reduce network traffic further.

As we will show in Section 8, the performance of Charm on a network of workstations was quite good, especially considering that the measurements were made in the presence of other activity on the workstations and the network.

Two important factors inherent in the workstation environment were found to impede performance.[3] The first is that background network traffic, often caused by other workstations sharing the Ethernet, can occasionally limit the available network bandwidth and result in message transmission delays. Since the sliding window protocol implemented uses a timeout mechanism for retransmission, long delays in acknowledgement can sometimes result in retransmission of packets, further compounding the problem [4].

The second problem was the effect of the operating systems running independently on each of the workstations. It is not possible to guarantee that all the Unix processes running the different copies of the Chare kernel (one per processor) are all scheduled on their respective processors at the same time. Occasionally, this too results in unnecessary timeouts in the sliding window protocol, which in turn unnecessarily retransmits timed-out packets. The skew in process scheduling caused by the independence of workstation operating systems also makes it very difficult to reliably determine the execution time of a Charm program. We are currently improving the implementation to minimize the effect of these problems. Note that Charm, with message driven execution, is better equipped to deal with these problems than message passing libraries, such as PVM [30, 12], because processors are not blocked waiting for just one message.

[3] Some of these problems can be eliminated by creating a contrived environment with an isolated network and no other processes running on each workstation, but that, in our opinion, defeats one of the primary objectives of using a network of workstations as a parallel machine – that of time-sharing it with other users.

[4] An interrupt based solution to this problem is currently being implemented.

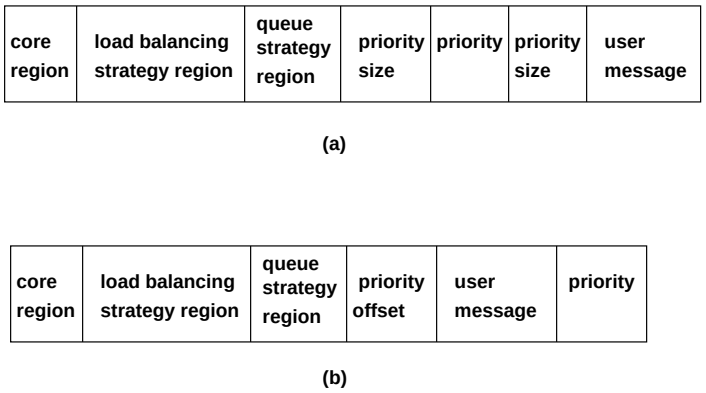


Figure 5: Layout of a message. The sizes of different regions are not proportionately shown.

6 What is in a message?

A message in Charm contains, in addition to the user data, some system information for the core which contains the type of message (new-chare, for-chare, for-boc, etc.) and data needed for its processing (address of chare, entry point, etc.). In addition, Charm provides various load balancing and queuing strategies which are plug compatible: to use a new load balancing or queuing strategy, all one needs to do is to re-link the program with the new strategy. These strategies require additional information which must be part of the user message. However, different strategies may (and do) require different amount of information. For example, the random load balancing strategy requires no information, while the ACWN strategy requires information about the load, which is "piggy-backed" on regular messages for efficiency. The *CkAllocMsg* call invoked by the user to allocate the message must allocate space for the core information and the load balancing and queuing strategies. This is accomplished by letting each strategy specify the amount of data it needs which is allocated by the system. Regions in the message are then allocated for different purposes, and each region is handled by the corresponding strategy. Thus, the core, each load balancing strategy, and each queuing strategy, define *core-size*, *ldb-size*, and *queue-size*, which is then used to allocate the message.

A minor complication arises because Charm allows priorities to be specified for a message: a message can have no priority, an integer priority, or a variable length bit-vector priority [27]. The message must also include this priority field. Since the priority can have different lengths for different messages in the same program, a size field is also necessary to determine the size of the priority.

Our first step, shown in Figure 5(a), was to group all the system related information into one chunk and allocate it right before the user message. One consequence of not having any system information after the message was that the user was allowed to allocate larger messages than what they finally sent out — this is useful if one cannot estimate the necessary message size at the time of allocation.

Since one needs to traverse from one region of the message to another (user message to core area, core area to ldb, core area to queue, message to priority, etc.), two words, each containing the size of the priority field, flank it on either side so that traversal from either direction is possible. Note that this layout implies that for traversal from the user to the core region, one needs to determine the length of the priority field, because it can be of variable size. Since the various regions of the message were traversed often, this became a major efficiency problem. Further, since a significant number of Charm programs did not use priority, the common case was affected by the presence of priority.

As a result, the priority field was moved to after the user message, retaining the other regions before it, as shown in Figure 5(b). An offset was maintained to keep track of the priority field. Now, all traversals not involving priority can be done in one operation: the variable lengths of the fields that vary from run to run (due to different strategies being linked) are pre-computed at initialization. With this design, it becomes necessary for the user to know the size of the message that is sent out at the time of allocation.

7 Implementation of other language features and strategies

Once the core kernel has been implemented, the remaining language features and strategies can be implemented using branch office chares. This is in line with our design objective of keeping a small core. For historical, as well as efficiency reasons, the implementation was done in a post-translated version of the branch office chare. The information sharing abstractions were implemented, mostly for reasons of efficiency, using shared memory abstractions on shared memory machines.

7.1 Information Sharing Abstractions

In addition to messages, chares can share data with the five information sharing variables (described in Part I[16]), namely, *read-only* variables, *write-once* variables, *accumulator* variables, *monotonic* variables and *distributed tables*. In this section, we discuss the implementation of these information sharing abstractions on shared and nonshared memory machines. On a nonshared memory machine, all the abstractions are implemented as branch office chares. However, on shared memory machines, a more efficient implementation is possible using shared variables: in this case we choose the efficiency criterion over the small core criterion.

On shared memory machines, each of the the first four variables is implemented as a shared entity with an associated lock. Operations are performed in a mutually exclusive manner using locks. A distributed table is implemented using a hash-table array. The key of an entry is used to hash into the array, and each array element is a chain of entries whose keys map to the same index. A lock is associated with each index in the array to provide mutually exclusive access to chains. The size of the array used to implement the distributed tables is a function of the number of processors requested and is allocated during system initialization.

On a nonshared memory machine, each of the five abstractions is implemented as a BOC: Each branch maintains a local copy of the variable in the case of read-only, write-once, monotonic, and accumulator variables. In the case of distributed tables, the entries are divided amongst the branches of the BOC. Read-only variables, and statically created accumulators and monotonic variables, are initialized during the initialization phase of the system.

Write Once variables, which are really a dynamic version of read only variables, are initialized by the *CreateWriteOnce* call. A copy of the variable is first sent to the branch on processor 0 of the corresponding BOC. This branch assigns the variable a unique index, which serves as the identifier for the write-once variable, and then broadcasts the value and identifier of the variable to each of the branch nodes. Each node, after creating a local copy of the write once variable, sends a message to the branch on processor 0 (along a spanning tree in order to reduce bottlenecks) saying that it has created the variable. When it has received an acknowledgement message from all the nodes, the branch on processor 0 sends the identifier of the write once variable to the specified address.

A write once variable can be read by the *DerefWriteOnce* call. This call returns the pointer to the local copy of the variable. The pointers to all the WriteOnce variables are stored in a table indexed by the identifier of the write-once variable.

The *Accumulate* call results in the application of the *add* function on the local value on the branch chare. The *CollectValue* call is used to (destructively) read the value of an accumulator variable. This call results in a broadcast to all branches. Each branch chare then combines its value with the values of the accumulator on its children in the spanning tree before sending the accumulated value up to its parent. At interior nodes of the spanning tree, the values are combined using the user defined combine function associated with the accumulator definition (See Part I). The branch on processor 0 communicates the final value to the chare specified in the CollectValue call.

An update on a monotonic variable is performed by the *NewValue* call. The *NewValue* call can be implemented in two different ways: *combining via a spanning tree* or *flooding*. In the *spanning tree* implementation, the call results in the branch chare updating its local value (with the corresponding update), and sending a copy of the new value up to its parent branch chare in the spanning tree on the processors. Every branch combines values it receives from its children with its own by waiting for some fixed interval of time before sending its local value up to its parent branch chare. The root of the tree broadcasts the value to all branch chares. In the *flooding* implementation, the call results in the branch chare updating its local value and sending a copy to its immediate neighbors (a dense graph on the processors is chosen). A processor, which receives a new value from a neighbor, first checks if the value provided is better than what it currently owns. If the value is better, it propagates a copy of the value to its own neighbors. In both of the above implementations, the value of every update may not be simultaneously available to every branch, but shall be eventually available. Users may choose the monotonic implementation best suited to their application. A monotonic variable can be accessed using the *MonoValue* system call; this call returns the value of the local copy of the variable on that node.

Updates on entries in a distributed table can be carried out by calling the system calls *Insert* and *Delete*. Again, as in the case of shared memory systems, a *hashed chaining* scheme is used. The key of an entry is hashed to obtain the processor number of the branch which stores the portion of the table to which this entry belongs and the index in the table on that branch. An update message

```
BranchOffice LoadBalance
{
   entry BranchInit:  (message InitMsg *msg)
   { /* initialize load balancing strategy */ }

   public AddPiggybackInfo(message)
   { /* add status information for neighbor */ }

   public ExtractPiggybackInfo(message)
   { /* receive status information from neighbor */ }

   public NewChareFromLocal(message)
   { /* what should be done with a new chare message generated locally?  */ }

   public NewChareFromNetwork(message)
   { /* what should be done with a new chare message from the network?  */ }
} /* branch office */
```

Figure 6: The generic interface for a load balancing strategy

is sent to the required branch, which carries out the update operation and back-communication of update, if specified in the call options. The *Find* call is used to read entries in distributed tables. The key provided is used (as described above) to determine the branch and index. A message is sent to the corresponding branch chare to find the entry and reply back to the supplied address.

7.2 Dynamic Load Balancing

Load balancing is significantly easier on shared memory machines than on message passing machines. As mentioned in Section 4.2, on a small shared memory machine, a shared queue is used for *new-chare* messages that do not have a fixed destination. The use for a shared queue for such messages improves the load distribution. This obliviates the need for an explicit load balancing module for shared memory machines. Saletore [26] has developed many different strategies to balance load on large shared memory machines.

On a nonshared memory machine, a suite of algorithms have been implemented, including random load balancing, adaptive contracting-within-neighborhood (ACWN) [15], and a token-based local manager strategy for load balancing of prioritized tasks [29]. Any one of these strategies may be linked together with a Charm program to produce the final executable code. Additional strategies

may also be added by users as long as they conform to the prescribed interface shown in Figure 6. The interface is general enough to permit the easy addition of more strategies to the suite. Load balancing strategies are implemented as branch office chares in the Chare Kernel. The basic interface gives control to the strategy whenever a new-chare message is received: it may have been created locally or may arrive from the network. Individual strategies may define other functions, including those to periodically send and receive status information from neighbors.

Figure 7 illustrates how the *adaptive contracting-within-neighborhood* (ACWN) [15] strategy can be implemented using the interface. At the branch initialization entry point *BranchInit*, the number of neighbors in a processor's neighborhood and the list of neighbors are recorded and stored. Note that this is performed on every processor. This information depends on the interconnection network amongst the processors and is fine-tuned for each architecture.

In the ACWN scheme, when a *new-chare* message is generated, the BOC function *NewChareFromLocal()* is called. The function determines the least loaded neighbor in the processor's neighborhood and sends the message to that neighbor. A processor may piggyback its own load status on the message in the process. If the local processor is itself the least loaded, the message is enqueued in its local queue. Processors also exchange explicit load status messages periodically to keep their status information updated on their neighbors. An entry point, NeighborStatus, is provided for this purpose. For every message that is sent out, the kernel also calls the public function AddPiggybackInfo, which adds information about load on the processor onto the message.

For a different load balancing scheme, e.g. the *gradient model* [18], the load balancing process may be awakened periodically by the kernel to balance loads whenever the pressure gradient falls or rises above a certain threshold.

Additional entry points may be added as desired for implementing different schemes. However they must include the functions shown in Figure 6.

7.3 Queuing Strategies

At any point in time, there may exist many messages ready for execution on a processor from which it may select. The sequence in which available messages are picked up for execution often

```
BranchOffice ACWN
{
   int NumNeighbors, NeighborsList[MaxNeighbors];

   entry BranchInit:  (message InitMsg *msg) {
      NumNeighbors = GetNumNeighbors(MyNodeID);
      GetListOfNeighbors(MyNodeID, NeighborsList);
      InitializePeriodicCall(PeriodicStatus, PERIOD);
   }

   private PeriodicStatus() {
      for each neighbor in NeighborsList {
         message = Allocate(StatusMsg-size);
         PiggybackLoadStatus(message);
         SendMsgBranch(NeighborStatus,message); /* send neighbor my status */
   } }

   public AddPiggybackInfo(message)
   { PiggybackLoadStatus(message); }

   public ExtractPiggybackInfo(message) {
      CopyLoadStatus(message);
      if (MyLoad() > minload) minhops = MAXHOPS;
      else if (NeighborLoad() > minload) minhops = 0;
      else minhops = MINHOPS;
   }

   public NewChareFromLocal(message)
   { SetHops(message, 0); LdbStrategy(message); }

   public NewChareFromNetwork(message)
   { IncrementHops(message); LdbStrategy(message); }

   private LdbStrategy(message) {
      if (GetHops(message) ≥ maxhops) EnQueue(message);
      else if (MyLoad() < minload) EnQueue(message);
      else if (GetHops(message) ≤ minhops) {
         neighbor = LeastLoadedNeighbor();
         PiggybackLoadStatus(message);
         SendMsgBranch(message, neighbor);
      }
      else EnQueue(message);
   }
} /* ACWN load balancing branch office */
```

Figure 7: The *adaptive contracting-within-neighborhood* (ACWN) load balancing strategy implemented as a BOC.

has a significant impact on various performance metrics. For example, the memory utilization is impacted significantly by this choice. In divide-and-conquer and search problems, which are

both tree structured computations, a FIFO strategy for selecting the next message will lead to an exponential growth in the memory required, whereas a LIFO strategy would require memory proportional to the depth of the search tree. In other applications, one may wish to associate priorities with messages, and following the priorities may mean a quicker completion time. In order to effect any search strategy, it is therefore necessary to be able to specify the order in which messages are to be processed. The Chare kernel provides a suite of different queuing strategies for this purpose. The queuing strategy specifies the order in which messages in the work pool managed by the kernel are to be processed. The application programmer simply links in the most suitable strategy to create the executable code. This feature also makes it possible for the programmer to experiment with different strategies to determine the most suitable one.

The queuing strategies provided include depth-first (stack), breadth first (queue), and a combination of the two. The combination based strategy attempts to provide a breadth-first strategy in low load situations (like near the root of a search tree). This permits the load balancing (ACWN) to distribute sibling nodes across the available processors. Once high utilization has been achieved, the load balancing algorithm (ACWN) keeps work local to the creating processor as far as possible. This suggests that the depth-first strategy is more efficient in this situation. This strategy has been found to be quite effective in tandem with the ACWN load balancing strategy.

Priority-based strategies are supported in Charm. Here, the user associates an integer or a bit-vector with each message, which specifies the priority of the message. The messages are processed in priority order on each processor. Note that the use of priorities does not eliminate the effects of non-determinism, although they are significantly minimized.

On shared memory machines, as described in Section 4.2, three distinct shared queues of messages are maintained by the queue manager. When a new message is to be picked for processing, the queues are examined in the order: *for-chare* message queue, *for-boc* queue followed by the *new-chare* queue. This order was empirically found to be the most efficient with respect to memory utilization and queue contention. The order in which messages in any queue are picked up is determined by the choice of the queuing strategy selected.

On message passing machines, the queue manager maintains distinct queues on each processor. There is no notion of a shared queue. The *new-chare* queue is periodically load-balanced across

processors by the load balancing BOC.

7.4 Quiescence Detection

Charm allows the user to detect periods in the computation when there are no *activation messages* in the system, either being processed or waiting in queues. The algorithm to detect quiescence was designed with two major objectives:

- To detect quiescence as soon as possible after it occurs.
- To have minimum overhead, as measured by the interference or elongation of execution time of the user program.

The algorithm automatically adapts to system loads — it generates few control messages (messages used by the quiescence detection algorithm) when the system is busy, and more control messages when the system is lightly loaded. Control messages generated in the latter case do not adversely affect system performance, because they only occupy computational resources of idling processors.

Quiescence detection is implemented using a BOC. The algorithm used has two phases. All communication between the branches occur along a spanning tree covering the processors. In the description below, all references to the *parent*, the *children*, the *root*, or the *sub-tree* of a processor are with respect to the corresponding entities in the spanning tree on all the processors. We denote the first and second phases of the algorithm as Phase 1 and Phase 2, respectively. The algorithm uses three kinds of control messages:

1. *initialization:* these are broadcast to all branches, and result in the initialization of Phase 1 or Phase 2 on all the branches.
2. *idle:* these messages are sent up to the parent during Phase 1. An idle message signifies that each processor in the sub-tree below has been idle at least once since the start of this phase. It does not necessarily mean that all the processors were idle *simultaneously*.
3. *activity:* these messages are sent up to the parent during Phase 2 and contain a report of activity (creation and processing) in the sub-tree rooted at the sending processing element.

We use the construct — **wait until (condn)** — in the description of our algorithm. The process executing the **wait until** [5] is suspended till such time as the **condn** becomes true. Each branch maintains the following counts:

- n_c: this is the sum of the number of *activation* messages **created** on this processor.
- n_p: this is the sum of the number of *activation* messages **processed** on this processor.

Each branch also has two other counts N_c and N_p — they are used to estimate the number of messages created and processed, respectively, in the sub-tree rooted at itself. These are initialized to zero at the beginning of Phase 1 and Phase 2, and are sent up with *idle* and *activity* messages.

The algorithm appears in Figure 8. Phase 1 is called on each processor immediately before the user computation begins. Only one phase of the quiescence detection algorithm will be active at any time.

In Phase 1, each leaf branch waits until its processor is idle and then sends an idle message to its parent with the counts N_c and N_p initialized to n_c and n_p, respectively. All other branches wait until they receive one idle message from each child, adding the values of N_c and N_p in these idle messages to their local values. Having received idle messages from all its children, the branch waits until its processor is idle, and then it sends an idle message to its parent. The idle message contains the values of the counts N_c and N_p, which have been incremented with the values of n_c and n_p, respectively, on that branch. When the root has received idle messages from all its children branches, it decides whether the system can be *idle* by comparing the values of N_c and N_p. If they are equal then there's a high probability that all *activation* messages have been processed in the system. If the two counts are not equal then the root initiates Phase 1 again, otherwise the root initiates Phase 2 on all the branches.

In Phase 2, the branches send up their *activity* report messages containing the new values of N_c and N_p. *Activity* messages from branches are combined in the same way as in the first phase of the algorithm. When the root branch has received one activity message from each of its children, it compares the old and the new values of N_c and N_p. If these values are the same, it implies that

[5] We use this construct only for convenience of description; it is not used in the implementation.

```
Phase 1()
{
    N_c = 0; N_p = 0;
    wait until (RecdMsgsFromChildren()); /* wait until messages have */
                                         /* been received from all children */
    add to local N_c and N_p the values recd. from children;
    wait until (Idle()); /* wait until this processor has no activation messages */
    N_c = N_c + n_c; N_p = N_p + n_p;
    if (RootSpanTree()) /* check if this processor is the root of the spanning tree */
        if (N_c ≠ N_p)
            Broadcast message to begin Phase 1
        else
            N^old = N_c /* N_c == N_p */
            Broadcast message to begin Phase 2
    else
        Send message with N_c and N_p to Parent in Spanning Tree
}


Phase 2()
{
    N_c = 0; N_p = 0;
    wait until (RecdMsgsFromChildren()); /* wait until messages have */
                                         /* been received from all children */
    add to local N_c and N_p the values recd. from children;
    wait until (Idle());
    N_c = N_c + n_c; N_p = N_p + n_p;
    if (RootSpanTree()) /* check if this processor is the root of the spanning tree */
        if (N^old == N_c)
            Report Quiescence
        else
            Broadcast message to begin Phase 1
    else
        Send message with N_c and N_p to Parent in Spanning Tree
}
CreateMessage() { n_c ++ }
ProcessMessage() { n_p ++ }
```

Figure 8: The quiescence detection algorithm

there has been no new activity in the system, and the root reports *quiescence*; otherwise the root initiates Phase 1 again.

7.5 Conditional Packing

One of the problems encountered when writing programs on nonshared memory machines is that messages that are transmitted from one processor to another need to be packed in a contiguous

format for transmission, since pointers are generally not valid across processors. This can be costly when the data structures are complex, for example, large graphs or trees. It is clearly necessary to "pack" messages into this format only when messages cross address space boundaries. Thus, for shared memory no packing is necessary, and on nonshared memory machines, packing is necessary only when the source and destination processors are different. The remainder of this section is therefore applicable only to nonshared memory machines.

A typical Charm program creates several thousand chares and lets the kernel decide their processor assignment dynamically. Thus, the decision of whether or not to pack a message before transmission can only be made at run time. If both source and destination chares are resident on the same processor, this overhead is avoided.

The kernel does not maintain any semantic information on the structure of messages. The Charm translator translates messages into C *structs*. Even if the kernel were to maintain information on the structure of every message, it is very easy for programmers to override this information using type-casting in C. Checking for such non-conformance is a hard problem. It cannot therefore determine if, and how, a message is to be packed.

This problem is addressed by requiring the user to provide "pack" and "unpack" routines for each message type that requires packing as part of the Charm program. These routines are unique in that they are defined by the user but invoked as and when necessary by the kernel. If a decision is made to send a message out to another processor, the appropriate "pack" routine is called at the source processor. Correspondingly, the "unpack" routine is called by the destination processor before the message is processed. We show in Section 7.2 that this permits the kernel to significantly reduce the average packing overhead for messages over the execution of the program.

7.6 Dynamic creation of branch office chares and other entities

A branch office chare is said to be created *dynamically* if it is created outside the *CharmInit* entry point. Since Charm does not allow any synchronous calls, establishing an entity, such as a branch office chare with a single unique name on remote processors, involves a split-phase transaction: the first phase is the request for creation, and the second phase is confirmation of creation id which is

sent to a user-specified entry point. Dynamic creation of BOCs is implemented using a statically created branch office chare.

The node requesting creation of a BOC sends a message to the branch on processor 0 asking for a unique id. In this message it also sends the address of the user code that needs to be invoked once the BOC has been created. Once the branch on processor 0 returns a unique id for the BOC, the requesting branch broadcasts the user's creation message asking all processors to create their branch. Each processor participates in an 'asynchronous' reduction operation once it has created its branch: when all processors are done, processor 0 sends a message with the id of the BOC to the specified address. The need for this protocol arises because the core region of the message has no space for any more information (in this case the address to which the BOC's id must be returned). Adding such information to the core region would increase its size, which would be wasted in the general case, which was an overhead we deemed unacceptable. Similar protocols are used to implement the dynamic creation of other entities, such as write-once variables, which are implemented using dynamic branch office chares.

8 Parallel Programming Using Charm

As should be evident from the earlier discussion, the Chare kernel supports a coarse-grained data-flow style of execution where a message is the equivalent of a token in a data-flow machine. The amount of available parallelism is determined by the number of messages that can be processed in parallel. Furthermore, it is possible to exercise *granularity control* by varying the amount of processing performed per message. This is done by varying the average amount of computation in an entry point. This ability to vary the grain size is essential in controlling the parallelization overhead in the Chare kernel.

In the following discussion, we describe how parallel programs are developed using the Chare kernel from the application programmer's point of view. We wish to emphasize, however, that the following discussion describes a general approach for performance improvement and cannot necessarily be applied to every application that one wishes to parallelize.

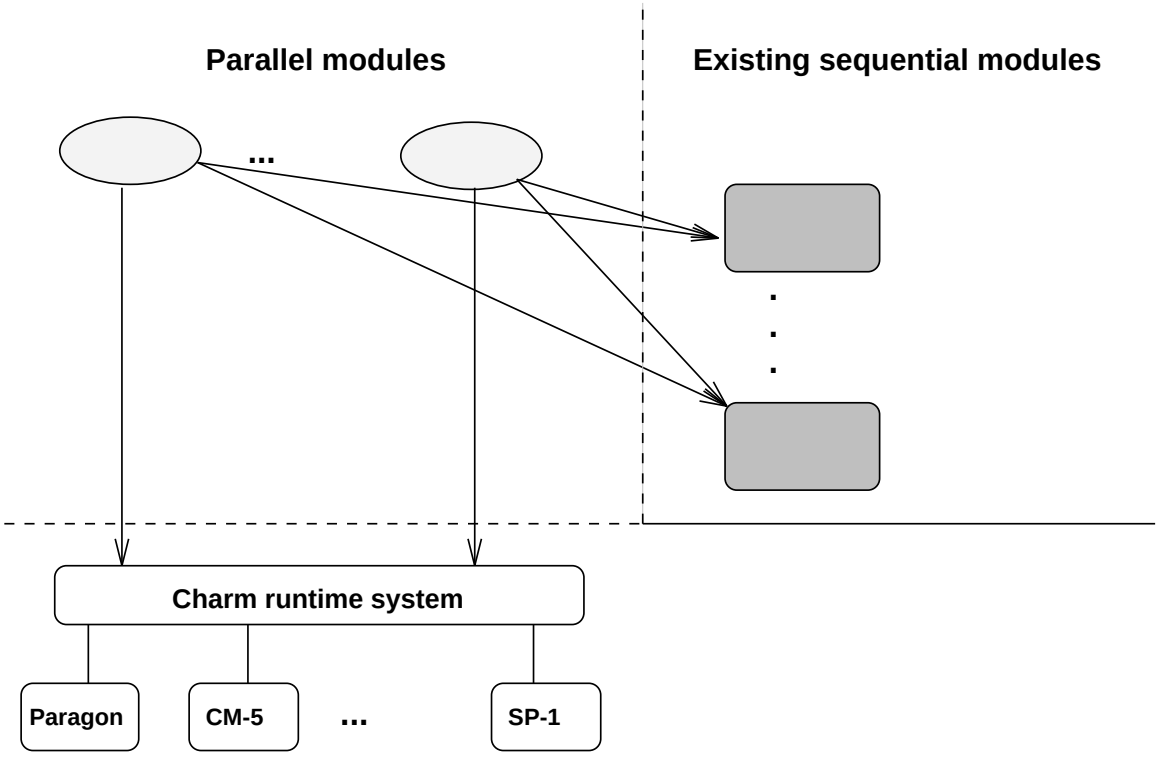


Figure 9: On each architecture, the Charm program can be linked with the most suitable memory management strategy, load balancing strategy, and queuing strategy available. The program is also built on top of an existing sequential implementation wherever possible.

8.1 Designing Charm Programs

In designing Charm programs, we list a few rules of thumb that we have compiled based on our experience in Charm programming. In Figure 9, we describe our philosophy diagrammatically on how Charm parallel programs should be developed. We believe that, in virtually every discipline, significant advances have been made in developing sequential algorithms. These algorithms have been developed after several years of effort in these application areas. In several cases, these algorithms are not amenable to parallelization, in that parallel versions of these algorithms may result in in significant loss of quality and poor speedups.

This is clearly evident in a wide range of VLSI CAD applications including test pattern generation [20, 21], cell placement [4, 24], global routing [6, 25] and a molecular dynamics application EGO. We believe it is therefore necessary to build the parallel algorithm *around* the best available sequential algorithm. By this, we mean it is necessary to reuse large modules of code that implement the best

available sequential algorithm in the parallel algorithm design.

The parallel algorithm can then be developed as follows. We assume that the problem instance is large enough to warrant the use of parallel processing. The design of a parallel algorithm typically involves developing a scheme to decompose the problem into smaller and smaller subproblems. Note that this decomposition can be made *independent* of the number of available processors. This decomposition is stopped when the size of the subproblems falls below some user-defined threshold. It is desirable to create enough messages representing subproblems so as to be significantly larger than the number of processors available. This permits effective load balancing. As we show in Section 9, the overheads incurred in creating large numbers of messages are usually very low. Once the threshold has been reached, the best available sequential algorithms can then be invoked to solve the subproblems efficiently.

This approach has been used very successfully in the design and implementation of a variety of parallel applications. For most of the applications, the parallel algorithm was developed around existing sequential code.[6]

8.2 Grainsize Selection

A consequence of the Charm cost model is that the user must make a choice of grainsize while writing Charm programs. For example, while writing a divide-and-conquer style program, one may decide to carry out sub-computation below a certain level sequentially; in a matrix-oriented computation, one may decide to block the sub-matrices below a certain size in a single chare. This decision must be based on the proportion of the overhead in relation to the useful computation time. Let g (grainsize) be the average computation time per message. The overhead is roughly proportional to the number of messages, and so is inversely proportional to the grainsize for a given computation. The summation of time-to-finish on all processors can be expressed as:

$$T = T_{\text{computation}} + T_{\text{overhead}} + T_{\text{idle}}, \text{ where the overhead } T_{\text{overhead}} = K * T_{\text{computation}}/g.$$

[6]The exception to this was the parallel Prolog compiler [23]. The lack of availability of the source code of a fast sequential Prolog compiler caused us to use a slower garden-variety compiler. This "error" quickly educated us on the importance of an efficient "sequential component" in the development of efficient parallel code.

The proportionality constant, K, reflects the overhead of sending and receiving a message, and in properly overlapped message-driven programs includes only the software overhead per message. Thus, if one increases the grainsize g continuously, beyond a certain point the contribution of the overhead term is sufficiently small that increasing g any further does not lead to appreciable decrease in T. Moreover, the smaller the grainsize in comparison to $T_{computation}$, the smaller is the idle time, as the load balancing strategy has more opportunities for balancing work (which is particularly useful for irregular computations mentioned in the assumption A3). Therefore, Charm programs should be written so as to yield as small average grainsize as possible, but large enough to subsume the overhead. An important consequence of this is that the grainsize decisions are *independent* of the number of processors in the system, as the overhead per message is largely independent of the number of processors in the system.

Note that, consistent with our minimality requirement, Charm leaves to the programmers the responsibility of decomposing the computations such that they lead to reasonable grainsizes. A language that automatically makes decisions regarding grainsizes may be desirable (and can be implemented on top of Charm), but it is not the function of Charm to make such decisions.

For a given application, it is important to note that, as we increase the grain size, we experience a reduction in the available parallelism. Moreover, if very few granules are created, each will perform large amounts of computation, leading to the likelihood of load imbalance. Reducing the grain size, on the other hand, can increase the parallelism, but at the expense of increased overhead. However, a range of grain sizes exists for which the application performance does not vary significantly. For a given machine, the lower bound of this range represents the *ideal* grain size for the application since it permits the maximum amount of parallelism to be exploited without incurring any observable overheads.

Based on several large applications developed so far using Charm, our experience suggests that in most cases an average grain size greater than 10-50 milliseconds is sufficient to restrict the overheads to within 5-10% of the one-processor "infinite grainsize" execution time. This corresponds to the uniprocessor performance of the application, and is determined by the efficiency of the sequential algorithms selected (see Figure 9).

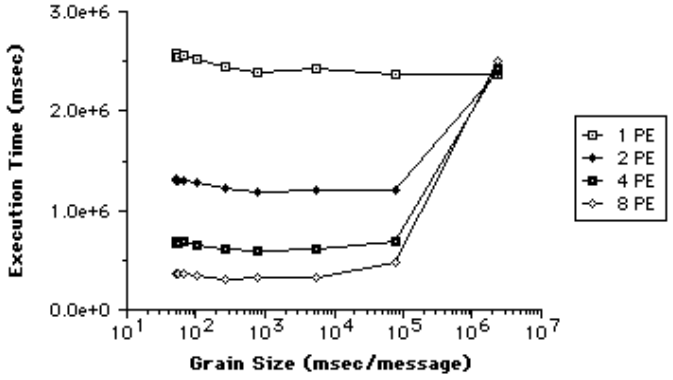
However, it should be clear that the ideal grain size will vary depending on a number of factors: **(a)** the computation speed of the underlying architecture, **(b)** the communication latency for message transmission, **(c)** the overheads in the target machine operating system (Charm is written as an application program on each of the target machines), **(d)** the average length of messages generated in the application, and **(e)** the amount of parallelism available in the application. In Charm, **(e)** can be viewed as the number of messages generated over the course of the application, since each message represents a segment of sequential computation.

Determining the ideal grain size automatically for any given application is a very hard problem. Moreover, for a given application, as mentioned above, the ideal grain size will differ from architecture to architecture. This may appear to contradict the portability across architectures that Charm claims to deliver. However, we show below that **(a)** it is not necessary to determine the ideal grain size in order to obtain efficient execution and **(b)** it is indeed possible to select a grain size that will work on all the available target architectures and thus meet *Charm's* portability criterion.

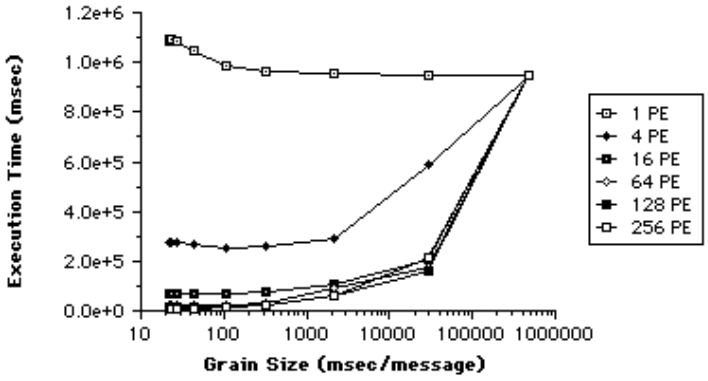
To study the effect of grain size on Charm programs, we performed the following experiment using a benchmark program: the *graph-coloring* program. We plotted the total execution time of the benchmark as we varied the grain size on each of the three architectures: the Encore Multimax, the NCUBE/2 and the network of workstations. The data was obtained on a network of Sun Sparc 2 workstations. We wish to emphasize that the data was measured when the network and the other workstations were in use – this represents a more realistic estimate of expected behavior. We also varied the number of processors to see its effect on the performance.

The *graph coloring* algorithm (Figure 10) uses parallel search to determine whether a given graph with 600 nodes and 2338 edges is 3-colorable. We varied the grain size in this example by varying the depth of the parallel search tree, thereby varying the amount of work that has to be performed sequentially at the leaf nodes of the search tree. The program exhibited sufficient parallelism and gave good speedups on all three machine configurations. The data is presented using a logarithmic scale for the grain size on the x-axis.

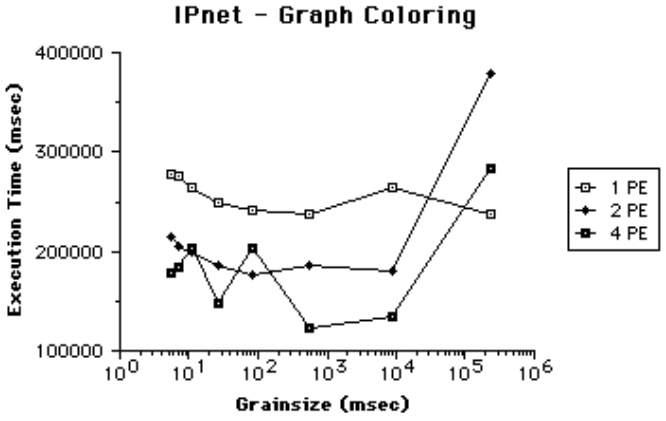
In Figures 10(a) the curve is quite flat as the grain size is varied from 10 ms to 100,000 ms per message for the *primes* example. As might be expected, this machine exhibits the lowest communication overhead, and has few processors. It can tolerate the widest range of grain sizes



(a) Encore Multimax



(b) NCUBE/2 Hypercube



(c) Network of Sun workstations

Figure 10: Performance of the *graph-coloring* benchmark on the Encore Multimax, the NCUBE/2 and a network of Sun Sparc 2 workstations. The effect on the execution time as the grain size and the number of processors are varied is plotted for each machine.

without affecting performance.

The NCUBE/2 data exhibits a flat curve for the range 10ms - 1000 ms for the grain size in the graph coloring example. The effect of load imbalance is evident as we go over a grain size 1000 ms on the NCUBE/2 for the smaller number of processors (Figure 10(b)).

On the Encore Multimax (Figure 10(a)), at the left end of the curve, a little performance degradation for the *graph coloring* problem is evident on one processor. The degradation is due to too small a grain size. However, even this effect disappears as the number of processors is increased. The smallest grain size shown in Figure 10 was the lowest possible in our formulation of the *graph-coloring* problem.

Figure 10 shows the characteristic curves observed when plotting grain size (average computation time per msg) vs. the total computation time of the parallel application.[7] It is clear from the graphs for the Multimax and the NCUBE/2 that if the grain size is not too small or too large, a large plateau is visible in the curve - this represents the range where the execution time is unaffected by changes in the grain size.

In Figure 10(c), the behavior of the *graph coloring* benchmark on the workstations is shown. For the *graph coloring* example, a very low grain size ($<$ 10 ms) affects performance as the number of processors is increased. This is because the high cost of communication cannot be amortized by overlapping computation in this range. However, as we go over 10 ms, the 4-PE case offers good speedups until load imbalance begins to affect performance. The system effects discussed in Section 5 preclude consistent behavior along the plateau as seen on the other two machines (except for the 1-processor case). However, the general behavior between grain sizes of 10 ms and 10,000 ms is similar to the plateau observed on the other two machines.

The right end of the curve represents the largest possible grain size. This is obtained when the entire program is executed sequentially. This is borne out in Figure 10(a,b), where as the number of processors is varied, all the curves meet for the largest possible grain size at the 1-processor execution time. Note that the same is not true for the network version: we attribute this to

[7] Similar curves were obtained for parallel Prolog [23], parallel circuit extraction [3], test pattern generation [22], and several other applications – they have been reported elsewhere.

costs incurred on the network version to poll the network for messages from other processors (even though there are no such messages). The shape of the curves on all three machines suggests that it is sufficient to pick any grain size that lies on the plateau of this curve and yet obtain very good performance. An interesting consequence of this characteristic of the curve is that the grain size can, at the same time, be made *independent* of the underlying target machine. Typically, the architectures with the worst computation/communication ratio tend to have the shortest plateau width. Thus, grain sizes that work for such architectures have been found to work quite well on other target machines as well.

9 Performance

Several large applications have been developed so far using Charm. They include several VLSI/CAD applications developed by P. Banerjee, B. Ramkumar, and others. A parallel Prolog compiler [23], a parallel circuit extractor [3], a parallel test pattern generator for sequential circuits [22], parallel logic synthesis based on transduction [10], parallel cell placement based on simulated annealing [17], and a parallel molecular dynamics program called EGO have been implemented using Charm. Most of these applications are reported in detail elsewhere – we do not repeat the algorithms here. We have observed that, on one processor, a Charm program typically loses less than 5% of its speed compared to a sequential C implementation of the same algorithm, and the speedups with increasing number of processors are excellent. More importantly, Charm outperforms programs written using vendor primitives because of capabilities of adaptive scheduling provided by message driven execution [13] and the availability of prioritization queuing and load balancing strategies [23, 3, 22, 10, 17].

A parallel Prolog compiler which exploits both AND and OR parallelism has been written using Charm [23], and is one of the first such systems to run efficiently on both shared and non-shared memory system. It was also one of the first implementations to demonstrate speedups on Prolog program on upto 256 processors (on an NCUBE/2) [23].

The parallel circuit extractor [3] was developed around the same sequential C code used by PACE2 [5] to permit a fair comparison of the two. On shared memory machines, the extractor outperforms

PACE2 [5], in spite of the fact that PACE2 is based on an algorithm designed specifically for shared memory machines.

The parallel test generator ProperTEST [21] was also one of the first parallel implementations of test pattern generation for sequential circuits to demonstrate the use of priorities to speedup up the parallel search consistently and improve fault coverage at the same time.

The parallel cell placement algorithm ProperPLACE [17] demonstrated the feasibility of using one of the best available sequential implementations of cell placement based on simulated annealing - Timberwolf [28] as the sequential component of the Charm application. A new parallel algorithm was built around Timberwolf and rendered portable through Charm.[8]

EGO is a parallel molecular dynamics application, which was originally a synchronous program written in Occam for the Transputers. It has been converted into a message-driven program written in Charm, and runs successfully on all parallel machines on which Charm is supported.

Several other applications have also been developed and are being currently developed using Charm. For reasons of space, they are not discussed here.

10 Discussion and Summary

We have shown that with an appropriate set of primitives, and an implementation of these primitives tuned to each specific machine, efficient portability can be attained. Charm recognizes locality of reference as the key principle that unifies MIMD machines, with or without shared memory. Chares, with their local variables, and message driven execution enhance locality. Implementation of messages is tuned to, and an appropriate form of load balancing is used for, each target architecture. The common modes of sharing are encapsulated in Charm data abstractions, so they are implemented efficiently on each target machine. Conditional packing ensures that programs written using this system can compete with even those written specifically for shared memory machines, as long as the grain size required is not too fine. Scalable techniques used in its design and implementation ensure that the system runs efficiently on machines with thousands of processors,

[8]The latest version of ProperPLACE and other CAD applications now use a new object-oriented message-driven environment called ProperCAD II [19], that has been fine-tuned for parallel CAD applications.

as confirmed by tests on a 1024 node NCUBE-II.

Acknowledgements

Our thanks to Weeg Supercomputing Center at the University of Iowa for access to the Encore Multimax, and the Sandia National Laboratories for access to their NCUBE/2 hypercube. Thanks are also due to Raymond Richards, who assisted in collecting some of the data reported in this paper, and to Ben Richards for writing the graph-coloring program in Charm. We also thank Wennie Shu for developing the first version of Chare Kernel, Kevin Nomura for developing the first shared version of Chare Kernel, Attila Gursoy for writing the memory management and queuing strategies, and Wayne Fenton for implementing write-once variables and timer checks.

References

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

[2] Agha, G.A. Concurrent object-oriented programming. *Communications of the ACM*, vol. 33, September 1990.

[3] Ramkumar B. and Banerjee P. ProperCAD: A Portable Object-Oriented Parallel Environment for VLSI CAD. *IEEE Transactions on Computer-Aided Design*, 13, no 7, July 1994.

[4] Sargent J.S. Banerjee P., Jones M.H. Parallel Simulated Annealing Algorithms for Standard Cell Placement on Hypercube Multiprocessors. *IEEE Trans. Parallel and Distributed Systems*, 1(1):91–106, January 1990.

[5] Belkhale, K.P, Banerjee, P. PACE2: An Improved Parallel VLSI Extractor with Parameter Extraction. In *Proceedings of the International Conference on Computer Aided Design*, pages 526–530, November 1989.

[6] R. J. Brouwer and P. Banerjee. PARAGRAPH: A Parallel Algorithm for Simultaneous Placement and Routing Using Hierarchy. *Proc. European Design Automation Conf. (EDAC-92)*, Mar. 1992.

[7] Chien, A., Dally, W.J. Concurrent Aggregates (CA). In *ACM SIGPLAN Principles and Practice of Parallel Programming*, Seattle, Washington, March 1990.

[8] Christopher Walquist. Implementation of charm machine interface on networks of workstations. Master's thesis, Dept. of Computer Science, University of Illinois, Urbana, December 1990.

[9] Dally, W.J. Fine-Grain Message-Passing Concurrent Computers. In *The Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, California, January 1988.

[10] De, K., Ramkumar, B., Banerjee P. ProperSYN: A Portable Parallel Algorithm for Logic Synthesis. *IEEE Transactions on Computer-Aided Design*, 13 no. 5, May 1994.

[11] Doulas, N., Ramkumar B. Efficient Task Migration for Message-Driven Parallel Execution on Nonshared Memory Architectures. In *Proceedings of the International Conference on Parallel Processing*, August 1994. (to appear).

[12] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputing level concurrent computations on a heterogeneous network of workstations. *Sixth Distributed Memory Computing Conference Proceedings*, pages 258–261, 1991.

[13] Attila Gursoy. *Simplified Expression of Message-Driven Programs and Quantification of their Impact on Performance.* PhD thesis, Departmant of Computer Science, University of Illinois at Urbana-Champaign, March 1994.

[14] L. V. Kale and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA-93.*, March 1993. (Also: Technical Report UIUCDCS-R-93-1796, March 1993, University of Illinois, Urbana, IL.

[15] Kalé, L.V. Comparing the Performance of Two Dynamic Load Distribution Methods. In *International Conference on Parallel Processing*, August 1988.

[16] Kale L.V. and Ramkumar B. and Sinha A.B. and Gursoy A. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994. (submitted).

[17] Kim S-H, Ramkumar B., Chandy J., Parkes S., Banerjee P. ProperPLACE: A Portable Parallel Algorithm for Standard Cell Placement. In *Proceedings of the International Parallel Processing Symposium*, Cancun, Mexico, March 1994.

[18] Lin, F.C.H, Keller, R. Gradient Model: A Demand Driven Load Balancing Scheme. In *International Conference on Distributed Systems*, pages 329–336, 1986.

[19] Steven Parkes, John A. Chandy, and Prithviraj Banerjee. ProperCAD II: A run-time library for portable, parallel, object-oriented programming with applications to VLSI CAD. Technical Report CRHC-93-22/UILU-ENG-93-2250, Center for Reliable and High-performance Computing, University of Illinois, December 1993.

[20] Patil S., Banerjee, B. A Parallel Branch and Bound Algorithm for Test Generation. *IEEE Transactions on Computer-Aided Design*, 9, no. 3:313–322, March 1990.

[21] Ramkumar, B., Banerjee P. Portable Parallel Test Generation for Sequential Circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–223, November 1992.

[22] Ramkumar, B., Banerjee P. ProperTEST: A Portable Parallel Test Generator for Sequential Circuits. *IEEE Transactions on Computer-Aided Design*, 1992. (submitted).

[23] Ramkumar B., Kale L.V. Machine Independent AND and OR Parallel Execution of Logic Programs – Part II: Compiled Execution. *IEEE Transactions on Parallel and Distributed Systems*, 5, no. 2, February 1994.

[24] Ravikumar, C.P., Sastry S. Parallel Placement on Hypercube Architectures. In *International Conference on Parallel Processing*, pages III: 97–100, August 1989.

[25] Rose, J.S. Parallel Global Routing for Standard Cells. *IEEE Transactions on Computer-Aided Design*, 9, no. 10:1085–1095, October 1990.

[26] Saletore, V.A. *Machine Independent Parallel Execution of Speculative Computations.* PhD thesis, Dept. of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1991.

[27] Saletore, V.A., Kalé, L.V. Parallel State-Space Search for a First Solution with Consistent Linear Speedups. *International Journal of Parallel Programming*, 1990.

[28] Sechen, C., Sangiovanni-Vincentelli A.L. The TimberWolf Placement and Routing Package. *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2:510–522, 1988.

[29] Sinha A.B., Kale L.V. A Load Balancing Strategy for Prioritized Execution of Tasks. In *Proceedings of International Parallel Processing i Symposium*, April 1993.

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