The Charm Parallel Programming Language and System: Part I — Description of Language Features*

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

*Dept. of Computer Science
University of Illinois
Urbana, IL 61801
E-mail: kale,sinha,gursoy@cs.uiuc.edu

**Dept. of Electrical and Computer Engineering
University of Iowa
Iowa City, IA 52242
E-mail: ramkumar@eng.uiowa.edu

Abstract

We describe a parallel programming system for developing machine independent programs for all MIMD machines. Many useful approaches to this problem are seen to require a common base of support, which can be encapsulated in a language that abstracts over resource management decisions and machine specific details. This language can be used for implementing other high level approaches as well as for efficient application programming. The requirements for such a language are defined, and the language supported by the *Charm* system is described, and illustrated with examples. Charm is one of the first languages to support message driven execution, and embodies unique abstractions such as branch office chares and specifically shared variables. In Part II of this paper, we talk about the runtime support system for Charm. The system thus provides ease of programming on MIMD platforms without sacrificing performance.

*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

Many parallel machines are commercially available. These include shared memory machines such as the Sequent Symmetry, Encore Multimax, and the KSR-1, and distributed memory machines such as the IBM SP-1, NCUBE-II, Intel iPSC/860 and Paragon, Convex Exemplar, and TMC CM-5. Many of these machines include hundreds of processors, each of which is a powerful microprocessor with performance of the order of tens of MFLOPs.

At the same time, there are many computation intensive applications that can benefit from parallel processing. These include applications in computational biology, computational fluid dynamics, design automation, weather prediction, various calculations/simulations in physics, discrete optimization, and many AI computations such as heuristic search, problem-solving, and planning.

However, programming these parallel machines remains a challenging and difficult task, except for a few highly regular kernels. The complexity of parallel programming arises from the inherent asynchrony of computational agents, with the concomitant correctness issues, and the need to consider performance issues beyond those in sequential programming, such as load balancing, network contention, and communication latencies.

In addition, different machines have different primitives, modes of operations, and cost tradeoffs. Even among MIMD[1] architectures, the programming environment supported on a machine is significantly affected by the characteristics exhibited by the architecture. For example, the memory organization of a machine often determines the support for inter-process communication on the machine. Shared memory machines, like the Sequent Symmetry, tend to support communication through shared variables whose access is controlled using mutual exclusion primitives. On distributed memory machines supporting a global address space like the KSR-1, communication is also supported through shared variables. However, a significant difference exists between the cost of accessing local memory and non-local memory. As a result, they are also called NUMA (Non-Uniform Memory Access) machines. Nonshared memory machines, like the Intel Paragon, support a significantly different style of inter-process communication — message passing using *sends* and *receives*. This form of communication isn't directly compatible with the shared variable approach. As a result of the differences in memory organization, programmers need to develop different versions of the same parallel program for different target architectures. This lack of *portability* is exacerbated by the rapid evolution of parallel machines, due to which even machines from the same vendor may not be directly compatible with their earlier machines. So, porting a parallel application written for one machine to another is expensive, thus adding to the development and maintenance costs of parallel software.

The complexity of parallel programming, exacerbated by the issue of portability, constitute hurdles that must be overcome before one can bring the substantial power of parallel processing to bear upon the broad class of significant applications. This paper describes Charm, a parallel programming language and its associated runtime system that is aimed at controlling the complexity of parallel programming. Charm was developed at the University of Illinois over the last several years, and is aimed at providing productive and effective parallel programming support for MIMD parallel machines. The targeted machines include both shared memory and distributed memory machines.

[1]Parallel algorithms for SIMD parallel machines are quite different from algorithms developed for MIMD parallel architectures. Since this difference is irreconcilable, except for data parallel languages, we restrict our attention to MIMD parallel machines. We believe that portability across SIMD and MIMD machines can be accomplished by implementing a common data parallel language such as High Performance Fortran [1, 2, 3] on SIMD machines using native machine primitives and on MIMD machines using a system such as Charm.

1.1 Requirements of a parallel programming system

A general technique for dealing with complexity involves employing a hierarchical structure. In order to identify the appropriate hierarchy to deal with the conceptual complexity of parallel programming, we note that the task of writing a parallel program for a specific machine involves:

1. deciding how to *decompose* the computation into parallel sub-computations,
2. deciding the *mapping* (assignment) of these computational actions to specific processors,
3. deciding when to *schedule* them, and
4. *expressing* these decisions using the machine dependent primitives

The highest level of hierarchy is then identified with the techniques which provide automatic decomposition, mapping, scheduling, and machine independent expression. The techniques at the lowest level provide only primitives to express the program in a portable manner, thus automating the task of machine independent expression. Various approaches can then be calibrated on this progression from high-level to low-level. In addition, approaches can also be distinguished by their generality or specialization. An approach may become specialized either by limiting itself to specific application domains, or to narrow parallel programming paradigms. We will examine extant classes of approaches with this hierarchy in mind, to motivate our approach.

1. One class of such approaches involves machine independent communication mechanisms. A programming system in this class allows users to express their parallel programs using generic communication primitives, which are abstracted from the primitives of the supported machines. The approaches in this class are characterized by their generality. Any computation that can be expressed on a MIMD machine can be expressed directly, with similar primitives. In Express [4] and in PVM [5, 6, 7], they are the primitives of a non-shared memory computer. These approaches are very useful toward achieving portability, and their utility is demonstrated by the popularity of systems such as PVM, and by the emergence of a evolving message passing interface standard (MPI) [8, 9, 10]. However, beyond portability, such approaches do little to control the complexity of parallel programming.

2. The next class of approaches define their own communication and synchronization mechanisms, which are independent of the machine primitives. Linda [11, 12] is an example of such a system — here processes communicate by depositing, fetching, or copying tuples from a common tuple space. In Strand [13] and Concurrent Prolog [14], processes, expressed as recursive clauses of a logic program, communicate by setting values to, and blocking for values of shared variables in a stream (represented as a list) of logic variables. The utility of such approaches depends on the appropriateness of the specific set of mechanisms for the particular parallel programming task. If the mechanisms are well-matched for the application at hand, and if the programmer is experienced in using them, they work very well.

3. In both of the above approaches, the programmer specifies the decomposition, the mapping, and scheduling. The High Performance Fortran [1, 2] standard is an example of approaches which free the programmer from the task of scheduling. The programmer still specifies the decomposition (via array distribution primitives) and mapping (implicitly, via the "owner computes" rule, which specifies that a computation is carried out on the processor that stores the variable or portion of the array being computed). Because it applies only to data parallel computations, this approach is not as "general-purpose" as the ones above. Another example of this class is Hudak's explicitly parallel functional programming [15, 16], where each function call can be mapped to specific processor under the programmer's control.

4. Systems such as Multilisp [17] and QLISP [18, 19, 20] allow programmers to explicitly specify the decompositions, while taking over both the tasks of mapping and scheduling. In Multilisp, for example, a computation may fork off a subcomputation to compute a value to be stored in a structure called a *future*. The parent computation proceeds as if the "future" already holds the value to be computed, while the system schedules the subcomputation on some available processor when possible. The parent may pass the future to other functions or even to other subcomputations. As soon as a process *touches* a future (i.e. needs the value of its content), it blocks to be awakened when the value in the future is computed. These approaches are not "general-purpose" because of the specialization induced by the computational paradigm of their base languages (LISP and functional programming in the case of the above examples) and the limited patterns of communication supported, e.g. communication occurs only via futures in Multilisp, and only via function call/return values in QLISP.

5. Parallelizing compilers, such as those for Fortran [21, 22, 23] aim at a very high level in the hierarchy: they attempt to take over the task of decomposition, mapping, scheduling, as well as machine dependent expression. They let the programmers write programs in their usual sequential languages (such as Fortran). A *parallelizing compiler* is used to detect the parallelism, and transform the program to a parallel program, which is then translated further for specific target machines. This approach has its obvious attractions: there is almost no additional parallel programming complexity. The pioneering work done at the University of Illinois [21, 22] and Rice [23] has demonstrated the viability of this approach for specific machines. However, this approach often fails to achieve the best possible results from a parallel machine for particular problems because the parallelizing compiler is often unable to specify the most efficient decomposition of a computation into parallel actions.

6. A related category of approaches involves languages such as Id [24, 25] and Sisal [26, 27], which can be thought of as providing sequentializing compilers for implicitly parallel languages. Programs specify *potential* decompositions of computations into parallel subcomputations, while the compiler choses which decompositions to ignore, and keeps track of data and control dependences to generate parallel tasks from the chosen decompositions. Fine-grained concurrent languages, such as Concurrent Aggregates [28, 29] and ABCL [30] also fall in this category.

7. Finally, domain specific packages allow the users to specify their problems, and possibly select solution strategies from the system's repertoire. These are also as high in the hierarchy as the automatic parallelizing compilers. However, they are obviously not general purpose programming systems — they trade off generality for feasibility, ease of development, and potentially higher efficiency. Examples in this class are rare, but we expect them to proliferate as parallel processing becomes more widespread. They may include packages for Finite Element computations, Computational Fluid Dynamics, Combinatorial Optimization, state-space search, etc. In addition to the data specifying the problem itself, the user of a CFD package may specify a differencing scheme, a numerical method, a strategy for solving the particular linear-system that may arise, and output options. The "system" would then link together different modules to satisfy the runtime choices made by the user.

The seven classes and their relative positions along the spectrum of low-level to high-level and (orthogonally) from general-purpose to specialized, is shown in Figure 1. Looking at this spectrum of approaches, it seemed apparent to us early in our research that there was a need for a general purpose language at a higher level in the hierarchy [31]. We need a language and system that satisfies the following requirements:

R1 *General purpose:* The system shouldn't be restricted to narrow classes of application domains, or to narrow parallel programming paradigms. It must be encapsulated in a language that

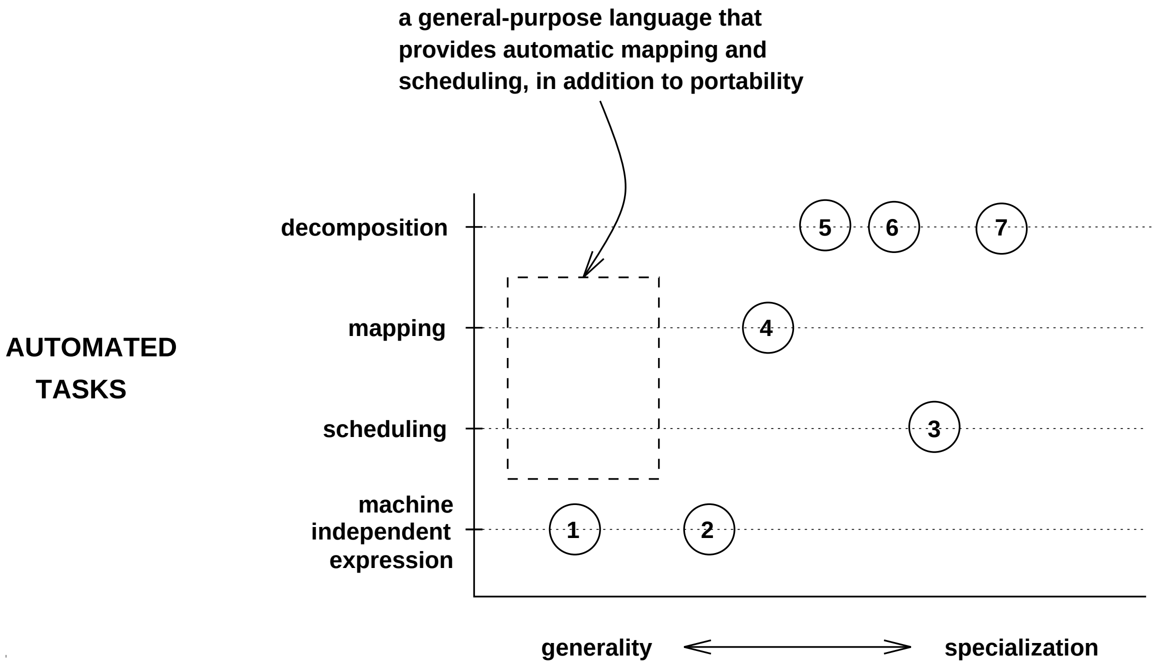

Figure 1: The spectrum of approaches to parallel programming.

can be used uniformly by implementors of individual schemes, or by end users. The set of primitives of the language must be rich enough to specify any computation that can be carried out on a MIMD machine with comparable efficiency (this is a "completeness" or "generality" requirement). The abstraction layer needed is a *minimal* explicitly parallel language that abstracts over the abilities of all current and foreseeable MIMD machines, and over resource management policies, and *no more*. The minimality is a concrete criteria: the compiler for this language should not make any "optimizing decisions" and so should not require any significant static analysis.

R2 *High-level:* The system, in addition to providing portability, should also take over the jobs of scheduling and mapping from the user, whenever possible. We have excluded automatic decomposition from the high-level requirements because the state of art in automatic decomposition (or its counterpart, automatic composition and automatic granularity control from implicitly parallel programs), including static analysis and compiler restructuring, is not adequate at this point. In addition, it is unclear if it would ever be feasible to produce a system that will automatically produce near an optimal decomposition. In some cases this may depend on new languages, which may face difficulty in being accepted by the user community. In future, if such automatic decomposition technology becomes available, it can be well supported by a system that automatically handles the task of scheduling and mapping.

A high-level, yet general purpose, approach has a further advantage. In the absence of domain specific knowledge, it provides an appropriate division of labor between the programmer and the system: the programmer specifies the decomposition of the computation into parallel actions, while the system can implement resource management and scheduling effectively.

Charm was designed to fit this niche. It provides more substantial support for parallel programming than the simple "portable mechanism" oriented systems. In addition, as shown in Figure 2, it can be used as a common base language for implementing other specialized high level languages, systems and packages, simplifying the task of developing them. Note that in order to support classes of approaches numbered 2 and 3, such a system must allow the programmer to over-ride its automatic

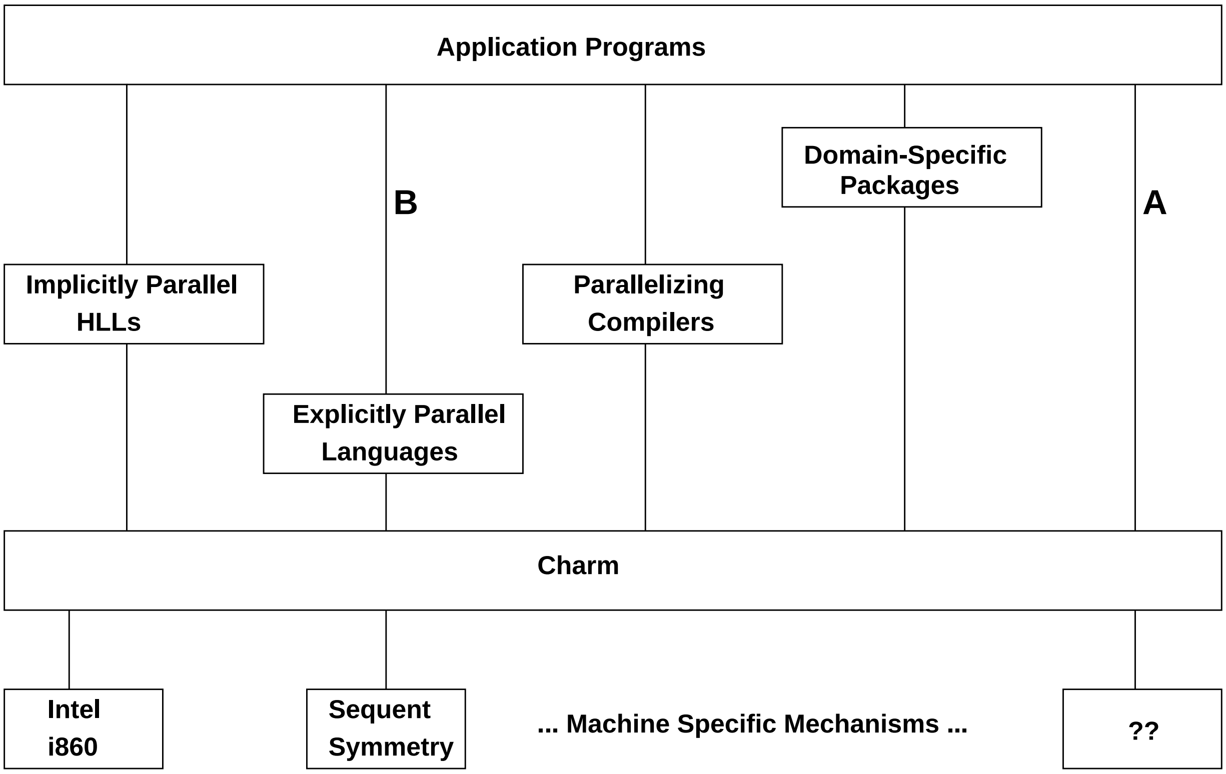

Figure 2: Intended uses of the Charm parallel programming system.

mapping and/or scheduling functions. Such an over-ride is also necessary to give the programmer the upper hand in cases when they can do a better job of mapping/scheduling based on their knowledge of the application.

A general-purpose, high-level parallel language should also satisfy the following requirements:

R3 *Efficiency:* The system must allow for the design and implementation of *efficient* parallel programs. There are two important efficiency considerations that the language must address: latency of communication and efficient portability.

The latency of communication — the fact that remote data will take longer to access than local data — is, and is likely to remain, a constant part of parallel processing hardware. Moreover, responses from a remote processor may encounter unpredictable delays due to ongoing computation on that processor. The language must provide features so that users can write programs which tolerate or hide such communication latencies.

For efficient portability, the primitives in the language should have a well defined cost model. Only with such a cost model can a programmer hope to write efficient and portable programs. So, the cost of the system primitives should not vary significantly from machine to machine. Nor should the compiler restructure the user code for optimization, making the cost of primitives unpredictable and requiring the user to second-guess the compiler[2].

Since the system is meant to abstract over shared memory and distributed memory machine capabilities, it should provide mechanisms that ensure efficient implementation of algorithms on both kinds of machines. An algorithm expressed in this language must run without change on both kinds of machines, yet be competitive with any implementation of the algorithm that directly uses the machine's capabilities.

[2]Note that we are not arguing against various structuring compilers for *high-level* languages.

R4 *Expressiveness:* A parallel computation consists of many asynchronous computational tasks that produce and share information with each other, and create new computational tasks. There are two aspects to expressiveness of a parallel programming system: the ability to express *task decompositions* and the ability to express *information sharing* between tasks.

The language we define must allow specification of such computational tasks, and allow for their dynamic creation. The mapping of these actions to processors and their scheduling should be automatically handled by the system, unless the user specifies otherwise.

Many explicitly parallel languages are based on a *universal information sharing* mechanism: a single mechanism that is deemed appropriate for various modes of information sharing. It is much more intuitive for the programmer to specify a particular information sharing mode, than to fit it into the single abstraction provided by such a "universal" mechanism based language. The language must provide *specific* abstractions to support various modes of information sharing.

R5 *Modularity and reusability:* Reusability is of special concern in parallel programming because of the variety of context assumptions that may be built into modules, particularly regarding data distributions. A matrix multiplication module written with the assumption that both matrices are replicated on every processor will not be usable in a context where they are distributed by rows, for example. The system must support features that permit writing of context independent modules. It should be possible to link together separately compiled modules, which are available in binary form only. Finally, to protect the investment in existing software (and sequential compilation technology), the system must support direct reuse of sequential subroutines.

The Charm language described below is designed to satisfy these requirements. The basic features of the language, including chares (tasks or objects) and messages, are described in Section 2. Chares are dynamically created and allow for expressing computations where dynamic task creation is necessary. They can be automatically mapped and scheduled, thereby providing the user with high-level capabilities. The message driven execution model of Charm and the rationale is described in Section 3. Section 5 discusses how specific information sharing mechanisms in Charm provide more expressive and efficient ways of sharing information across a multitude of parallel programming architectures. An additional kind of Charm process, called branch office chare, aimed at satisfying generality and modularity requirements, is described in Section 6. Section 7 provides a discussion on how Charm supports modularity. Additional features of Charm, such as conditional packing, quiescence detection, and prioritization, are discussed in Sections 8, 9, and 10. Section 11 describes the cost model for the different system primitives. The paper concludes with Section 12, which includes a comparison with some other systems, including those that occupy the same niche as Charm.

2 The Charm Base Language: Chares and Messages

A parallel program includes many sub-computations that are sequential in nature. There is no reason to invent a new language for expressing these sub-computations. We chose to employ C as the base language for Charm[3]. As a consequence, a Charm program may include function and type definitions as in C. In addition, any sequential threads of control in the parallel constructs must be expressible in C. With this decision, it becomes possible to retain large portions of sequential

[3]A C++ based version of Charm, called Charm++, was recently designed and implemented by Krishnan and Kale [32].

application code while parallelizing them. C is also a pragmatic choice as it is available on all parallel machines, and has an acceptable performance.

Next, we must define the parallel constructs in the language. Parallelism entails the existence of multiple focii of control. So we need a construct that captures the notion of a focus of control. One possibility is to associate each processor with one focus of control. This leads to the "one process per processor" view which is supported by most vendor supplied software. However this conflicts with requirement R4, which stipulates that dynamic creation of work be allowed. In addition, as stipulated by requirement R2, the user need not have to specify mapping of work to processors. Therefore we choose to separate the notion of a processor from the construct that encapsulates a focus of control. We call this construct, which specifies data and computation that will be mapped as a unit to a single processor, a *chare* — for chore or a small task. There may exist zero, one or more chares per processor at one time.

The chares will need to exchange information with other chares. A common mode of information exchange occurs when a chare produces data that is needed by another chare. This mode is supported by the notion of a message, which is a directed communication from one chare to another. Syntactically, a message is defined to be a collection of data, and in Charm it has the same syntax as that of a C structure declaration (see Figure 7).

A chare is allowed to handle multiple messages addressed to it: a separate section of code within a chare handles each incoming message. One possible way of specifying such sections is to require the programmer to provide a single function for handling each *type* of incoming message. However in different contexts and in different phases of its lifetime, a chare may have to deal with messages of the same type in very different manners. So we provide the notion of a named *entry-point* function. An entry-point has a single message type and an arbitrary C-code block associated with it. With this it is possible to have a single message type associated with many distinct entry-points. Note that this is directly analogous to the notion of methods in object-oriented programming.

When a chare sends a message to another chare it directs the message to a specific entry-point. The code in the entry-point function, which is triggered by the message, may access the fields of the message and the local variables of the function. Each chare instance may also have local variables that can be accessed from all of its entry points. The code at different entry-points may need to execute similar or identical computations. Such computations can be expressed as *private functions*. Private functions can be called only from within a chare, and they, like entry-points, can access the local variables of the chare. A chare is very similar to an object. It provides data encapsulation. However, chares in Charm[4], do not provide other object attributes, such as inheritance and polymorphism.

An example of the syntax of a chare definition appears in Figure 7. A chare definition includes the declaration of its data area (local variables), followed by declaration of a sequence of entry-point definitions. Each entry-point definition consists of an entry-point name, followed by a declaration of a message associated with it, and a block of C code. This block may contain arbitrary C code, including function calls. In addition, it may contain calls to Charm primitives, such as those described in Section 2.1.

2.1 Basic system calls

A Charm program, as defined so far, is a collection of chare *definitions*. At runtime, a single instance of a special chare, called *main*, exists. Execution of a Charm program begins with the

[4]Therefore, following Peter Wegner's [33] terminology, Charm can be thought of as an object-based system. Note that Charm++, the C++ based version supports inheritance and virtual functions.

creation of an instance of the main chare and the execution of a special entry function in the main chare called *CharmInit*. Other new chare instances can be created (from inside the main chare or other subsequently created chares) using the CreateChare call. It takes as parameters the name of the chare that is to be created, the entry-point to which the message is addressed, and a pointer to a message of the type associated with the entry-point. An entry-point is a variable of type *EntryPointType*. It can be specified as chare_name@entry, which refers to the entry-point entry in the chare chare_name. If the CreateChare call is made inside chare_name, then the entry-point could just be specified as entry. Note that an entry-point is a first class object. One can set a variable, say entry_name, of system defined type EntryPointType as: entry_name = chare_name@entry.

The CreateChare primitive, like all other Charm primitives, is non-blocking. From the programmer's viewpoint, the call deposits a *new-chare* message in a pool of such messages, and immediately returns. Eventually, a chare instance is created on some processor under the control of the runtime system of Charm (called the *Chare Kernel*). As soon as the chare is created, it executes the message using the code at the entry-point named in the creation call.

When a chare instance is created, it is identified by a unique *chare-id*. This address[5] of a chare instance may be obtained by the system call MyChareID(&chare_id), where chare_id is a variable of type *ChareIDType*. The addresses of chare instances can be then sent as fields in messages to other chares, which can pass them along to other chares, and so on. A message can be sent from one chare instance to another using the SendMsg call. It takes as parameters a pointer to a message, the address of the destination chare, and an entry point where the message needs to go.

3 The Message Driven Programming Model

In traditional message passing, a processor, after issuing a request for a receive, must idle until the specified message arrives. This waiting may not always be dictated by the algorithm, *i.e.*, the algorithm may have more relaxed synchronization requirements. This is particularly true for global operations, such as reductions. Yet the use of blocking primitives forces unnecessary synchronization and may cause idle time. This idle time can be decreased by moving the sends earlier and postponing the receives as much as possible in the code. In many cases, such local rearrangement of communication can increase the utilization of processors. However, this strategy cannot handle cases with more complex dependences and unpredictable latencies, nor can it handle global operations [34].

In accordance with our latency tolerance requirement (R3), we want to avoid the idling of processors under this condition. This is accomplished in Charm by allowing multiple chares to exist on each processor and by employing a **message driven** [6] execution model. In this model:

1. A chare is scheduled for execution only when there is a message available for it. Unlike processes, chares are neither perpetually executing, nor is their execution time-sliced on processors. A chare is always ready to execute any available message directed to it. Now, the likelihood of unnecessary processor idling is minimized: as long as there is a message available, a processor will schedule the execution of the chare for which the message is intended.

2. The execution of an entry-point code is *atomic*: it completes without interruption on the scheduled processor — there is no time-slicing, interrupt, or preemption.

[5]We use the term *address* in this paper to denote an abstraction of the notion of a reference, pointer, or a chare identifier. It does not denote the physical memory address of the chare.

[6]Active Messages represents a more recent effort for supporting message driven execution model; a comparison is provided in Section 12.

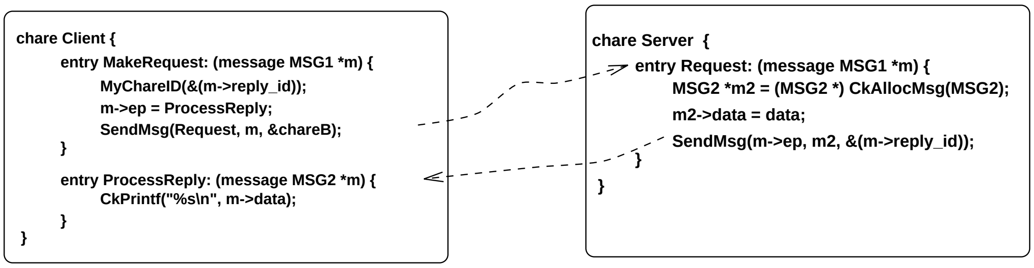

Figure 3: This figure illustrates the split-phase programming style.

3. No two entry-points of the same chare instance can execute concurrently.

Conceptually, one can think of the system operating with a pool of messages[7], a collection of passive chares, and a collection of active processors. Each processor may pick a message from the message pool, identify the chare that it is meant for, and schedule the execution of the code at the entry-point designated by the message. Note that there is no receive call, synchronous or asynchronous, in Charm. In fact, there are no calls that allow chares to interact synchronously with chares on *remote* processors. For efficiency and convenience, Charm does provide local synchronous calls between chares through *public* function calls.

The message driven programming model and the absence of any non-local synchronous primitives, such as receive, in Charm leads to a unique style of programming called "split-phase" or "continuation-passing". In this style, a synchronous request must be split over into two code blocks: the first block issues the request and provides the servicer of the request with the address of the second block of code to which the reply must be sent. The second block of code is activated whenever the reply arrives and must therefore be capable of processing the reply. A simple example in Figure 3 illustrates the split-phase programming style. Let A and B be instances of the client and the server chares, respectively. Chare A needs some data from Chare B, so it sends Chare B a request message for the data, along with the address to which the data must be returned: in this case it is another entry point (ProcessReply) in Chare A. Chare B, on receiving the request, sends back the data to Chare A at the specified entry point, which is invoked and handles the data when the message arrives. Notice that Chare A suspends after the entry point MakeRequest is executed, and is reactivated with the arrival of the reply from Chare B. The request is therefore *split* across two phases: make request and receive data.

A message-driven execution model allows one to adaptively schedule computations within and across modules as illustrated below. A more detailed exposition of the advantages of message driven execution can be found in [34].

3.1 Latency tolerance within a module

Consider the following example abstracted and modified from a real application — a core routine in parallelized version of a molecular mechanics code, CHARMM. Each processor has an array A of size n. The computation requires each processor to compute the values of the elements of the array and to compute the global sum of the array across all processors. Thus, the i^{th} element of A on every processor after the operation is the sum of the i^{th} elements computed by all the processors.

In the SPMD model, this computation can be expressed with a single call to the system reduction

[7] In reality, an implementation may maintain multiple pools, segregated by the type of message or the processors on which they may execute.

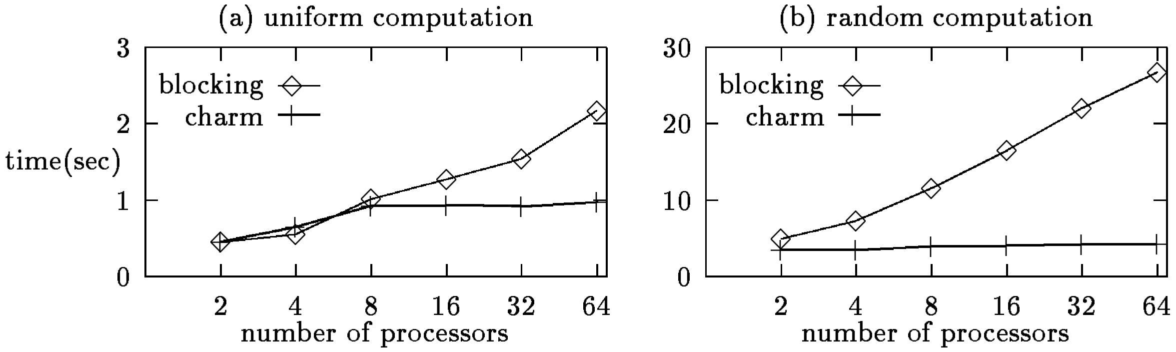

Figure 4: Concurrent Reductions, k=160,n=40960

library (e.g., `gssum`, on an Intel machine) preceded by the computation of the array on every processor. Alternatively, one can divide the array A into k parts, and in a loop, compute each partition and call the reduction library for each segment separately. Each call to the reduction library is a blocking one, i.e. the code cannot initiate the local computation belonging to the block before receiving the result of the current reduction.

However, the computations for each partition are completely independent. In particular, computation of the next k items (i.e., the next partitions) are not dependent on the result of the reduction, and so could be started even before the reduction results from the previous partitions are available. With this (message driven) strategy, a process that has just finished computing a partition is willing to either process the result of the reduction of any previous partition or compute the next partition.

This computation was programmed in C (using the Intel supplied native reduction library) for the blocking SPMD version and in Charm for the message driven version, and run on an Intel/Paragon machine. Figure 4-(a) shows the performance results of the case k=160, n = 40960, and up to 64 processors. The effect of pipelining of reductions in Charm is apparent from the flattening of the curve beyond eight processors. The increase up to eight processors with Charm can be attributed to the increase in the branching factor of the spanning tree used by the reduction library.

Each processor in the above experiment did a fixed amount of computation per element of the array before calling the reduction. In real applications, this computation is likely to vary from processor to processor. Figure 4-(b) shows results of the computation for the same parameters but with a random amount of computation added to each partition. The performance benefits of message driven execution become more significant when there exist irregularities in the computation. The blocking version makes every processor wait at a barrier for the last processor to arrive at the barrier, thus making the completion time the *sum of maxima* (for all partitions) as opposed to the *maximum of the sum* for the message driven version.

3.2 Latency tolerance across modules

The message-driven paradigm allows different modules that might have some concurrent computations to share processor time. Consider the computation (taken from Figure [34]) shown in Figure 5: module `A` invokes two other modules `B` and `C`. In the SPMD model, module `A` cannot activate `B` and `C` concurrently even if the computations in `B` and `C` are independent of each other. As a result, the processor time is not fully utilized, as illustrated in the same figure. In a message-driven paradigm, the idle times on a processor can be utilized by another module if it has some work to do. Such a scenario is illustrated in Figure 6 (taken from [34]). Module `C` gets processor time (by virtue of having its message selected by the scheduler) while `B` waits for some data, and vice versa, thus

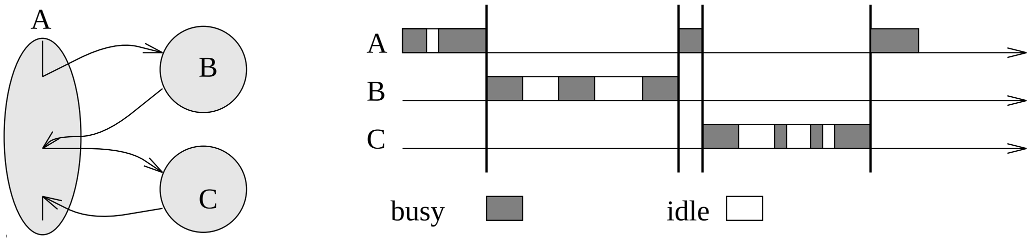

Figure 5: SPMD modules cannot share the processor time.

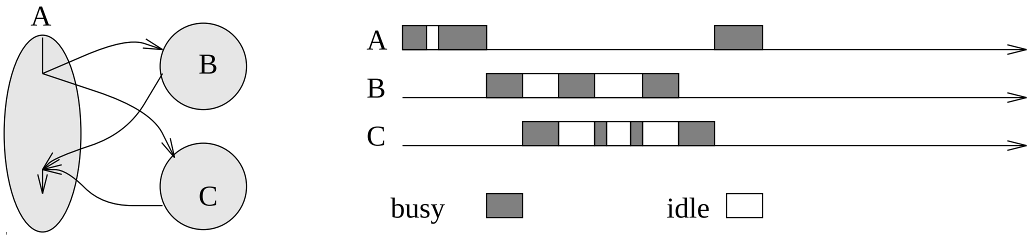

Figure 6: Message-driven modules share the processor time.

achieving a better overlap than the SPMD program.

Libraries constitute an important part of the software development process. They provide reusable, portable code, and they hide details from application programmers. There are many SPMD parallel libraries for commonly used kernel operations, such as numerical solvers, FFT, etc. The SPMD style does not encourage use of multiple concurrent libraries. When faced with performance loss in a situation, such as in Figure 5, an SPMD programmer typically breaks the library abstraction, combines modules B and C with A, and then tries to achieve better overlap by static movement of code augmented by "wild-card" receives. On the other hand, message-driven style encourages creation of smaller and more reusable modules. Therefore, we expect libraries to be a major strength of message-driven systems in the future.

4 Example Program 1

A simple Charm program is shown in Figure 7. In this program, a number of new chares are created in CharmInit. The user input is read using the CkScanf call, which is similar to *scanf* of C. The message that is sent to each chare is allocated using the CkAllocMsg system call, and contains the value of the seed, the address of the main chare, and some user-defined common data.

Each new chare is eventually mapped and scheduled by the runtime system on some processor selected by it. Thereafter, upon creation, each chare instance calculates, in the Start entry-point, a value which depends on the seed and the common data. This value is returned in a message to the Return entry-point of the main chare, where the values calculated by all chare instances are added. Messages are deallocated using the CkFreeMsg system call. The ChareExit call is used to signal to the system that a chare has completed execution. This call results in the de-allocation of memory occupied by the chare.

Once all chares have returned their computed values to the main chare, the aggregate value is printed

```
message { int seed; CharelDType parent; DataType data[SIZE];} DownMsg;
message { int value;} UpMsg;

chare main {
    int i, j, n, total; DataType data[SIZE];
    entry CharmInit: {
        DownMsg *m;
        CkScanf("%d",&n);
        read_in_data(&data);
        for(i=0; i<n; i++) {
            m = CkAllocMsg(DownMsg);
            m→seed = i;
            for (j=0; j<SIZE; j++) m→data[j] = data[j];
            MyChareID(&(m→parent));
            CreateChare(compute, compute@start, m); }
    }

    entry Result: (message UpMsg *result) {
        total + = result→value;
        CkFreeMsg(result);
        if (−−n ==0) { CkPrintf("The final Total is: %d", total); CkExit();} }
}

chare compute {
    entry Start: (message DownMsg *m) {
        UpMsg *up = CkAllocMsg(UpMsg);
        up→value = calculate(m→seed, m→data);
        SendMsg(m→parent, main@Result, up);
        CkFreeMsg(m);
        ChareExit(); }
}
```

Figure 7: A Simple Charm Program.

out using the CkPrintf system call. This call is similar to the *printf* call, except it is guaranteed to be atomic, i.e., multiple CkPrintf calls from different chares will not be garbled together. The main chare signals that the program has completed execution by calling the CkExit system call. This call results in the termination of all Charm processes, and the collection of performance and debugging data that the user might have requested.

5 Information Sharing Abstractions

The primitives described in Section 2 permit two forms of information sharing:

- An entry-point may use information generated by another entry-point belonging to the same chare, via the local variables of that chare.
- Information produced by one chare can be sent to another in a message.

In principle, these mechanisms are sufficient to express any information sharing needs. However, a single mechanism to share information is not sufficiently expressive (see requirement R4). It is much more intuitive for the programmer to specify a particular information sharing mode, than to fit it into the single abstraction provided by such a "universal" mechanism based language. A pure actor program allows messages as its only information sharing mechanism. Therefore one must use another actor to implement read-only sharing. A C-Linda program must use shared tuple space to send data created by a computational action to another. Of course, a good actor compiler may detect such "read-only" use of an actor and optimize it with a "read-only variable", but that depends on how good the compiler is, and it may not even be possible in some cases to detect it. Linda's tuple analysis faces similar hurdles. More important, such "universal" primitives prevent the programmer from providing this information, which they could have easily given.

Charm supports five specific modes in which information can be shared by chares. Each mode is provided as an abstract data type (ADT). Variables of each such type can be created statically (by initializing them inside the CharmInit entry-point of the main chare) or dynamically (any time during the execution of the program), and can be accessed and mutated only via the defined functions of the corresponding ADTs. Each of the ADTs may then be implemented by the runtime system differently on different machine architectures, thereby ensuring that the abstractions provide efficient portability.

ReadOnly variables: In some computations, many chares need read access to values that are created at the beginning of the computation (but are not known at compile time), and are not altered thereafter. Such information sharing can be specified by declaring a variable to be readonly in the declaration section. of the program. These variables can be assigned values only in the CharmInit entry-point of the main chare. They can be accessed from any chare using the ReadValue(variable_id) system call, which simply returns the value of the variable.

In the sample program in Figure 7, the data needed to calculate the value (in compute) was initialized in the CharmInit entry point. This data was passed to all the chare instances in the message; a more efficient implementation would be possible with the *readonly* abstraction. The common data could be declared as a readonly variable, initialized using the ReadInit(data) call, and accessed using the ReadValue(data) call. Messages would no longer carry the data for every chare that is created. Write-once variables are the dynamic counterpart of readonly variable; they can be initialized from anywhere in the program.

Distributed table: A distributed table is a set of entries, where each entry is a "record" with an integer key, and an arbitrary (untyped) data field. A special data-type called table is defined by Charm. One may declare many different tables of this type in a program. Distributed tables are accessed and modified only via the three calls: Insert, Delete, and Find. Unlike read-only and write-once variables, for which the access is synchronous, and immediate, accesses to the entries in the table are all asynchronous. Thus a call to find the data corresponding to a given key does not return with the data. Instead, it deposits a request to send the data to a given chare at a given entry-point.

Accumulators: Consider a computation in which many dynamically mapped chares are sending messages to each other, and we would like to count the total number of messages generated during the entire computation (or during a particular phase of the computation). Of course, each chare could store its own count, and send this count to a *counter* chare before it terminates. However, this method is not desirable because it is not scalable — with a large number of chares running on many processors, the "counter" chare will become a bottleneck. As this type of information sharing requirement is quite common, Charm provides the accumulator abstraction. More generally, an accumulator object has a commutative-associative operator as the sole mutator, and an identity element with respect to this operator.

We define the data associated with the accumulator as a message (this also facilitates conditional packing of the data; see Section 8). In addition to the data, the definition of an accumulator requires three user defined functions: one for initializing the accumulator data (initfn, one for "adding" to the accumulator (addfn, and one for combining two copies of the accumulator (combinefn, if needed. The accumulator abstraction is defined by three calls: CreateAcc, Accumulate, and CollectValue. The accumulator may be created in the *CharmInit* entry-point of the main chare by using the call: CreateAcc(ACC_TYPE, msg). This call creates an instance of the accumulator, and initializes it by calling the initfn with the given message as a parameter. The call returns a unique address (of type AccIDType) for the accumulator. This address can be used to access and modify the accumulator in the rest of the program. The address can be sent in messages to other chares, or it can be assigned to a readonly variable, so all chares can use it. The accumulator may also be created dynamically from any chare using the variant of the call CreateAcc(ACC_TYPE, message, entry, chare_id), which signals the runtime system to send the address of the new accumulator instance to the named entry-point (entry) of the designated chare (with the chare_id), after it has been created.

Any chare which knows the address of an accumulator instance (say *accid)* may "add" to it by using the call: Accumulate(accid, addfn(..)). The final value of an accumulator can be read by calling CollectValue(accid, entry, chare_id), which returns immediately without any value, but a message is eventually sent to the entry-point entry of the chare instance designated by chare_id, containing the final value of the accumulator. The CollectValue call results in destruction of the accumulator variable. Hence the call should be used only once, and only when one is sure there are no more Accumulate operations possible on that variable.

The user program never calls the *combineFn()* function explicitly. It is used by the runtime system in case it has made multiple copies of the accumulator for efficiency on the target architecture, e.g. in a non-shared memory machine implementation. (See part II of this paper.)

Monotonics: Sometimes many chares need to read and update a shared variable, but the update operation is idempotent (i.e. repeated application of the same update operation are equivalent to one update operation) as well as commutative-associative, and the variable successively takes on monotonically "decreasing" values in some metric. In branch-and-bound computations, such a variable is needed to store the cost of best solution known so far. Every chare needs to know what the current best bound is, and when someone finds a new solution, the best bound *may* have to change to a smaller value, if the new value is smaller. Such information sharing is specified by declaring the variable as a monotonic variable.

The monotonic data abstraction is defined by three calls: CreateMono, NewValue, and MonoValue. The CreateMono call has the same set of parameters as the CreateAcc call, and works in an identical manner. A new value can be deposited into a monotonic variable by using the call: NewValue(monoID, updateFn(args..)). An upper bound of the current value of a monotonic variable can be obtained by the call: MonoValue(monoID). The value returned by the MonoValue call will be either the value assigned during initialization, or provided thereafter by some NewValue call, and be better than or equal to the best value provided by a NewValue call by the same process. In addition, the system will make efforts to provide the best value of the monotonic variable supplied by any NewValue call until that point in time.

6 Branch Office Chares: Replication and Sequential Interface

The programmer's model of a Charm computation, as described so far, includes chares that may dynamically create other chares, send messages to each other, and share information via other specifically shared "global" variables. Note that the "processor" is not a part of the ontology so far. We now introduce a construct that brings in the notion of processors.

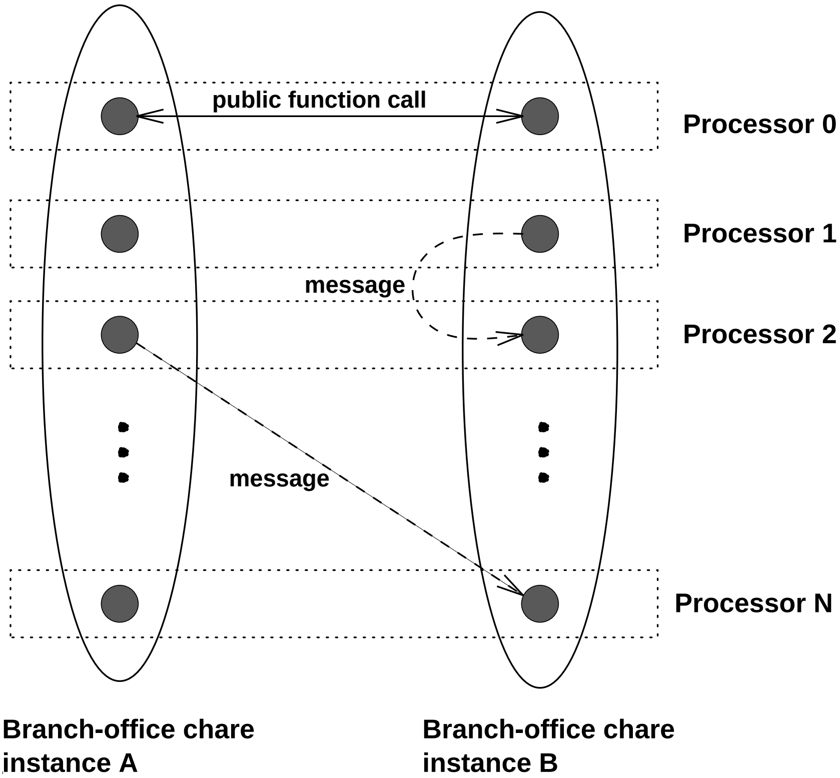

Figure 8: This figure shows two instances of a branch office chare, and how different branches can interact with others using public function calls if they are on the same processor or messages if they are on distinct processors.

In many parallel applications, similar work is done by each processor. Such an application could be programmed using chares, where one chare is created on each processor. However substantial initial bookkeeping is required to make sure that each chare knows the address of a chare on a particular processor. As this is a common case, we prefer to have more convenient way of expressing it using a new construct: *branch office chare* (BOC), which is a replicated process. A branch chare of the BOC exists on every processor. However all the branches of an instance of a branch-office chare can be referenced by one name.

A chare and a branch of a BOC on the same processor, or branches of two BOCs on the same processor could interact with each other using messages. However messages have overheads of creation and scheduling. Such overheads can be eliminated with a synchronous, sequential function call interface between the local "branches" of the application and other chares. Therefore, in addition to receiving messages at entry-points like chares, BOCs also provide *public functions*. A chare (which happens to be running on some processor under the control of the dynamic load balancing strategy) may interact with the local branch of a BOC via a sequential *public function* call. This combination of features makes the branch office chare a versatile and useful abstraction. Figure 8 shows two branch office chares, and how different branches can interact with others using public function calls if they are on the same processor or messages if they are on distinct processors.

The syntax of a branch-office chare is similar to that of a chare, and an example is shown in Figure 9. A BOC declaration consists of its data area (local variables), entry point definitions, and definition of private and public function calls (identical to those of a chare).

The CreateBoc system call is used to create an instance of a branch office chare. It takes as parameters the name of the BOC to be created, a creation message, and the entry point to which the message is addressed. BOCs can be created statically from the CharmInit entry-point in the main chare. In this case, the call returns the address of the new branch-office chare. BOCs can also be created dynamically in the middle of a computation from any chare. In this case, the last two parameters are required, and the address of the BOC is sent to the chare identified by chare_id, at the specified entry-point entry2.

In the following discussion, the address of a new branch-office chare instance is denoted by the variable boc. Different branches of the same or different BOCs can communicate with each other using the SendMsgBranch call. A branch can send a message to all other branches using the BroadcastMsgBranch call. Chares and other BOC branches on a processor may call a public function fn of a BOC identified by boc, which is an instance of a BOC named boc_name, using the BranchCall system function.

Branch-office chares can be used in a variety of contexts:

1. Static Load Balancing: Even without its public functions, the fact that the BOCs are replicated processes allows them to be used for simple statically mapped SPMD style programming. In comparison with the traditional mechanisms for supporting SPMD style programming, such as Express [4] or the send/receive primitives of the native operating systems on the distributed memory machines, BOCs provide the advantages of message-driven execution. Also, one may have as many BOC instances in a single program as needed. This separates and simplifies the flow of control in many programs.

2. One can build new dynamic (i.e. time varying) and distributed data structures using the BOCs. The public functions provide a sequential function-call interface to such a data structure, while messages between branches are used to coordinate the distributed data structure.

3. BOCs can be used to implement and provide local "services" of various kinds. For example, the memory manager in the Chare Kernel (the runtime support system for Charm) is written as a BOC, and so is the message-receptionist in the parallel implementation of Actors reported by Agha and Houck [35]. In this use, the BOC does not send any messages to other branches, but simply acts as a local sequential object, with a globally unique instance address.

4. BOCs can also be used to implement services that require communication. For example, various dynamic load balancing strategies and the quiescence detection algorithm, provided in the chare kernel are written as BOCs. Also, Charm libraries, such as the library for various reductions, are written as BOCs. Again, the BOC uses the sequential interface via functions to deposit local information, and communication with other branches to implement the functionality of the services.

Figure 9 shows a simple program written using a BOC: a message is sent around in a ring on the available processors starting at processor 0. The BOC is created using the **CreateBoc** call in the **main** chare.

7 Modularity and Reuse

Supporting modularity and reuse is more difficult in a parallel context than in a sequential context. First, there are a seemingly mundane set of issues that need to be addressed, such as the fact that pointers are not valid across address space boundaries. So, for example, one cannot pass a function-pointer to another module directly. Second, a reusable module must be able to work with a large variety of ways in which the input data is distributed among processors. Third, for

```
chare main {
    entry CharmInit: {
        MSG1 *msg; ChareNumType boc;
        msg = (MSG *) CkAllocMsg(MSG);
        boc = CreateBoc(Ring, Ring@Start, msg); }
}


BranchOffice Ring {
    int right;
    entry Start: (message MSG *msg) {
        right = (CkMyPeNum()+1) % CkMaxPeNum();
        if (CkMyPeNum()==0) SendMsgBranch(msg, Ring@Pass, right); }

    entry Pass: (message MSG *msg) {
        if (CkMyPeNum()==0) CkExit();
        else {
            PrivateCall(print());
            SendMsgBranch(msg, Ring@Pass, right); }
    }

    private print()
    { CkPrintf("Sending message to right neighbor %d", right); } }
}
```

Figure 9: A ring program.

scalability, modules should be able to exchange or pass data in a fully distributed fashion. When a module with entities spread over hundreds of processors, wishes to pass data to another module spread over hundreds of processors, one must ensure that the data exchange is not centralized. In this section, we describe the features in Charm that support modularity and reuse, and briefly discuss how these features achieve the objectives.

A Charm program is written as a set of *modules*. Charm supports separate compilation of modules. A module can contain names of chares, BOCs, C functions, messages, C type definitions, and specifically shared variables. These names are *internal* to the module. Modules can interact with each other by referencing *external* names (defined in other modules). An *external* name is referred to by specifying the module and the name being referenced. E.g., accesses to a function F, or a chare C, or an entry point E inside chare C which are defined in a module M would be made as *M::F*, *M::C*, *M::C@E*, respectively.

Each module has an *interface* statement that includes prototypes of all the names in the module that may be referenced by other modules. A module includes the *interface* statement of each module with which it interacts, as well as its own interface statement.

Names, external and internal, can be passed in messages or via functions to other modules. This mechanism can be used by a client to request a certain service from a server process defined in another module, such that the result of the server's computation is sent to a specified chare at a specified entry-point. This is very useful, because the server might have been written separately without any knowledge of the static entities or the names of possible clients. Similarly, consider a situation when a module M1 invokes M2, and wishes to have M2 call a function F defined in M1 possibly at some other processor than the one where the invocation occurred. Assume again that M2 was written independently of M1 (and so cannot contain an interface statement for M1). M1 cannot pass a pointer to the function F, because it will not be valid on the processor where F is to be invoked. Charm provides a mechanism for converting locally valid function pointers to globally valid function references for this purpose. Such references can be propagated in messages, and dereferenced via another system call when the function is to be invoked.

Charm supports distributed exchange of data among modules via BOCs (Section 6) and distributed tables (Section 5). A module may send data to another via a BranchCall carried out on all the processors. Alternatively, two modules may exchange data via a distributed table, thus obviating the need for either module to know where the particular data items are located. Thus, BOCs and tables act as a "glue" to interconnect modules through distributed interfaces. Finally, as illustrated in Section 3.2, message-driven execution also allows Charm modules to be composed efficiently.

8 Conditional Packing

The data structure to be passed in messages may sometimes be large and complex. Consider an array being passed in a CreateChare message. On a shared memory system, the message need only store a pointer to the base of the array and its size. However, on nonshared memory systems, a pointer is not valid across processors. So the whole array must be copied in each message. A chare may also want to send a dynamically created data-structure, such as a graph or a tree, which uses pointers. Again, the data structure must be copied (or "packed") into a contiguous structure without pointers before it can be sent in a message.

Charm encourages a programming style that counters the unpredictability of available work by creating many small chares in the hope of being able to distribute them as needed. So in a message passing system, each processor, typically, creates many chares that are not actually sent out to any other processor, but executed locally. When the system is in saturation (all the processors have sufficient work), this happens to most chares. This state of affairs is desirable because it offers the flexibility of responding to load fluctuations as they arise. Now, however, packing each message in a format suitable for across-processor transmission seems quite wasteful. Also, such packing is unnecessary and wasteful on shared memory machines. It would be better to pack only those messages that actually leave address-space boundaries, leaving other messages free to contain pointers. However, the programmer doesn't know which one of the created chares (or messages) will end up going to another processor, as it is the decision of the dynamic load balancing strategy. Charm, on the other hand, cannot know how to pack messages since their structure is known only to the corresponding application code.

Charm provides an interesting solution to this problem. It allows messages to contain pointers. However, if a message needs to move across address space boundaries, the kernel calls the appropriate code in the user program for packing the message into a contiguous space and eliminating explicit pointers. Conversely, before a message received from outside the address space is scheduled for execution, the system calls another entry-point to unpack the message. The functions for packing and unpacking are provided by the programmer along with the definition of the message-type.

Thus, only those messages that are actually sent out are packed, and the rest of the computation can proceed as it would on a uni-processor or a shared memory machine, using pointers to represent data structures efficiently. This feature is instrumental in satisfying the requirement R3 about ensuring competitive efficiency on shared memory machines. As experimental evidence, conditional packing led to three-fold improvements in speed on iPSC/2 for a parallel prolog interpreter [36] implemented using Charm.

```
message { int a; varSize float b[]; } MSG;
...
{
    MSG *msg; int i, n, sizes[1];

    CkScanf("%d", &n);
    sizes[0] = n;
    msg = (MSG *) CkAllocMsg(MSG, sizes);
    for (i=0; i<n; i++) CkScanf("%f", &(msg->b[i]));
}
```

Figure 10: The declaration and allocation of a variable size message.

The special case of variable size arrays occurs very frequently in many applications. Charm provides a *varSize* array field in message definitions for this special case. When a message with varSize fields is allocated, the user must specify the sizes of all the varSize array fields in the message. The system then handles the packing and unpacking of this messages automatically. Figure 10 shows the declaration of variable sized message-type MSG. The *CkAllocMsg* call takes an additional parameter (sizes), which is an array whose elements are the sizes of the variable sized arrays in the message.

9 Quiescence Detection

In some computations, it is useful to know when there are no more messages in the system. Charm allows a user to specify an entry function of a chare with address to which a message is sent when the system becomes quiescent. The user-defined code at that entry-point then decides the course of action. In simple cases, the action may be just to terminate the execution by calling CkExit(). However, other interesting uses are also possible. For example, a read-eval loop for languages such Prolog can be written by having the code at the *quiescence* entry-point read the next query and start its execution. In general, quiescence detection can be used to detect the end of a phase of computation which involves arbitrary and unpredictable amount of communication per processor. Quiescence detection has been efficiently implemented [37], so that (a) the condition is detected very quickly after it occurs, and (b) the overhead of the algorithm is very small.

10 Prioritized Execution and load balancing

In many computations, the order in which available tasks are selected for execution [38] can affect various performance metrics. In Charm, the programmer can assign priorities to messages; the message-queuing strategy chosen by the user will then schedule the highest priority member of the queue. The priority can be an integer, or an arbitrarily long bit-vector, depending on the queuing strategy option chosen. Bit-vector priorities are especially useful for obtaining good and consistent speedups in state-space search and related problems [39, 40] which involve speculative work. Integer priorities are useful in many seemingly regular computations which may have critical paths [41] that must be prioritized, particularly in the presence of message-driven execution.

Charm has many different load balancing strategies for message passing machines. Different strategies may be effective in different application-specific and sometime machine-specific contexts.

Therefore, Charm allows the user to link any one of the available strategies from its library, and to specify parameters for the strategy to tune it further.

Typically, for reasons of scalability and to avoid bottlenecks, it is desirable that load balancing strategies be distributed in nature. However, experimental results [42] have shown that existing fully distributed load balancing strategies do not balance priorities well resulting in the concentrations of low and high priority work neighborhoods. Therefore, in addition to distributed strategies, Charm also provides fully and partly centralized load balancing strategies. These load balancing strategies fare much better in balancing priorities than fully distributed strategies.

11 The Cost Model

Charm provides a relatively simple cost model to the programmer. The cost of various operations can be understood in terms of the cost of a message and the cost of a function call. The cost of a message has two components, one fixed and one a linear function of the message size, the proportionality constants vary somewhat from machine to machine.

The cost of BranchCall and PrivateCall is that of a sequential function call, and so is the cost of the following calls for accessing and updating specifically shared variables: ReadInit, ReadValue, DerefWriteOnce, Accumulate, MonoValue, and NewValue. The cost of the following calls is that of a single message: CreateChare, SendMsg, and SendMsgBranch. The cost of the distributed table operations Insert, Find, and Delete is that of two messages. There are a set of calls whose cost to the overall system is that of a *single* message per processor; in terms of critical path (i.e., the time between the call and the completion of the action) the cost can be considered to be *log p* messages, where p is a number of processors. The calls in this category are: BroadcastMsgBranch, CreateBOC, WriteOnce, and CollectValue.

It might be argued that the cost model is inadequate because it does not account for interconnection topologies, and the variation in communication latencies and processor speeds across machines. Clearly, for portable design of parallel programs, it is desirable to be able to ignore these machine-dependent features. But more important, with the current generation of parallel computers, it is becoming clear that the interconnection topology is not a significant determinant of performance. The end-to-end message delays on machines with advanced routing networks (such as wormhole routing) vary very little with the inter-processor distance. The network bandwidth *is* affected by the average communication distance, and for the rare set of problems where the communication bandwidth is a significant issue (over-riding processor utilization, say), the user may indeed have to refine the cost model to include the interconnection topology. The relative communication latencies are rendered unimportant due to the message-driven execution in Charm — as long as there is sufficient work on each processor, the actual network latency of messages does not affect the performance of an application. The only overhead the programmer needs to be concerned with is the per-message overhead incurred by Charm and the underlying operating system overheads while sending and receiving a message. This is a software overhead, and it scales at the same rate as the application code with variation in CPU speed. So, the user's cost model doesn't have to change due to variation in latencies and processor speeds, for most applications.

12 Discussion

One of the initial motivations we gave for developing Charm was to fill the niche identified in the spectrum of approaches to parallel programming shown in Figure 1. Charm certainly accomplishes this objective, as chares are dynamically load balanced, their execution is scheduled automatically with the arrival of messages, and Charm programs run portably and efficiently across a range of MIMD machines. How this efficient implementation is accomplished is described in part II of this paper. In addition to filling this niche, Charm provides a new parallel programming paradigm. This paradigm:

- Uses message-driven execution as its basic scheduling mechanism, to efficiently deal with communication latencies and delays in remote responses.
- Recognizes that information is shared in many specific modes in parallel programs, and employs several information sharing abstractions for this purpose.
- Employs multiple replicated "processes" with sequential as well as parallel interfaces — as embodied in the branch-office chares of Charm — as an important and useful program structuring device.
- Demands that different modules in a parallel program should be separately compilable, and that one should be able to compose them into large modules or applications without hindering performance, and in particular, without having to give up the performance advantages of message-driven execution.

Charm was intended as as a general-purpose high-level language, which could be used to support other language design efforts, as shown in Figure 2. This objective has also been attained as is substantiated by the fact that Charm has been used as a back-end for a data parallel language called DP [43], a parallel Prolog compiler [44], a high-level synchronization language called Dagger [45], a domain specific language called Divide-and-conquer [46], and an Actor language called Hal [35].

Charm is one of the first languages to employ message-driven execution in stock multi-computers [31]. The idea of message-driven execution is clearly implicit in earlier work on Data Flow machines, which depended on special-purpose hardware to support it. Special purpose hardware for message-driven execution was also the focus of projects such as the J-Machine [47] and Mosaic [48]. The work on macro data flow [49] has focussed on bringing these concepts on general purpose hardware (stock multicomputers).

The work on Active Messages [50] is more recent than Charm. In this model, a message interrupts the recipient process, and invokes the handler rountine specified in the message. Active messages only provides a low-level mechanism for writing message driven programs. For example, if a second message arrives while the first one is being processed, the user's handler code must handle it explicitly, possibly by buffering it. In contrast, in Charm, the runtime system buffers and schedules the messages automatically. Active Messages is a single-process based model, unlike Charm which supports multiple objects per processor. Active messages implementations carried out at the operating system level can deliver messages even faster than the vendor's send-receive primitives on some machines. In fact, the communication layer of Charm has recently been implemented on the CM-5 using Active Messages.

Split-C [51] is a programming language that provides a global data space, where accesses to global data are through unique split-phase operators, which separates the request for data from its use (this aspect is similar to "futures"). The primary differences with Charm are that it is also a single-process model and that it provides a global address space, and its primitives do not permit an adaptive overlap of computation and communication.

The observation that information is shared in many specific modes was also made independently in [52]. However, they used it for annotations and optimizations in a parallelizing compiler. The dataflow community also developed similar notions — often called "sideways" communication primitives — in the context of functional programs. In particular, their notion of accumulator is very similar to that in Charm with an important difference. In a functional program, it is trivial to ensure the safety of the read operation, whereas in a Charm program, the user must make sure that all potential operations that can add to the accumulator have terminated before accessing the final value of the accumulator. Distributed tables have similarities with the I-structures in dataflow to some extent, as also with the tuple spaces of Linda. In Linda, the accesses to tuples are blocking whereas in Charm an entry from a distributed table is accessed in a non-blocking split-phase manner. Moreover, tuples are the only information sharing mechanism in Linda, whereas tables are one of many in Charm.

Actors [53] , a construct proposed by Hewitt and developed by Gul Agha, embodied one of the early proposals for message driven execution. Each actor has a behavior associated with it. Actors do not issue "receive" statements, but execute only when triggered by a message. Thus the basic notion of chares has much similarity with Actors. Actors, however, permit further concurrency within a single actor, while Charm uses chares to define a boundary between parallel and sequential — only one method within a chare may execute at a time. The branch-office chare construct, and the use of information sharing abstractions other than messages further distinguish Charm from Actors. The Actor model provides a theoretical background that is applicable to a system such as Charm. One of the first implementations of the Actor model on stock multicomputers was carried out using Charm by Houck and Agha [35].

The notion of concurrent aggregates was developed at MIT by Chien and Dally [28, 29] at the same time that the branch-office chares were implemented in Charm [54]. Concurrent aggregates were designed for fine-grained machines, and were implemented in a simulator. The members of a concurrent aggregate are analogous to a branch of a branch-office chare, except that they do not necessarily have a member on every processor. So, calls to a concurrent aggregate may go to a remote processor. The BOCs, on the other hand, are explicitly designed to provide a local, sequential access to branches.

One of the important attributes of Charm is the richness and specificity of the constructs it provides. As a result, the Charm run-time system has a clear understanding of the events in the application program, at a level much closer to the application than the machine. For example, whereas a traditional SPMD system will be able to note that a message went from processor X to processor Y (along with its message type), the Charm run-time system can discern between messages for creation of new chares, messages to existing chares, messages for requesting and fetching data from distributed tables, along with information about the entry-points and chare instances at which a message is directed. This specificity can be exploited in a variety of ways for supporting parallel programming with Charm. In particular, performance feedback and debugging tools can be built that provide the user with application-level feedback allowing them to home onto the trouble-spots in their source program easily. A preliminary step in this direction is represented by Projections [41], a graphical performance display tool, which exploits the specificity only minimally by distinguishing between different kinds of messages.

The extensive support for modularity in Charm, the ability to compose modules without losing the efficiency of message-driven execution, and the mechanisms for distributed data-exchange across modules (provided by branch-office chares and distributed tables) makes Charm an excellent framework for developing flexible and reusable parallel libraries. We expect that this capability of Charm will be leveraged by us and others for developing libraries for various application areas in computational science and engineering.

References

[1] High performance FORTRAN forum. High Performance FORTRAN specification. Technical report, Rice University, Houston, TX.

[2] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna fortran — a language specification. Technical report, ACPC technical report series, University of Vienna, Vienna, Austria, 1992.

[3] Edward Kornkven and Laxmikant Kale. Dynamic adaptive scheduling in an implementation of a data parallel language. Technical Report 92-10, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, October 1992.

[4] J. Flower, A. Kolawa, and S. Bharadwaj. The Express way to distributed processing. In *Supercomputing Review*, pages 54–55, May 1991.

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

[6] 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.

[7] J. Dongarra et al. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7, No. 2:166–175, 1993.

[8] M. Snir, W. Gropp, and E. Lusk. Document for a standard message-passing interface: point to point communication. draft, 1993.

[9] A. Geist and M. Snir. Document for a standard message-passing interface: collective communication. draft, 1993.

[10] L. Clarke et al. Document for a standard message-passing interface: groups, contexts, communications. draft, 1993.

[11] D. H. Gelernter . Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[12] N. Carriero and D. Gelernter . How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, pages 323–357, September 1989.

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

[14] Taylor S., Safra S. and Shapiro E. A Parallel Implementation of Flat Concurrent Prolog. Technical Report CS87-04, Weizmann Institute of Science, October 1986.

[15] P. Hudak. Para-functional programming: a paradigm for programming multi-processor systems. In *12th ACM Symposium on Principles of programming languages*, pages 243–254, January 1986.

[16] P. Hudak. Para-functional programming. In *Computer*, volume 19, Number 8, August 1986.

[17] R. Halstead. Multilisp: a language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems*, volume Volume 4, No. 4, October 1985.

[18] R. P. Gabriel and J. McCarthy. *Qlisp*, pages 63–90. Kulwer Academic Publishers, Boston, 1988.

[19] *Qlisp: Experience and New Directions*, volume SIGPLAN Notices, V. 2, No. 9, September 1988.

[20] I. Mason, J. Pehoushek, C. Talcott, and J. Weening. Programming in qlisp. Technical Report stan-cs-90-1340, Stanford University, 1990.

[21] D. J. Kuck et al. The effects of program restructuring, algorithm change, and archictecture choice on program performance. In *International Conference on Parallel Processing*, pages 129–138, August 1984.

[22] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. In *Communications of the ACM*, volume Volume 29, No. 12C, December 1986.

[23] J. R. Allen and K. Kennedy. Automatic transformation of fortran programs to vector form. In *ACM Transactions on Programming Languages and Systems*, volume Volume 9, No. 4, pages 491–542, October 1987.

[24] R. Nikhil, K. Pingali, and Arvind. Id nouveau. Technical Report Computational structure group memo 256, MIT Laboratory for Computer Science, 1986.

[25] R. Nikhil. Id version 88.1 reference manual. Technical Report Computational structure group memo 284, MIT Laboratory for Computer Science, 1988.

[26] D. C. Cann, C-C Lee, R. R. Oldehoeft, and S. K. Skedzielwski. Sisal multiprocessing support. Technical Report UCID-21115, University of California, Lawrence Livermore National Laboratory, July 1987.

[27] D. C. Cann and J. Feo. Sisal versus fortran: A comparison using the livermore loops. In *Supercomputing*, November 1990.

[28] A. A. Chien and W. J. Dally. Concurrent aggregates. In *The second ACM Symp. on Principles & Practice of Parallel Programming*, pages 187–196, March 1990.

[29] A. Chien. *Concurrent Aggregates: an object-oriented language for fine-grained message-passing machines.* PhD thesis, MIT, July 1990.

[30] Akinori Yonezawa. *ABCL: an object oriented concurrent system.* The MIT Press, 1990.

[31] L. V. Kale and W. Shu. The Chare Kernel language for parallel programming: A perspective. Technical Report UIUCDCS-R-88-1451, Department of Computer Science, University of Illinois, August 1988.

[32] 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.

[33] Wegner, P. Conceptual Evolution of Object-Oriented Programming. In *Proceedings of OOPSLA*, 1989. keynote talk.

[34] 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.

[35] C. Houck and G. Agha. Hal: A high-level actor language and its implementation. Technical Report UIUCDCS-R-92-1728, Department of Computer Science, University of Illinois, September 1992.

[36] L . V. Kale and B. Ramkumar. Implementing a parallel Prolog interpreter on multiprocessors. In *5th International Parallel Processing Symposium*, pages 543–548, Anaheim, CA, April 1991.

[37] A. B. Sinha, L. V. Kale, and B. Ramkumar. Quiescence detection in dynamic process executions. submitted, 1993.

[38] L . V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Comp. Science*, pages 146–181. Springer-Verlag, 1993.

[39] V. Saletore and L. V. Kale. Obtaining first solutions fast in parallel problem solving. In *The North American Conference on Logic Programming*, pages 390–408, Oct 1989.

[40] L. V. Kale and V. Saletore. Parallel state-space search for a first solution. *International Journal of Parallel Programming*, 19:251–293, 1990.

[41] A. B. Sinha and L. V. Kale. A load balancing strategy for prioritized execution of tasks. In *Workshop on Dynamic Object Placement and Load Balancing, in co-operation with ECOOP's 92*, Utrecht, The Netherlands, April 1992.

[42] A. B. Sinha and L. V. Kale. A load balancing strategy for prioritized execution of tasks. In *International Parallel Processing Symposium*, New Port Beach, CA., April 1993.

[43] E.A. Kornkven. Compiling data parallel languages for multi-threaded execution. Technical Report 90-03-01, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana, IL 61801, April 1994.

[44] B. Ramkumar and L. V. Kale . Compiled Execution of the REDUCE-OR Process Model on Multiprocessors. In *North American Conference on Logic Programming*, October 1989.

[45] A. Gursoy and L.V. Kale. Dagger: Combining the benefits of synchronous and asynchronous communication styles. In *International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[46] A. Gursoy and L . V. Kale. High-level support for divide-and-conquer parallelism. In *Proceedings of Supercomputing '91*, pages 283–292, November 1991.

[47] W. Dally et al. The j-machine: a fine-grained computer. In *IFIP Congress*, 1989.

[48] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *Computer*, 21, No. 8:9–24, August 1988.

[49] A. S. Grimshaw. *Mentat: an object-oriented macro data flow system.* PhD thesis, University of Illinois, Urbana-Champaign, June 1988.

[50] T. vonEicken, D. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. *ACM*, pages 256–266, 1992.

[51] D. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. vonEicken, and K. Yelick. Parallel programming in split-c. Technical report, University of California, Berkeley.

[52] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *The second ACM Symp. on Principles & Practice of Parallel Programming*, March 1990.

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

[54] L. V. Kale *et al. The Charm Programming Language Manual (4.1)*, 1994.