# Converse : An Interoperable Framework for Parallel Programming

Laxmikant V. Kale and Milind Bhandarkar and Narain Jagathesan and Sanjeev Krishnan
Department of Computer Science
University of Illinois
Urbana, IL 61801
Email : {kale,milind,narain,sanjeev}@cs.uiuc.edu
Phone : (217) 244-0094
WWW : http://charm.cs.uiuc.edu

**Abstract**

Many different parallel languages and paradigms have been developed, each with its own advantages and niches. To benefit from all of them, it should be possible to link together modules written in different parallel languages in a single application. As the paradigms sometime differ in fundamental ways, this is hard to accomplish. This paper describes a framework, *Converse*, that supports such multi-lingual interoperability. The framework is meant to be inclusive, and has been verified to support the SPMD programming style, message-driven programming, parallel object-oriented programming, and thread-based paradigms. The framework aims at extracting the essential aspects of the runtime support into a common core, so that language-specific code does not have to pay overhead for features that it does not need.

## 1 Introduction

Research on parallel computing has produced a number of different parallel programming paradigms, architectures and algorithms. There is a wealth of parallel programming paradigms such as SPMD [9, 19], data-parallel [14, 13, 4], message-driven [15, 16], object-oriented [6, 24, 5, 10, 2], thread-based (Chant [12], CC++ [5]), macro-dataflow (P-RISC[21]), functional languages, logic programming languages, and combinations of these.

However, not all parallel algorithms can be efficiently implemented using a single parallel programming paradigm. It may be desirable to write different components of an application in different languages. It is also beneficial to combine pre-written modules from different languages into a new application. For this, we need to support interoperability among multiple paradigms. Such interoperability is not currently possible, except for a specific subset of languages designed together for this purpose (e.g. HPF and PVM).

This paper describes the design and rationale of *Converse*, an interoperable framework for combining multiple languages and their runtime libraries into a single parallel program. It is based on a software architecture that uses message driven execution and "thread objects" to compose

multiple separately compiled modules written in different languages without losing performance. Converse will also facilitate development of new languages and notations for specific purposes, as well as support new runtime libraries for these languages. This multi-paradigm framework has been verified to support traditional SPMD systems, thread-based languages, and message-driven concurrent object-based languages, and is designed to be suitable for a wide variety of other languages. Our initial implementation includes Charm, Charm++, DP-Charm (a data parallel language), PVM, NXLib, and SM (a simple messaging layer). The latter three will be supported both in SPMD as well as multithreaded mode.

The next few sections describe the rationale used in the design of *Converse*. Section 2 describes the model of computation the framework targets, and establishes a classification of parallel languages based on their control structures. Section 3 describes the architecture of Converse including its core components: a universal scheduler and a minimal machine interface. This section also describes components that support thread based programming. Section 4 illustrates how the interoperability provided by Converse can be utilized. Preliminary message passing performance of our implementations is presented in Section 5. Section 6 provides a summary and identifies areas of future work.

# 2 Model of computation

This section describes the general parallel computational model our framework supports. A computation consists of multiple communicating processes. A parallel program in this model consists of a set of parallel modules written possibly in different languages.

Languages and their implementations differ from each other in many aspects. The important aspects are:

1. How do they deal with concurrency within a process?
2. How does control transfer from one module to another?

## 2.1 Concurrency within a process

The aspect that is critical from the point of view of interoperability is how the language deals with concurrency within a single process (i.e. within a single processor, in the common implementations). Concurrency within a process arises when there is more than one action the process could take at some point(s) in time. There are three categories of languages in this context:

- No concurrency/Single Process Modules: Some languages, such as PVM, do not allow concurrency within a process. They require that the programmer of a module fully specify what the next action should be, given the current state. Thus the programmer may issue a wild-card receive, but must then continue execution based on the message received. More typically, modules in such languages block after issuing a "receive" for specific messages (identified by tags and source processors, for example). During this "blocking" the semantics requires that no other actions should take place within the same process (i.e. there should be no side effects, when the receive returns, beyond the expected side effect of returning the message). Thus such languages do not require scheduling.

- Concurrent objects: Concurrent object-oriented languages such as Charm allow concurrency within a process. Such languages permit asynchronous method invocations — the caller is not made to wait for the invocation to complete. There may be many objects active on a processor, any of which can be scheduled depending on the arrival of a message corresponding to a method invocation. Moreover, an object can handle any message that arrives; it is not necessarily blocked waiting for a particular message. Such objects are called message-driven objects.
- Multithreading: Another set of languages allow concurrency by threads— they permit multiple threads of control to be active simultaneously, each with its own stack and program counter. The threads execute concurrently under the control of a thread scheduler.

Most languages can be seen to fall within one of these three categories, as far as internal concurrency is concerned.

## 2.2 Control Regime

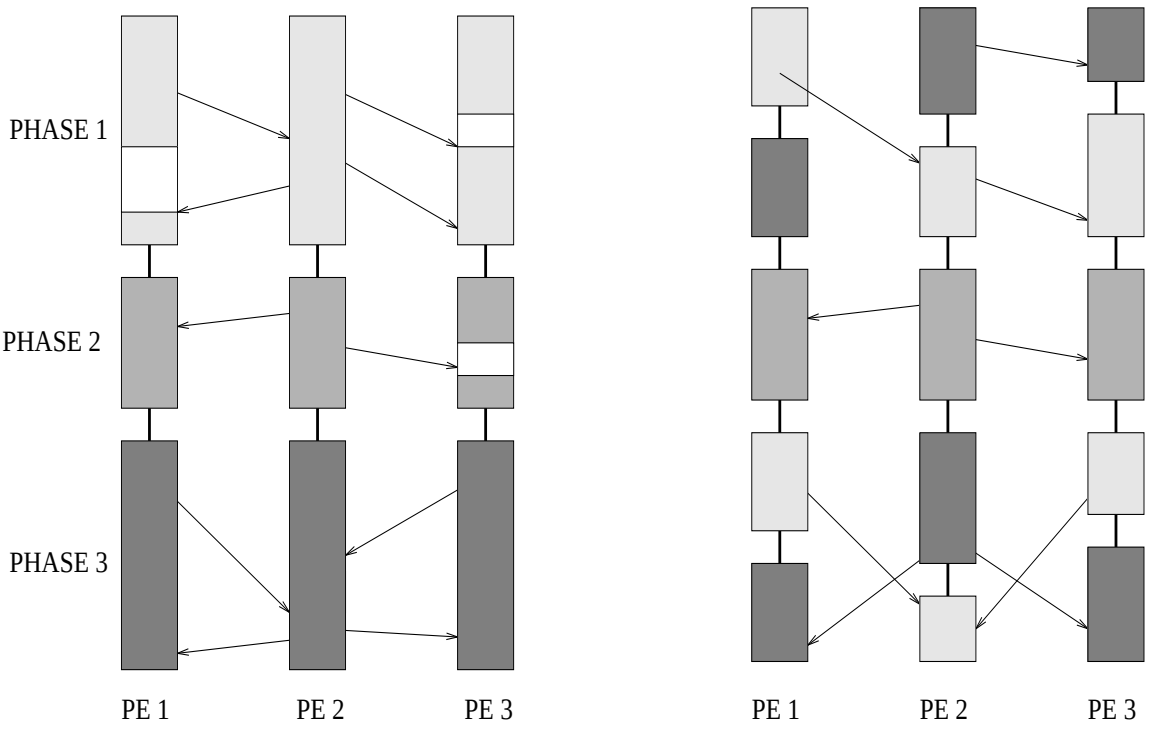
Another related aspect is the *control regime* for a language, which specifies how and when control transfers from one program component to another within a single process. Modules interact via *explicit* and *implicit* control regimes.

The *explicit control regime* consists of non-overlapping modules of a parallel program from different languages, as described in Figure 1(a). The transfer of control from module to module is explicitly coded in the application program in the form of function calls. Thus all processors execute modules from different languages in a deterministic, loosely synchronous manner. This explicit interoperability is sufficient for many scientific applications which can be programmed in a loosely synchronous manner, enabling different non-overlapping phases of a program to be coded in different languages. It is suitable for languages of the first type which have no concurrency within a process.

The *implicit control regime* (Figure 1(b)) is motivated by a need to reuse parallel software components in an overlapped manner, so that entities in different modules can be simultaneously active. The transfer of control from module to module is implicit; rather than being decided by the application program, it is decided dynamically by a scheduling policy in the run-time system. This model allows an adaptive sequence of execution of application code with a view to providing maximal overlap of modules for reducing idle time. Thus, when a thread in one module blocks, code from another module can be executed during that otherwise idle time. Implicit interoperability is suitable for concurrent languages with concurrent objects or threads within a process.

## 2.3 Prioritization in Implicit Control Regimes

In an implicit control regime, the runtime system must make scheduling decisions. When an entity— an object or a thread— relinquishes control, the system must choose between the multiple possible modules or computational actions that it can continue. The possible actions are represented by ready threads or messages for local objects. Which one of these possible actions should it pursue next? In some applications, this decision may be unimportant, while in others it




(a) Explicit Interoperability (b) Implicit Interoperability

Figure 1: Control Regimes

may impact performance in a more significant way, so it is necessary to associate priorities with possible actions. Such applications include: discrete event simulation (especially with the optimistic concurrency control protocols where time must be used as a priority); branch-and-bound problems, where the lower-bound of a node must be used as a priority to get good speedups [23]; state space search problems, where bit-vector priorities are needed to ensure consistent and monotonic speedups [22]; and numerical computations with critical paths where priorities can be used to speed up the critical path [11]. Such prioritization mechanisms can be provided only by allowing the application to select the type of queueing strategy it wants to use.

The interoperability framework, therefore, must be able to support priorities and prioritized queuing for languages and computations that require them, while not penalizing performance for those that do not.

To summarize, then, the computational model allows three kinds of entities: Single-process modules (SPMs), message driven objects, and threads. Modules in languages with implicit control regimes interleave their execution with each other under the control of a scheduler. Those with explicit control regimes (SPMs) transfer control across modules via function calls only. The SPM modules are allowed to invoke computations in concurrent regimes by explicitly transfering control to them. In the next section we describe the run time framework that follows from this computational model.

# 3 Design and Architecture of Converse

The design of the Converse framework is based on the following fundamental guidelines:

1. *Completeness of coverage*: the framework should be able to efficiently support most, if not all, approaches, languages and libraries for parallel programming. More concretely, any particular language or library that can be portably implemented on MIMD computers should be able to run on top of Converse, using its facilities and interoperating with other languages. This should be possible without undue overhead for (a) any remote operations such as messages, and (b) any local scheduling it requires, such as the scheduling of ready threads. An acceptable overhead in this context is a few tens of instructions over and above the cost of such operations in a native implementation— i.e., a direct implementation of a particular language on a particular machine.

2. *Need based cost*: The Converse framework, being general, must support a variety of features. However, each language or paradigm should incur only the cost for the features it uses.

To satisfy these requirements, the architecture of Converse is component-based, rather than monolithic. The system consists of multiple components, each of which is fully specified via a detailed interface specification. For each component, multiple alternative implementations may exist. Thus, an application that requires sophisticated dynamic load balancing might link in a more complex load balancing strategy with its concomitant overhead, while another application may link in a very simple and efficient load balancing strategy.

An important observation that influenced this design is the fact that threads and message-driven objects (i.e. modules using implicit control strategies) need a scheduler, and a single unified scheduler can be used to serve the needs of both. The other components of Converse are a machine interface, message managers, thread objects, individual language runtimes, and a few other support modules, as shown in Figure 2.

## 3.1 The core components

The unified scheduler, an assortment of queuing strategies and a simple but flexible machine interface that is implemented differently on different machines constitute the core components of Converse. These modules are based on a generalized notion of messages, which is described next.

### 3.1.1 Generalized Messages

In order to unify the scheduling of all concurrent entities, including message-driven objects and threads, we generalize the notion of a message. A generalized message is an arbitrary block of memory, with the first word specifying a function that will handle the message. The function may be specified by a direct pointer or by an index into a table of functions. The latter method has the advantage of working even on heterogeneous machines, and requires less space than a pointer, and is therefore used in most of our implementations. Any function that is used for handling messages must first be registered with the scheduler. A generalized message can represent any one of the following:

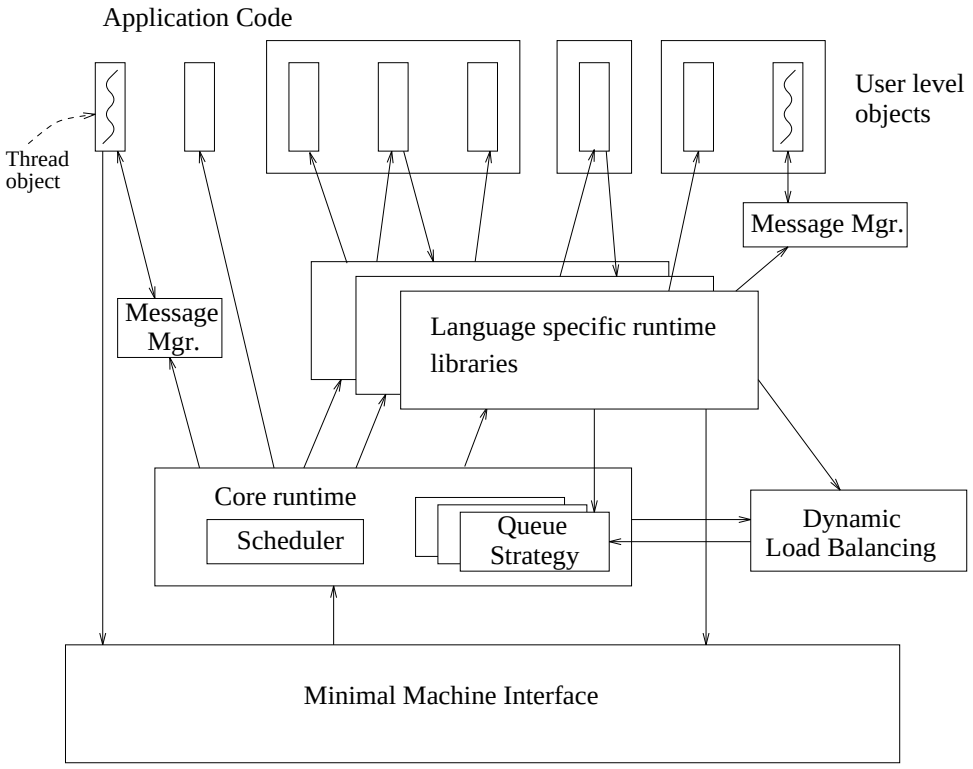



Figure 2: Software Architecture for Interoperability

1. a message sent from a remote processor
2. a scheduler entry for a ready thread
3. a delayed function with its argument

### 3.1.2 The Scheduler

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

Converse supplies two additional variants of the scheduler for flexibility. `ScheduleFor(n)` runs the scheduler loop for $n$ iterations. E.g. This call is useful for SPM modules to allow a

```
void Scheduler()
{
     while ( not done ) {
        DeliverMsgs();
        message = Dequeue(SchedulerQueue);
        (HandlerOf(message))(message);  // Deliver it to language-specific handler
     }
}

void DeliverMsgs()
{
     while ( there is a message available from the machine layer ) {
        message = CmiGetMsg();          // Get message from the machine layer
        (HandlerOf(message))(message);  // Deliver it to language-specific handler
     }
}
```

Figure 3: Pseudo code for scheduler loop in core runtime system

certain amount of concurrent execution while they wait for data. `ScheduleUntilIdle()` runs the scheduler loop until there are no messages left in either the network's queue or the scheduler's queue.

For modules written in the explicit control regime, control stays within the user code all the time. However, for modules in the implicit control regime, control must shift back and forth between a system scheduler and user code. For these apparently incompatible regimes to coexist, it is necessary to expose the scheduler to the user program, rather than keeping it buried inside the run-time system. An SPM module can explicitly relinquish control to the scheduler to allow execution of multi-threaded and message-driven components[1].

### 3.1.3 Machine Interface

The machine interface is divided into two parts: the MMI (Minimal Machine Interface) and the EMI (Extended Machine Interface). The MMI contains calls which *must* be implemented to port Converse to a particular machine. The EMI calls can be implemented using the MMI calls, although it is advisable to implement them at a lower level for particular machines, for the sake of efficiency.

#### MMI: Minimal Machine Interface

The MMI layer defines a minimal interface between the machine independent part of the

[1] A typical interaction between the two control regimes may proceed as follows. The SPM module may carry out a possibly parallel computation with sends and receives, and then invoke a function $f$ in a concurrent module (such as one written in Charm). This module may change its state and deposit some messages for other entities. When this function $f$ returns, the SPM module explicitly invokes the scheduler, which executes the concurrent computations triggered by the previously deposited messages. The result of the concurrent computation is passed by function calls to the SPM module before the scheduler returns.

runtime such as the scheduler and the machine dependent part which is different for different parallel computers. Portability layers such as PVM and MPI also provide such a portable interface. However, they represent an overkill for our requirements. For example, MPI provides a "receive" call based on context, tag and source processor. It also guarantees that messages are delivered in the sequence in which they are sent between a pair of processors. The overhead of maintaining messages indexed for such retrieval or for maintaining delivery sequence is unnecessary for many applications. The interface we propose to develop is minimal, yet it is possible to provide an efficient MPI-style retrieval on top of this interface.

The MMI module is responsible for process creation, process coordination at the initiation and termination points, communication and other machine-specific utilities. The CmiInit() call must precede any other calls to the machine interface. The CmiExit() call must not be succeeded by any other call to the MMI.

The communication primitives include those for sending, broadcasting, and picking messages. The MMI supports both synchronous and asynchronous send calls.
`CmiSyncSend(destproc,buf,len)` takes a generalized message stored in `buf` of length `len`, and sends it to the processor `destproc`. When the call returns, the caller may use and change data in `buf`. The `CmiAsyncSend` call is provided so that the application program may continue to work while the machine is trying to send data from `buf`. The `CmiAsyncSend` call returns a handle that the user can check with a `CmiSendDone` call which returns the status of the send.

The MMI provides many variants of broadcast calls. Note that the broadcast is called only by the processor sending the message. Thus, our broadcast is *not* a barrier.

For retrieving messages that have arrived from other processors, the MMI provides the `CmiGetMsg` call, which returns a pointer to a recently received message. After retrieving a message with this call, the scheduler simply invokes the handler indicated by the message. To optimize this further, when desired, the MMI provides a `CmiDeliverMsgs` call, which invokes the handler for all the messages that have been received from the network by the MMI layer.

For supporting no concurrency languages (SPM), which may require that no other activity takes place in user space while the program is blocked waiting for a specific message, the MMI provides a `CmiGetSpecificMsg` call, which waits for a message for a particular handler buffering any messages meant for other handlers.

Efficient, flexible buffer management for the received messages is an important issue. The complexity here arises due to variations in different machine and application contexts. On some machines, it may not be possible to give the user code control of the system buffer in which the message was received for an indefinite time period. Also, some application programs may be able to consume data in messages as they arrive from the network, while others may require that the data be queued before it is processed. To avoid buffer copying to the greatest extent possible, while still keeping the design portable, we provide the following buffer management protocol: By default, MMI owns the message buffer. If a handler needs to keep the buffer (e.g. for queuing the message), it should explicitly call `CmiGrabBuffer(&buf_ptr)`, which transfers the ownership of the buffer to the handler. On machines where message buffers reside in system space, MMI will transfer a copy of the buffer.

The `CmiPrintf` and `CmiScanf` calls provide atomic writes and reads to standard output and input, respectively. `CmiPrintf`s from different processors are sent to the user's terminal, and

the MMI guarantees that data from two separate `printfs` is not interleaved. Similarly, the `scanf` calls from different sources are effectively serialized. The MMI provides blocking and non-blocking versions of `scanf`. The non-blocking version specifies a handler to receive the result of a `scanf`, which is sent in the form of a formatted string, which the recipient can re-scan using `sscanf`, for example.

The MMI provides a number of utility calls including timers with different resolutions, and calls to determine the logical processor number and the total number of processors.

### EMI: Extended Machine Interface

The calls in the EMI are concerned with scatter and gather style communications, processor groups, and global memory operations. The `CmiGatherSend` call gathers data from multiple addresses of given sizes, packs them into a single message and sends it to the given destination. It is not necessary that a message sent via a *gather* is received via a *scatter* call, or vice-versa. The scattering related calls are more complex because they must also specify how to identify a message for which scattering needs to be done in a particular manner. The scatter-related calls are "advance receive" calls, in that it is expected (although not required) that these calls are made before the actual message arrives. The calls specify how to identify their target with offsets and values. They also specify which parts of matching messages must be copied to which of the user data areas. Two variants of this call are provided, one of which simply scatters the data on receipt of the message, while the other enqueues a short empty message in addition. The latter is sometimes necessary to notify the recipient that the data has arrived.

Often entities in a subgroup of processors need to engage in group communication. The machine layer, which is knowledgeable about topology and other communication aspects, is best able to optimize such group operations. For this reason, the EMI provides calls for establishing process groups, broadcasting to an established process group, and carrying out reductions and other global operations, as well as spanning-tree based operations within a processor group.

For transferring data between local and remote processors transparently, Converse provides asynchronous *get* and *put* calls, and global pointers. A global pointer is an opaque handler, which specifies a particular address on a particular processor. The EMI allows one to convert a local address into a local pointer and pass it around. The synchronous calls wait until the specified *get* or *put* operations complete, while the asynchronous calls return immediately, and allow the operation to complete at a later time.

## 3.2 Supporting Threads

The components that are useful for supporting multithreading are described below.

### 3.2.1 Message Managers

A message manager is simply a container for storing messages. It stores a subset of messages that are yet to be processed, serving as an indexed mailbox. A message manager basically provides calls to insert and retrieve messages. Messages may be retrieved based on one or more "identification marks" on the message. A tag and a source processor number are examples of such identification marks in PVM. Instances of message managers provided in Converse can be customized to either

one or two tags and placed at arbitrary positions within the messages. Another call allows one to probe for the existence of a particular message specified by its tags. Retrieval or probes are allowed to "wildcard" the tag field. The message managers can be used by threaded languages as well as SPMs.

### 3.2.2 The thread object

Thread-based programs can be thought of as a collection of multiple objects, each one with its thread of control and making progress independent of other objects. Of course if there is only one processor, control needs to switch back and forth among these objects, under the control of some scheduler, and concurrency control mechanisms such as locks must be provided to allow threads to share data in a safe manner in spite of the interleaving of control among them.

A threads package typically consists of these components:

- an ability to freeze or suspend the execution or running of a thread and to resume the execution of a previously suspended thread
- a scheduler that manages the transfer of control among the objects
- a concurrency control mechanism such as locks

Many thread packages have been developed in the past few years [20, 8]. The Posix standard for threads has also emerged. However the *gluing together* of scheduling, concurrency control and other features with the ability to suspend and resume threads is problematic from the point of view of interoperability. E.g. the particular scheduling strategy provided by the threads package may not be appropriate for the problem at hand. Converse separates the capabilities of thread packages modularly. In particular, it provides a *thread object* [1] that encapsulates the essential capability of a thread— the ability to suspend and resume a thread of control— by encapsulating the stack and the program counter.

The basic primitives provided are the following:

(a) **Create** a thread: create the thread object, set its initial data and return a pointer to this object.

(b) **Resume** a thread: continue or start the execution of a particular thread.

(c) **Suspend** the current thread: stop the execution of the current thread and transfers control to another thread.

(d) **Awaken** a thread: add the thread to the ready list of its scheduler.

(e) **Yield** control to scheduler: suspend a thread and immediately awaken it.

(f) **Exit** the current thread.

The thread object is not meant to be used by the end user directly (although it can be so used). Rather, runtime systems of individual languages or packages may use the thread object to

implement their thread functionalities easily. For example, *tSM*, the threaded simple-messaging package, provides to its users the following calls that make use of the thread object internally:

`tSMCreate()`: Create a new thread, and schedule it for execution via the converse scheduler.

`tSMReceive()`: block the thread waiting for a particular (tagged) message.

The low level calls to thread object are not exposed to the users of *tSM*.

The thread object is primarily implemented through the C language calls to `setjmp` and `longjmp` which allow state information (program counter, stack pointer and registers) to be *saved* and later *jumped* to. Our current implementation is based in part on the Cthreads thread package. Each thread object holds fields such as a pointer to the stack, jump buffers that hold the state of a thread and its resumer and a pointer to a scheduler when applicable.

### 3.2.3 Synchronization mechanisms

Two types of synchronization mechanisms are supported— *locks* and *conditional variables*.

**Locks** are implemented by having queues attached to each lock. The thread trying to obtain a lock continues (after setting the lock to its locked state) if the lock can be obtained. If not, the thread is placed in a queue for the lock, and the thread is suspended. A thread which releases the lock causes the shifting of ownership of the lock to the first thread in this queue and **awakens** this thread so that it can continue executing when it is scheduled.

**Condition variables** allow several threads to block on a single condition. Calls are provided for threads to **wait** on a condition variable, and for threads to either **signal** a condition variable, causing the unblocking of one of the threads, or to **broadcast** a condition variable, which causes the unblocking (i.e. awakening) of all the threads that are waiting on the condition variable.

The functionality outlined above is an extension of the Posix threads standard. The only notable difference is that the scheduler is separated out and some thread management calls are exposed to the user. This allows one to have different implementations of thread schedulers, and further allows a certain amount of freedom in the manipulation of threads.

## 3.3 Language and paradigm specific runtime modules

Most parallel languages implemented using Converse would require at least a small runtime library of their own. Strictly speaking, such libraries are not part of the Converse framework. However, their interfaces are specified in part by Converse.

Each language runtime can be part of an object by itself, with encapsulated data of its own. When created, a language runtime registers one or more handlers with Converse. These language-specific handlers then implement the specific actions they must take on receipt of messages from remote or local entities. The language handlers may process such messages immediately, or enqueue them in the scheduler's queue, to be picked up in accordance with their priority, for example. In the latter case, to avoid infinite regress, the handler stored in the message may be changed to point to a second handler defined by the language runtime. This handler knows that

any messages given to it have come from the queue, and therefore does not attempt to re-enqueue them. The language handlers also implement other language-specific functions that are called by entities in their language. These functions may end up sending messages using the MMI or invoking other components of the Converse framework.

### 3.3.1 Dynamic load balancing

The need for load balancing arises in parallel programs in many contexts. A particular situation of interest is when the program creates a piece of work or a task that can be executed on any processor[2]. An example of this occurs when a programmer specifies creation of a parallel object in a language such as Charm. Such objects can be located on any processor under the system's control. It is not necessary for the system to immediately allocate them to a particular processor. The *seeds* for such objects can float around the system until they "take root" on a particular processor. However, once they are anchored to a processor it is harder to move them to other processors because other objects must be informed of their new locations. To deal with such "agenda items" or seeds, Converse provides a dynamic load balancing module.

A language runtime may hand over a seed, in the form of a generalized message, on any processor. Monitoring the load on processors, the load balancing module moves such seeds from processor to processor until it eventually hands over the seed to its handler on some destination processor. This module may interact with a local scheduler and may send messages to its counterparts on remote processors for exchanging load status information. It can also make calls to other entities for ascertaining the load on the local processors. Although the interface to the load balancing strategy is fully defined, there are a large number of load balancing modules supported in Converse. Each one is often useful in a different situation. Depending on the application, the user is able to link in a different load balancing strategy.

### 3.3.2 Support for Tools

In order to use various performance feedback, simulation and debugging tools on programs developed in the above framework, Converse supports a standard for an event trace format. This consists of two parts: a standard format which must be adhered to by all language implementors, and an extensible self-describing format which may be language-specific. In addition to recording message send, receive and processing events, object or thread creation must also be recorded. Converse provides a module to record these traces. Again, many variants of this module are provided, depending on the sophistication of the tracing desired.

[2]Other kinds of load balancing situations include dynamic object migration and quasi-dynamic load balancing. In object migration, entities such as message-driven objects or individual threads are moved from one processor to another while the computation is in progress. Supporting this involves queues for forwarding messages to migrated objects [7]. In quasi-dynamic load balancing, after a phase or period of computation has completed, the load and communication patterns in that phase are analyzed, and a new global distribution of entities to processors is derived. After moving the entities to their new destinations and updating their addresses with all acquaintances, the computation proceeds to the next stage. Both migration and quasi-dynamic load balancing can be implemented on top of Converse as Converse libraries. These, however, are beyond the scope of this paper.

# 4 Utility of Converse

The utility of Converse is manifested in the following ways:

1. The programmer does not have to convert an entire application to a particular language. For each part of the overall application, the most suitable language or paradigm can be used.
2. Pre-existing libraries written in different languages can be reused in a single application.
3. Development of parallel languages, coordination languages, or libraries, based on new paradigms is simplified.

These benefits are illustrated with the examples in this section.

Consider the Fast Multipole Algorithm used for computing electrostatic or gravitational interactions among a large number of particles [18]. When the algorithm begins execution, it is given a set of particles on each processor. Its first task is to form a tree by recursively dividing the space based on the number of particles in each partition. This subdivision, in its simple formulation, can be implemented in a traditional single-process module. (Alternatively, a more sophisticated variant for this phase can be implemented using message-driven objects that overlap communication and computation.) Next, an all-to-all communication phase is required to transfer particles to their destination cells. We would like to continue execution of each cell as soon as all of its particles have arrived, this phase can be better implemented using message-driven objects such as in Charm++. The logic of individual cells can be naturally expressed as threads which would communicate along the edges of the tree formed using any other traditional message passing primitives, such as PVM or NXLib.

Large applications are often written through a collaborative team effort. For example, we are involved in a Grand Challenge application project developing a molecular dynamics application. The group involves subgroups at different sites, and each group often has a favorite language in which they prefer to develop their portions of the project. The core molecular dynamics program, *NAMD* [3], carries out basic biophysics calculations including short-range electrostatic forces, and depends on the Fast Multipole Algorithm (FMA) to compute long-range electrostatic forces. There are two implementations of FMA, one in PVM and the other in Charm++. As a result, both a Charm++ and PVM version of NAMD are being maintained. With Converse it will be possible to use the Charm++ version of NAMD with the PVM-based FMA module.

The third benefit of Converse has to do with the ability to put together a new language quickly and efficiently. As an example, consider a small "coordination language" that supports simple message-driven threads. Threads can be dynamically created and can send messages with a single tag to other threads. Individual threads can block for a specific message (with a particular tag) and must be continued when the message is received. By using the facilities by the message manager and thread object, as well as the Converse scheduler, one of us was able to implement this language in about a day's time. The entire runtime for this language consists of about 100 lines of C code. For more complex languages, the effort required is dominated by the compilation and optimization aspects, as it should be. The back-end and the runtime library become relatively simple by using Converse.

# 5 Implementation and Performance

The basic Converse framework has been implemented on networks of Sun workstations, clusters of HP workstations connected by an ATM switch, Cray T3D using the FM package [17], networks of Suns using the Myrinet switch with the FM package, IBM SP-2, and Intel Paragon running SUNMOS. It will be ported to all the machines that Charm currently runs on in the near future. (These include the CM-5, Convex Exemplar, nCUBE/2, network of RS/6000s, and some shared memory machines.) The Charm runtime system itself has been retargeted for Converse. Prototype implementations of PVM, NXLib, and SM (a simple messaging layer) are complete.

The machine interface of Converse is meant to be implemented at the lowest level on individual machines. On some machines, such as Cray T3D, the lowest and most efficient layers of the system were available to us. On other machines, it is necessary to secure the vendor's cooperation to implement the machine interface most efficiently. For example, we were able to obtain efficient implementations of Converse on a network of Suns connected by the Myrinet network, through the cooperation of the Concurrent Systems Architecture group at the University of Illinois, which is developing the low-level interface to the switch.

In this section, we will demonstrate some simple performance measurements carried out in our current implementation.

## 5.1 Message Passing Performance

The first set of experiments (Figures 4, 5 6, 7, 8) involves simple message passing performance. This was measured using a round trip program that sends a large number of messages back and forth between two processors. Using this, the average time for one individual message send, transmission, receipt and handling was computed for various machines. On the receiving processor, for every message, the message was delivered to a handler which responded by sending a return message.

The performance data for various machines is shown in figures 4 through 8. Overall, the performance is almost as good as that of the lowest level communication layer available to us on these machines. For example, the FM library using Myrinet switches delivers messages up to 128 bytes in 25 $\mu$seconds, whereas Converse messages need about 31 $\mu$seconds. On the T3D, the performance is very close to the best possible on the Cray hardware for short messages. The jump at 16K bytes (Figure 5) is due to copying during packetization, which we believe can be eliminated.

Note that there was no queueing overhead in the first set of experiments. In the second experiment, we incorporated the queueing overhead. Each handler upon receiving a message enqueues it in the scheduler's queue. The scheduler then picks a message from its queue and schedules it for execution. This cost of scheduling is paid only by languages such as Charm which use the queue for scheduling objects. This experiment was done only on one machine (Sun workstations connected by Myrinet switches — Figure 6) to illustrate the magnitude of scheduling overhead. The scheduling is seen to add about 9 to 15 $\mu$seconds for short messages. For large messages, the relative difference becomes negligible.

Thus, although Converse provides a broad functionality, it achieves its objective of ensuring

that languages and applications pay the overhead only for features that they use.

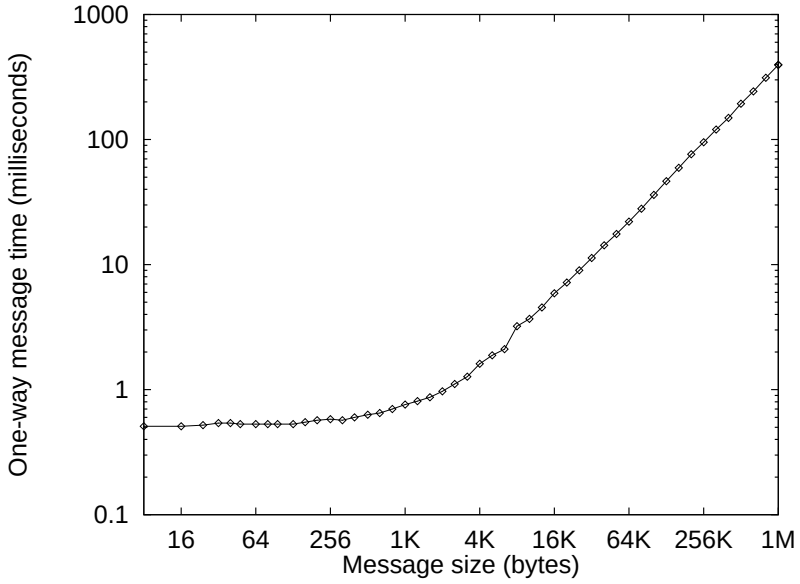



Figure 4: Message Passing Performance on ATM-connected HPs

# 6 Summary and Future Work

We presented the design and rationale for a comprehensive framework for supporting interoperability among a wide range of parallel languages and paradigms. The design is based on the fact that entities in different parallel languages can be classified into three basic categories from the point of view of the scheduling of the processor: (1) single-process modules which permit no concurrency and require programmers to transfer control among modules explicitly; (2) message-driven objects, and (3) threads, which both allow for concurrency and transfer control among their modules implicitly under the control of a scheduler. A unified scheduler and a generalized notion of messages allowed these three basic paradigms to coexist. The thread object (which supports the thread abstraction without intertwining scheduling functionality) and the generic message manager (which can be used to store and retrieve messages) further facilitate the design and implementation of individual language runtimes.

Although we are convinced of the breadth and flexibility of the Converse design, it is clear that additional research and implementation effort is needed for Converse to fulfill its promise — that of supporting interoperability between a wide variety of languages without loss of efficiency.

Our initial studies have indicated that further performance improvements are possible with low level optimizations in Converse's message passing implementation. We plan to streamline the performance of Converse and continue porting it to other machines. Preemptive messages (interrupt messages) will be investigated in the future. Design of appropriate primitives for parallel file I/O and their implementations on different machines will also be the subject of future research. This important area is complicated because of the lack of consensus on a common I/O architecture. Many of the languages in which we are interested are object based. We plan to work toward a standardized representation of objects across languages which will permit communication

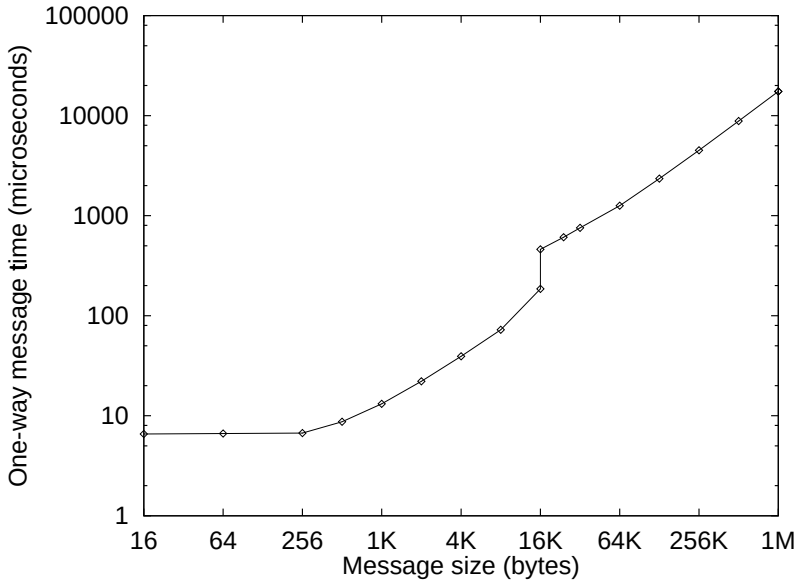



Figure 5: Message Passing Performance on Cray T3D

of first class objects and methods. Finally, we will use the feedback from implementing multiple languages and multi-lingual applications to refine the design and implementation of Converse.

**Acknowledgements:** The authors would like to thank Terry Allen for his help in writing this paper, Robert Brunner for conducting many of the performance experiments, and these as well as other members of the Parallel Programming Laboratory including Attila Gursoy and Joshua Yelon for their input in the design of Converse. We would also like to thank Prof. Andrew Chien and his research group at the University of Illinois for making the Myrinet and their FM implementation available to us. We are grateful to Prof. Klaus Schulten and the Theoretical Biophysics group at Beckman Institute at the University of Illinois, the Pittsburgh Supercomputing Center, and Argonne National Laboratory for the use of their computing facilities.

## References

[1] Thread objects. Technical report, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1995.

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

[3] J. Board, L. V. Kale, K. Schulten, R. Skeel, and T. Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

[4] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language, 1992.

[5] K. Mani Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. Technical Report Caltech-CS-TR-92-13, Department of Computer Science, California Institute of Technology, 1992.

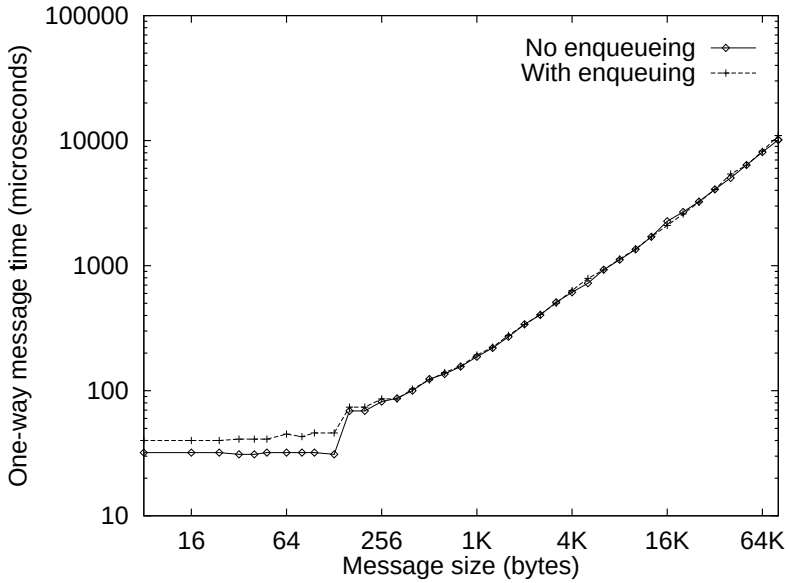



Figure 6: FM Message Passing Performance

[6] A. Chien. *Concurrent Aggregates*. MIT Press, 1993.

[7] N. Doulas and B. Ramkumar. Task migration in message driven systems. In *First Annual Workshop on Message Driven Execution and Charm*, Urbana, Illinois, Oct 1994.

[8] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1994.

[9] 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.

[10] A. Grimshaw. Easy to use object oriented parallel processing with Mentat. *IEEE Computer*, May 1993.

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

[12] M. Hainer, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing '94*, November 1994.

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

[14] S. Hiranandani, K. Kennedy, and C. Tseng. *Compiler support for machine independent parallel programming in Fortran-D*. Elsevier Science Publishers B.V., 1992.

[15] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, August 1990.

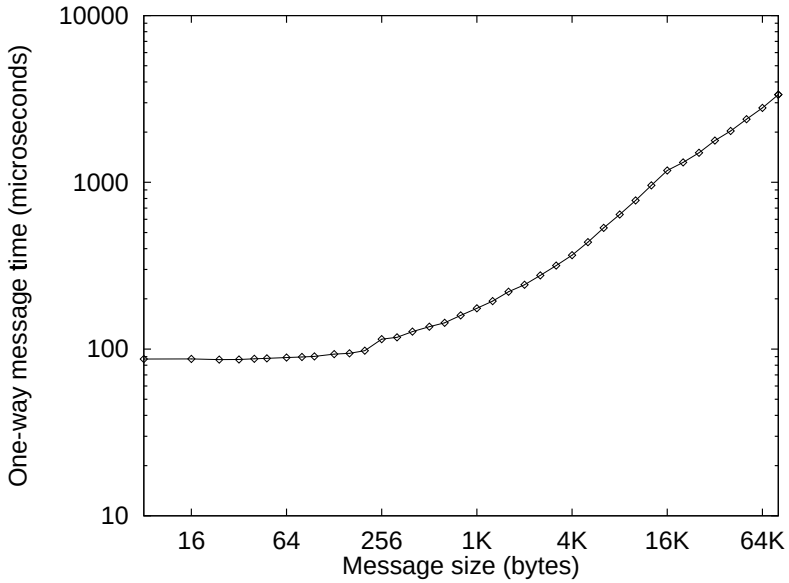



Figure 7: SP1 Message Passing Performance

[16] L.V. Kale and S. Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, September 1993.

[17] V. Karamcheti and A. Chien. A comparison of architectural support for messaging in the tmc cm-5 and the cray t3d. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita, Italy, June 1995. To appear.

[18] S. Krishnan and L. V. Kale. A parallel adaptive fast multipole algorithm for n-body problems. In *Proceedings of the International Conference on Parallel Processing*, August 1995. To Appear.

[19] Message Passing Interface Forum. *Document for a Standard Message-Passing Interface*, November 1993.

[20] B. Mukherjee, G. Eisenhauer, and K. Ghosh. A machine independent interface for lightweight threads. *Operating Systems Review of the ACM SIG in Operating Systems*, pages 33–47, January 1994.

[21] R. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, aug 1993.

[22] V. Saletore and L.V. Kale. Consistent linear speedups for a first solution in parallel state-space search. In *Proceedings of the AAAI*, August 1990.

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

[24] A. Yonezawa. *ABCL: An Object Oriented Concurrent System.* MIT Press, 1990.

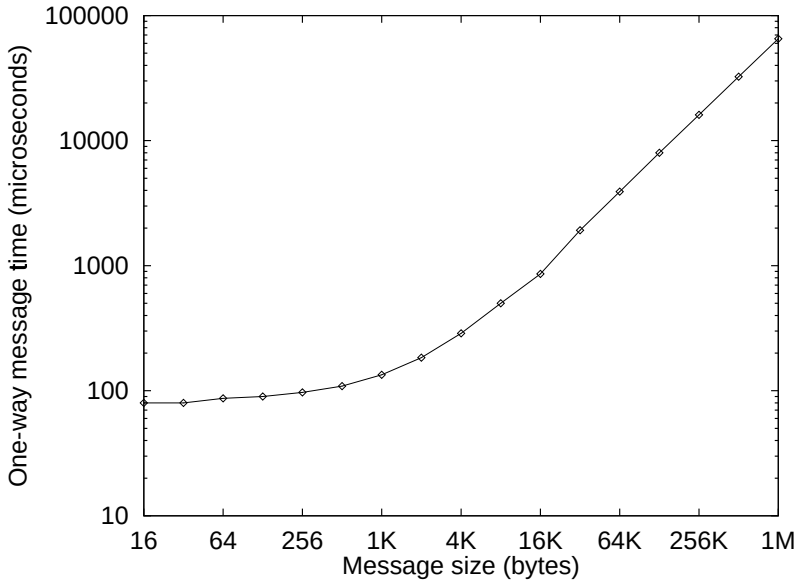



Figure 8: Paragon Message Passing Performance

# Appendix : Converse API Reference

## 1 Initialization and Completion

void ConverseInit(void)
This call initializes all Converse components, such as the scheduler, machine interface, and other libraries. This must be the first Converse call in the entire program.

void ConverseExit(void)
This call wraps up all Converse components. No Converse call may be made after this call.

## 2 Scheduler Calls

void CsdScheduler(int NumberOfMessages)
This call invokes the Converse scheduler. The `NumberOfMessages` parameter specifies how many messages should be processed (i.e. delivered to their handlers). If set to -1, the scheduler continues processing messages until CsdExitScheduler() is called from a message handler.

void CsdExitScheduler(void)
This call causes the scheduler to stop processing messages when control has returned back to it. The scheduler then returns to its calling routine.

void CsdEnqueue(void *Message)
This call enqueues a message in the scheduler's queue, to be processed in accordance with the queueing strategy. This call is usually made from a message handler when the message is not to

be processed immediately, but may be processed later (e.g. depending on the message's priority). Also, it is used to enqueue local ready entities, such as threads.

# 3 Converse Machine Interface

## 3.1 Message Handler Calls

int CmiMsgHeaderSizeBytes(void)
This call returns the size of the message header in bytes.

void CmiSetHandler(int *MessageBuffer, int HandlerId)
This call sets the handler field of a message to `HandlerId`.

HANDLER CmiGetHandlerFunction(int *MessageBuffer)
This call returns the handler function pointer for a message. This usually involves looking up a table using the handler-id stored in the message header at the sending processor. HANDLER is defined as `typedef void (*HANDLER)(void *)` .

int CmiRegisterHandler(HANDLER HandlerFunction)
This call registers a message handler with the CMI and returns a handler index which can be subsequently used to specify the handler for a message.

## 3.2 Timer Calls

double CmiTimer(void)
Returns current value of the timer in seconds. This is typically the time spent since the `CmiInit()` call. The precision of this timer is the best available on the particular machine, and usually has at least microsecond accuracy.

## 3.3 Point-To-Point Communication

void *CmiGetSpecificMsg(int HandlerId)
This call waits until a message for the specified handler is available, and returns a pointer to the message buffer. Ownership of the message buffer is maintained with the CMI (e.g. another call to `CmiGetMsg()` or `CmiSpecificMsg()` can overwrite the contents of this buffer.). The message handler should explicitly call `CmiGrabBuffer()` to acquire ownership of the message buffer.

CommHandle CmiAsyncSend(unsigned int destPE, unsigned int size, void *msg)
Initiates an asynchronous send of `msg` of length `size` bytes to processor `destPE` and returns a communication handle which could be used to enquire the status of this communication. Message buffer for `mesg` should not be reused or freed until communication is complete.

void CmiSyncSend(unsigned int destPE, unsigned int size, void *msg)
Sends `msg` of size `size` bytes to processor `destPE`. Message buffer for `msg` could be reused after the call returns.

int CmiAsyncMsgSent(CommHandle handle)
Returns the status of asynchronous send specified by communication handle `handle`.

void CmiReleaseCommHandle(CommHandle handle)
Releases the communication handle `handle` and associated resources. It does not free the message buffer. `handle` could be reused by CMI for another communication after this call succeeds.

CommHandle CmiVectorSend(int destPE, int HandlerId, int len, int sizes[], void *DataArray[])
Initiates an asynchronous send of data to processor `destPE`. The data consists of `len` pieces residing in different areas of memory, which are logically concatenated. The `DataArray` array contains pointers to the pieces; the size of `DataArray[i]` is taken from `sizes[i]`. `HandlerId` is inserted at the appropriate place in the combined message and the corresponding handler will be invoked on the receiving side. This function returns a communication handle which could be used to enquire about the status of communication using `CmiAsyncMsgSent()`. Individual pieces of data as well as the arrays `sizes` and `DataArray` should not be overwritten or freed before the communication is complete.

void CmiGrabBuffer(void **pbuf)
Transfers the ownership of the buffer pointed to by `*pbuf` to the calling procedure. If `*pbuf` points to a system buffer, CMI copies the buffer contents to newly allocated user space and updates `*pbuf` to point to the new buffer.

## 3.4 Global Pointer

int CmiGptrCreate(GlobalPtr *gptr, void *lptr, unsigned int size)
This function creates a global pointer by initializing contents of `*gptr` to point to memory on the local processor pointed to by `lptr` of `size` bytes. `*gptr` could then be sent to other processors, and could be used by `CmiGet()` and `CmiPut()` to read and write this memory by remote processors. This functions returns a positive integer on success.

void *CmiGptrDref(GlobalPtr *gptr)
This function returns the address of local memory associated with global pointer `gptr`.

int CmiSyncGet(GlobalPtr *gptr, void *lptr, unsigned int size)
Copies `size` bytes from memory pointed to by global pointer `gptr` to local memory pointed to by `lptr`. This is a synchronous operation and the calling processor blocks until the data is transferred to local memory. This function returns a positive integer on success.

CommHandle CmiGet(GlobalPtr *gptr, void *lptr, unsigned int size)
Initiates copying of `size` bytes from memory pointed to by global pointer `gptr` to local memory pointed to by `lptr`. This function returns a communication handle which could be used to enquire about the status of this operation.

CommHandle CmiPut(GlobalPtr *gptr, void *lptr, unsigned int size)
Initiates copying of `size` bytes from a processor's local memory pointed to by `lptr` to the memory pointed to by global pointer `gptr`. This function returns a communication handle which could be used to enquire about the status of this operation.

## 3.5 Group Communication

void CmiSyncBroadcast(unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all processors excluding the processor on which the caller resides.

void CmiSyncBroadcastAllAndFree(unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all processors including the processor on which the caller resides. This function frees the message buffer for `msg` before returning, so `msg` must point to a dynamically allocated buffer.

void CmiSyncBroadcastAll(unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all processors including the processor on which the caller resides. This function does not free the message buffer for `msg`.

CommHandle CmiAsyncBroadcast(unsigned int size, void *msg)
Initiates asynchronous broadcast of message `msg` of length `size` bytes to all processors excluding the processor on which the caller resides. It returns a communication handle which could be used to check the status of this send using `CmiAsyncMsgSent()`. `msg` should not be overwritten or freed before the communication is complete.

CommHandle CmiAsyncBroadcastAll(unsigned int size, void *msg)
Initiates asynchronous broadcast of message `msg` of length `size` bytes to all processors including the processor on which the caller resides. It returns a communication handle which could be used to check the status of this send using `CmiAsyncMsgSent()`. `msg` should not be overwritten or freed before the communication is complete.

## 3.6 Processor Ids

int CmiNumPe(void)
Returns total number of processors in the machine on which the parallel program is being run.

int CmiMyPe(void)
Returns the logical processor identifier of processor on which the caller resides. A processor Id is between `0` and `CmiNumPe()-1`.

## 3.7 Input/Output

void CmiPrintf(char *format, arg1, arg2, ...)
This function does an atomic `printf()` on `stdout`. On machine with host, this is implemented on top of the messaging layer using asynchronous sends.

void CmiScanf(char *format, void *arg1, void *arg2, ...)
This function performs an atomic `scanf` from `stdin`. The processor, on which the caller resides, blocks for input. On machines with host, this is implemented on top of the messaging layer using asynchronous send and blocking receive.

void CmiError(char *format, arg1, arg2, ...)
This function does an atomic `printf()` on `stderr`. On machine with host, this is implemented

on top of the messaging layer using asynchronous sends.

### 3.8 Processor Groups

void CmiPgrpCreate(Pgrp *group)
Creates a processor-group with calling processsor as the root processor.

void CmiPgrpDestroy(Pgrp *group)
Frees resources associated with a processor group `group`.

void CmiAddChildren(Pgrp *group, int penum, int size, int procs[])
Adds `size` processors from array `procs[]` to the processor-group `group` as children of processor penum. This function could be called only by the root processor of processor-group `group`.

CommHandle CmiAsyncMulticast(Pgrp *group, unsigned int size, void *msg)
Initiates asynchronous broadcast of message `msg` of length `size` bytes to all processors belonging to `group` excluding the processor on which the caller resides. It returns a communication handle which could be used to check the status of this send using `CmiAsyncMsgSent()`. `msg` should not be overwritten or freed before the communication is complete. (Note: *Caller need not belong to* `group`.)

int CmiPgrpRoot(Pgrp *group)
Returns the processor id of root of processor-group `group`.

int CmiNumChildren(Pgrp *group, int penum)
Returns number of children of processor `penum` in the processor-group `group`.

int CmiParent(Pgrp *group, int penum)
Returns processor id of parent of processor `penum` in the processor-group `group`.

void CmiChildren(Pgrp *group, int node, int *children)
Fills in array `children` with processor ids of all the children processor `node` in processor-group `group`. This array should atleast be of size `CmiNumChildren()`.

## 4 Message Manager Calls

To use the following calls, include the file "SM.h".

MSG_MNGR *CmmNew(void)
This call returns a new initialized message manager that can store and retrieve messages.

CmmPut(MSG_MNGR *mm, void *msg, int tag, int size)
(AND)
CmmPut2(MSG_MNGR *mm, void *msg, int tag1, int tag2, int size)
This call puts the message `msg` into the message manager `mm`'s data structure along with its `tag` and `size` fields.

(Note: *In the following calls,* `tag` *parameters may be wildcarded by placing a value of* **CmmWildCard** *in them. The actual values of the tag(s) of the message, if any, are returned in the* `rettag` *parameters if they are non-NULL.*)

int CmmProbe(MSG_MNGR *mm, int tag, int *rettag)
(AND)
int CmmProbe(MSG_MNGR *mm, int tag1, int tag2, int *rettag1, int *rettag2)
This call returns the size of the message with tag `tag` that is stored in the message manager `mm`, and returns -1 if such a message is not found.

int CmmGet(MSG_MNGR *mm, void *addr, int tag, int size, int *rettag)
(AND)
int CmmGet2(MSG_MNGR *mm, void *addr, int tag1, int tag2, int size, int *rettag1, int *rettag2)
This call copies at most `size` bytes of a message stored in the message manager `mm` with the tag(s) into the address pointed to by `addr`. The return value is the length of the message.

int CmmGetPtr(MSG_MNGR *mm, void *addr, int tag, int *rettag)
(AND)
int CmmGetPtr2(MSG_MNGR *mm, void **addr, int tag1, int tag2, int *rettag1, int *rettag2)
This call allocates memory for the message in the message manager `mm` with the tag(s) and returns this address in `*addr`. The return value is the length of the message.

# 5 Thread Manipulation

To use the following calls, include the file "thr_defns.h".

## 5.1 Thread Object Calls

int CthInit(void)
This call initializes some variables that are used by the thread calls library, and should be called before any other thread calls are made.

THREAD *CthCreate(THRFN fn, void *arg)
(AND)
THREAD *CthCreateOfSize(THRFN fn, void *arg, int stacksize)
These are calls to create a thread. This function takes a function pointer `fn` and its void pointer argument `arg`. The second call can be used if a stack of size other than the standard STACKSIZE is to be allocated for the thread.

int CthResume(THREAD *thr)
This call causes an immediate context switch to the specified thread `thr`. The thread `thr` continues to run until it, in turn, gives up control using CthResume or some variant of CthResume.

int CthSuspend(void)
This function is a variant of CthResume — it immediately causes a context-switch to some other thread. This function differs from CthResume in only one way: it makes its own decision about which thread to transfer control to. It always chooses a thread from a "ready pool" which is maintained by the user.

By default, CthSuspend always selects the thread which has been in its ready-pool the longest. This selection strategy may be altered by the user on a per-thread basis by calling CthSetStrategy below.

int CthAwaken(THREAD thr)
The thread is added to CthSuspend's ready-pool. This essentially constitutes permission for CthSuspend to transfer control to thread `thr`. CthAwaken must only be called on a thread when it can be shown that it is indeed acceptable for the thread to continue execution.

THREAD *CthSetStrategy(THREAD thr, THRFN suspfn, void *susparg, THRFN awakefn, void *awakearg)
CthAwaken and CthSuspend work together. By default, CthAwaken "adds a thread to the ready pool" by pushing it on a FIFO queue. By default, CthSuspend "finds a thread in the ready-pool" by popping this same FIFO queue. Together, this behavior guarantees that CthSuspend always can find a thread which is in the ready-pool.

Using CthSetStrategy, you may alter the way CthAwaken and CthSuspend work together. The purpose of such modification is to give you control over the order in which CthSuspend selects threads for execution. Note that you should not otherwise change the semantics of CthAwaken and CthSuspend: only the order of selection should be altered.

Each time a CthAwaken is performed on a thread t, thread t's `awakefn` is called. The `awakefn` must perform the task of CthAwaken: it must store the thread t in a location such that it can later be found by CthSuspend.

Each time a thread t calls CthSuspend, thread t's `suspfn` is called. The `suspfn` must look for a ready thread to transfer control to. It does this by looking in a location where CthAwaken stores threads. Once found, the `suspfn` must resume the thread using CthResume.

Note that CthSetStrategy overrides the behavior of CthAwaken and CthSuspend only on a per-thread basis. In a modular program, it is therefore possible for each module to control the order in which its own threads are scheduled.

int CthExit(void)
This call is used by a thread that has finished execution. The thread ceases to exist, and transfer is controlled to some other thread using CthSuspend. If the thread that exited had a special scheduling strategy, that strategy is used to choose the next thread.

int CthYield(void)
This call simply calls CthAwaken on the current thread (thereby adding the current thread to CthSuspend's ready-pool), after which it calls CthSuspend. This may cause a transfer of control to another thread. Control will probably come back to the thread that yielded, given that it is now in CthSuspend's ready-pool.

THREAD *CthSelf(void)
This call returns a pointer to the currently executing thread.

# 6 Synchronization Mechanisms

To use the following calls, include the file "sync.h"

## 6.1 Locks

Locks (or mutexes) are synchronization mechanisms that can be used by user programs to provide mutual exclusion to critical sections. Threads that attempt to "lock" such a variable are suspended if the lock is already taken and are awakened when the lock becomes available to them.

LOCK *CtsNewLock(void)
This call can be used to create a new lock variable.

CtsLockInit(LOCK *lock)
This call can be used to initialize a lock `lock` that was earlier allocated.

int CtsTryLock(LOCK *lock)
This call is a nonblocking attempt to lock `lock`. It returns 1 immediately if `lock` is available after making the current thread `lock`'s owner and returns 0 if `lock` is already locked.

int CtsLock(LOCK *lock)
This call is used by a thread to wait until it obtains the ownership of `lock`. Several threads making this call may be queued up at the lock, which is then "given" to each in turn.

int CtsUnLock(LOCK *lock)
This call is used by a thread to relinquish the control of `lock`. An error value is returned if the thread attempts the unlock is not `lock`'s owner.

## 6.2 Condition Variables

Condition variables are synchronization mechanisms that are used to implement trigger like functionality. Threads can wait on a condition variable. Other threads can either signal or broadcast this condition variable causing the awakening of either one or all of the threads waiting on this variable.

CONDN *CtsNewCondn(void)
This call returns a new initialized condition variable.

int CtsCondnInit(CONDN *condn)
This call can be used to initialize a condition variable that was earlier allocated. This call causes all the waiting threads on this condition variable to be awakened.

int CtsCondnWait(CONDN *condn)
This call is used by thread that want to wait on the condition variable `condn`.

int CtsCondnSignal(CONDN *condn)
This call releases one of the threads waiting on the condition variable `condn`.

int CtsCondnBroadcast(CONDN *condn)
This call releases all the threads waiting on the condition variable `condn`.

### 6.3 Barriers

Barriers are a specialization of condition variables. A barrier is a condition variable whose *k*th wait is a broadcast for some initial k. That is, the barrier waits for k threads to reach a particular point before it lets them all go.

BARRIER *CtsNewBarrier(void)
can be used to create a new barrier.

int CtsBarrierReinit(BARRIER *bar, int num)
This call (re)initializes the barrier `bar` to free any threads waiting on it and then to await the arrival of `num` threads.

int CtsAtBarrier(BARRIER *bar)
Following the initialization of the barrier, the `num` participating threads need to make this call before they can proceed beyond this point in the program. This call hence blocks all but the last thread to make this call, and awakens them all upon the arrival of this thread at the barrier.