CONVERSE : An Interoperable Framework for Parallel Programming

Draft

Laxmikant V. Kale and Narain Jagathesan and Sanjeev Krishnan
Department of Computer Science,
University of Illinois, Urbana-Champaign.
Email : {kale,narain,sanjeev}@cs.uiuc.edu
Phone : (217) 244-0094

November 7, 1996

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 document describes a proposed framework that will support such multi-lingual interoperability. The framework is meant to be inclusive, and has been verified to support paradigms including: SPMD programming style, message-driven programming, parallel object-oriented programming, and thread-based paradigms. Suggestions and criticisms of this design are anticipated from the community of language designers, which we hope to incorporate into the design. The framework aims at extracting only 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 [GS91, Mes93], data-parallel [HKT92, Hig93, BBG$^+$92], message-driven [Kal90, KK93], object-oriented [Chi93, Yon90, CK92, Gri93, Agh86], thread-based (Chant [HCM94], CC++), macro-dataflow (Nikhil's P-RISC), 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 be able 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. Therefore 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 document describes *Converse*, an interoperable framework for combining multiple languages and their runtime libraries into a single parallel program, which is under development at the Parallel Programming Laboratory at the University of Illinois, Urbana. 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 new 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 (a data parallel language), PVM, Nxlib, and possibly MPI in the near future. 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 core runtime system and scheduling model we have developed. Section 4 describes the thread objects used for supporting thread-based languages. Section 5 describes a minimal machine interface which is used as the abstraction for the underlying parallel machine. The appendix gives details of all the runtime library calls provided by the various modules in the framework.

2 Model of Computation

This section describes the general parallel computational model our framework will support. For the sake of simplicity, we confine this description to a private memory model (section 3.2.1 describes how it extends to a shared memory architecture). A computation consists of multiple processes, which communicate by explicit message passing. A parallel program in this model consists of a set of parallel modules written possibly in different languages. It is possible to have more than one module of a single language.

In this context how do modules from different languages coexist? Languages and their implementations differ from each other in many aspects. However, 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 : 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 object-oriented languages such as Charm allow concurrency by permitting more than one object to be waiting for messages simultaneously, and handle it via message driven scheduling. Such languages have many objects active on a processor, any of which can be scheduled depending on the arrival of a message. There is no stack associated with an object : it is assumed that the entire state of the object is encapsulated inside its local data.

- 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. Languages and paradigms such as data parallel languages and functional languages can be implemented in one of the above categories. For example, HPF can be implemented using a statically scheduled SPMD style and DP using message driven objects on top of Charm.

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. We have identified two regimes for modules in different languages to interact in the same parallel program : *explicit*, and *implicit* interoperability.

The *explicit interoperability regime* consists of non-overlapping modules of a parallel program from different languages, as described in Figure 1. 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. All processors transfer control to the next module only when the current module has completed all its work ; there are no outstanding messages, hence a module cannot receive a message sent by another module. This explicit interoperability is sufficient for many scientific applications which can be programmed in a loosely synchronous manner ; it enables 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.

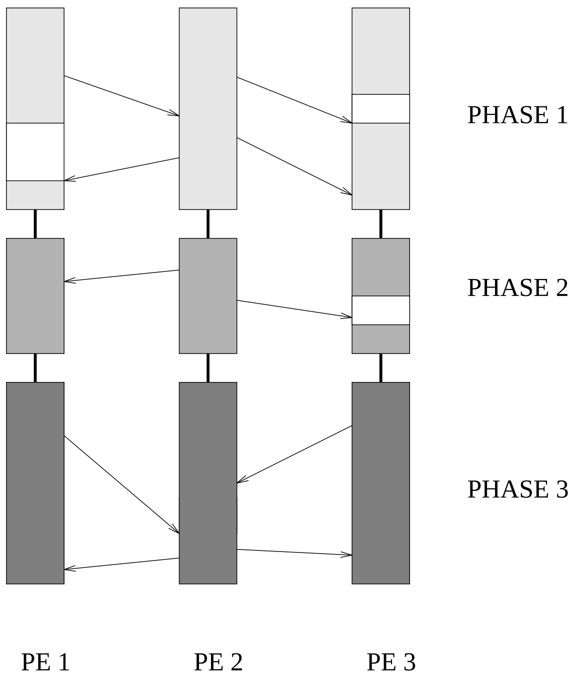


Figure 1: Explicit interoperability

The *implicit interoperability regime* (Figure 2) is motivated by a need to reuse parallel software components from different languages in an overlapped manner, so that entities in different modules can be simultaneously active. The transfer of control from module to module is implicit : it is not decided by the application program, instead 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, for example, when a thread in a module executes a “receive” statement and is waiting for a message, code from another module can be executed during that idle time. Implicit interoperability is suitable for languages having concurrent objects or threads which allow concurrency within a process[1].

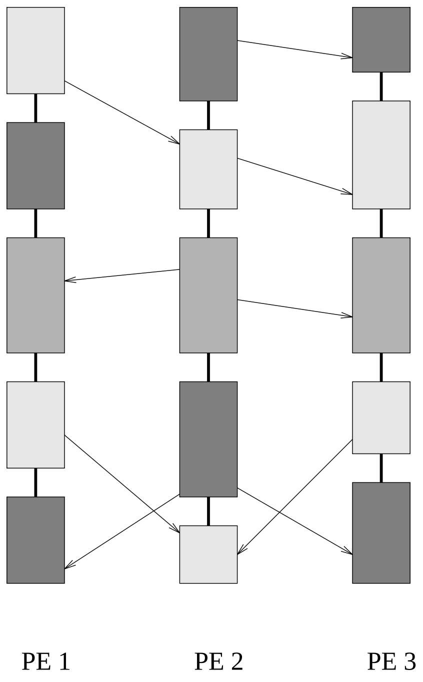


Figure 2: Implicit interoperability

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. How can these apparently incompatible regimes coexist in a single framework ? For concreteness, imagine a PVM process and a Charm module. It is clear that for the Charm computations to execute, the PVM module must relinquish control to the scheduler. For this purpose, we provide the scheduler as a user callable function instead of keeping it buried inside the run-time system. A typical interaction between the two control regimes is shown below.

```
Compute
Send
Compute
Receive
Send
Receive
Compute
Invoke function in Charm module to initiate its computation by sending messages
Invoke scheduler
Extract results from Charm computation
Continue with SPMD processing
```

[1]Preemptive threads is the only control regime not included in the computational model to be supported, at least for now. The reasons for this decision include program complexity and implementation complexity. Preemptive threads can interleave in many unexpected ways that lead to difficult bugs. Also, their implementation requires operating system support for cheap user-level interrupts. Note though that the model does not preclude the underlying implementation from using preemptive threads underneath. For example, the "processes" mentioned above may be implemented on a shared memory machine via preemptive threads.

The three lines including the scheduler invocation may also be separated out into a function to be provided by the Charm module writer for the explicit purpose of interfacing with SPMD programs. The Charm module initiates its computations by sending some messages into the system. When the scheduler is invoked, these deposited messages trigger the Charm computation.

Modules from one language may interact with those in another language through function invocations. For now, we will assume that that is the only method of interaction — in particular that modules in one language cannot send a message to a module in another language. This restriction will be eliminated at a later time.

To summarize, then, the computational model allows three kinds of entities : SPMD modules, 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 communicate via sends and receives (blocking as well as nonblocking) and invoke the scheduler at their will at any time. In the next section we describe the run time framework that follows from this computational model.

3 The core runtime system

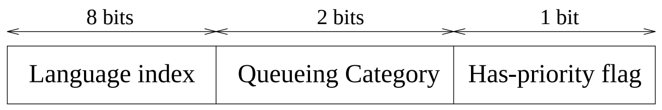
This section describes the software components required for supporting the two models of interoperability described above.

3.1 Message Formats

In order to allow implicit interoperability between modules from different languages, it is necessary for the core runtime to be able to distinguish messages from different languages. To achieve the latter, message formats must adhere to a minimum standard so that the core runtime system can extract the required information from the message. The fields required by the core include :

- a *language index* which identifies the language-specific handler to be invoked when the message is delivered
- a *queueing category* which specifies whether the message is to be queued or it is non-queueable. On machines which support interrupts this field also specifies whether the message is a preemptive message, in which case the machine layer directly invokes the language-specific handler as soon as the message is received.
- a flag to indicate whether the message is prioritized or not.
- an optional message priority field can be also used when prioritized queuing strategies are needed. The format of this will be specified later.

In the initial prototype implementation these fields will be placed in the first two bytes of the message as follows :



The core runtime will export macros for accessing and setting these fields, and a global variable which indicates the length of the core-specific fields. Thus the format of these fields does not need to be part of the standard. The macros are described in the Appendix.

3.2 Scheduling

In order to support the implicit interoperability model, we need a scheduler which dynamically decides the order of execution of application modules, and a loop which repeatedly polls the underlying machine interface for messages. Figure 3 describes the software architecture involving all these components. Note that these components could also be used by message-driven or thread based languages instead of their own language-specific schedulers.

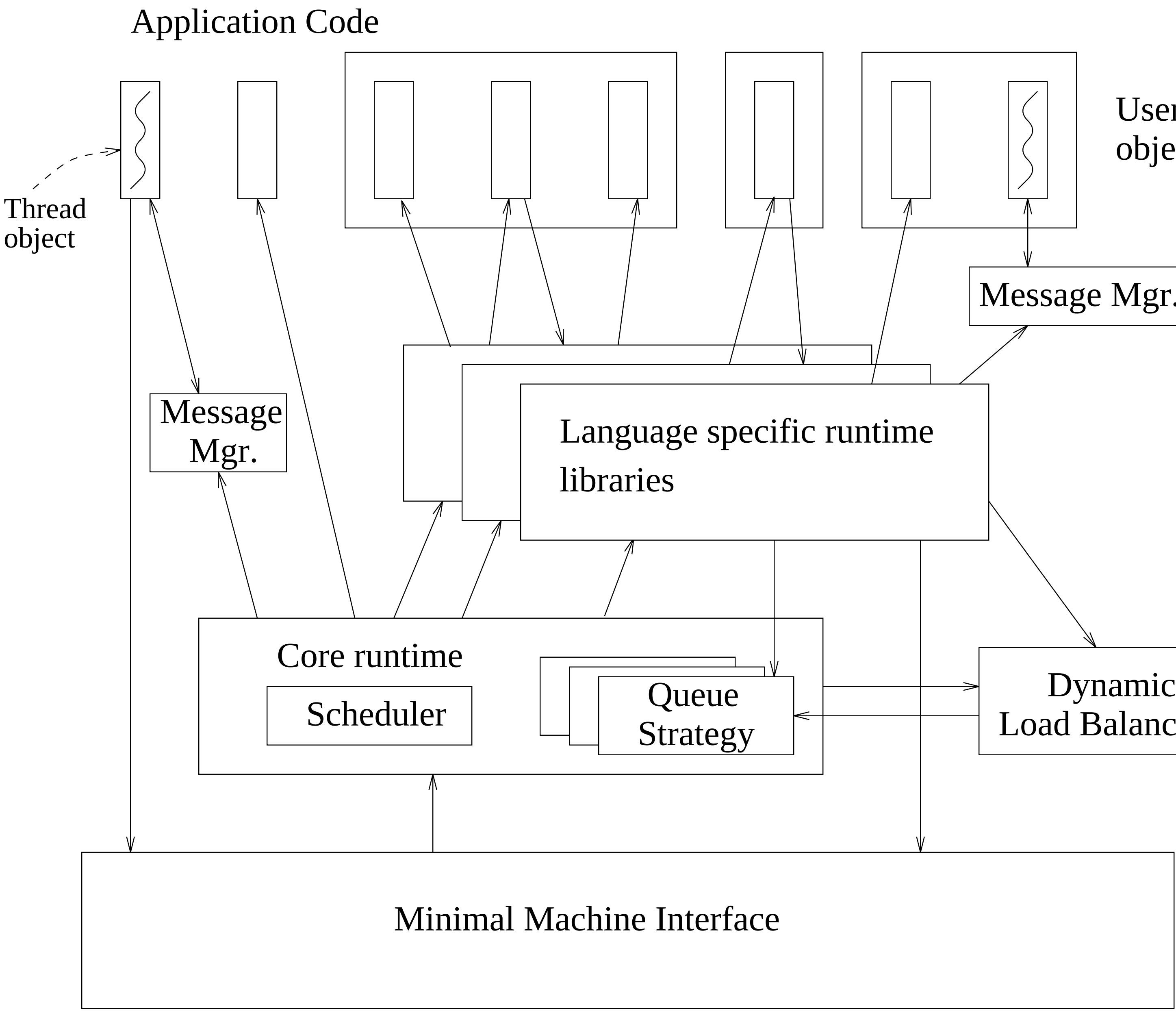


Figure 3: Software Architecture for Interoperability

The scheduling support in our framework consists of two components :

- A scheduler which repeatedly picks messages from the underlying machine's interconnection network, inserts them into a queue, removes a message from the queue and delivers it to the

```
SchedulerLoop()
{
        while ( not done ) {
                while ( there is a message available from the machine layer ) {
                        Get message from the machine layer
                        if (message is non-queueable)
                                Deliver it to language-specific handler
                                 (Transfer control to language runtime)
                        else
                                Enqueue the message into scheduler queue
                }

                Dequeue a message from scheduler queue
                Deliver it to language-specific handler
        }
}
```

Figure 4: Pseudo code for scheduler loop in core runtime system

appropriate language-specific library. This scheduler is described in pseudo code in Figure 4.

- A set of queueing libraries which define the queueing strategy to be used, such as FIFO, LIFO, priority-based, etc. The specific queuing strategy is chosen by the application programmer at link time.

The need for user-selectable queueing strategies is an important factor in the design of the scheduler. For example, many applications critically depend on the support for a prioritization mechanism to select an action to execute among the many possible ones on a single processor. These 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 [SK93]; state space search problems, where bit-vector priorities are needed to ensure consistent and monotonic speedups [SK90]; and numerical computations with critical paths where priorities can be used to speed up the critical path [Gur94]. Such prioritization mechanisms can be provided only by allowing the application to select the type of queueing strategy it wants to use.

At the same time, the framework does not penalize languages/paradigms that do not need scheduling (prioritized or otherwise). Such a language's runtime library would simply set the category of their messages to be non-queueable, which allows the messages to bypass the queues. The message is delivered to language-specific runtime as soon as the core scheduler finds the message (i.e. gets it from the network).

It may be noted that the above message-driven scheduling model is similar to a non-preemptive thread-based scheduling model. In a thread-based model, each processor has several threads which are scheduled by a scheduler in a non-preemptive manner. When a scheduler dispatches a thread, the thread executes atomically until it explicitly relinquishes control, (perhaps when it is waiting for a message) or completes execution, whereupon control returns to the scheduler, which schedules the next thread. A priority queue is used to decide which thread is to be scheduled. We model the "ready queue" in the thread scheduler as a message-queue ; in our model, a thread is ready to be executed only if there is a message for it. In order to relinquish control even if there is no message expected, a thread can inform the scheduler to add itself to the scheduler's queue (or, for example, send a "message" to itself) and then relinquish control. Thus the message-driven scheduling model we propose subsumes the functionality of a thread-based model.

Could we not use an existing threads package, say one compatible with Posix threads, for the purpose of this scheduling ? Yes, provided it supports user-selectable queueing strategies, and a mechanism for threads to wait for messages and resume after their arrival (see the discussion on message managers in section 3.4).

The core runtime library calls provided by the framework are described in the Appendix.

3.2.1 Shared memory architecture

The preceding discussion has concentrated on a private memory architecture. For machines with a shared address space, a similar software architecture will be required. The implementation can, however, be optimized by directly delivering messages to the core runtime. Additional features like "get" and "put" operations, and global pointers will also be supported. The suggestions made in the recent proposal for PORTS1 [BMH95] seem to be adequate for this purpose.

3.3 Standards for Language Implementors

In order to use various performance feedback, simulation and debugging tools on programs developed in the above framework, it is necessary to develop a standard for an event trace format. This will consist 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. For the sake of uniformity (so that generic tools can be developed), each entity in the system that is the source or destination of a message must be classified as one of the three types (SPMD module, thread, or object). In addition to recording message send and receive events, object or thread creation and dispatch must also be recorded.

3.4 Message Managers

Message managers are entities that are used in the system to organize a subset of messages that are yet to be processed. A message manager typically stores the messages in an indexible data structure, retrievable on one or more indices. For example, a message manager for PVM might store messages such that they are retrievable by "source processor" and "tag". The language specific routines attached to this handler may query the message manager for particular messages. A scheduler of the message manager handles the cases when a requested message has not yet arrived by storing the request and suspending the routine until the message that was requested arrives.

A message manager based scheduler can be implemented in conjunction with a message manager that stores unclaimed messages that have been received and requests for messages. The scheduler is given control whenever a message arrives for it. The scheduler checks to see if a thread has been waiting for this message and if so, resumes the thread, else stores the message until a request for it arrives. The thread then executes until it (a) completes or (b) needs to wait for another message at which point of time it suspends informing the scheduler about the message it is waiting for, or (c) it needs to wait for a lock. Threads which wait for locks are eventually added to the *ready list* of threads when the lock becomes available.

A listing of the calls to be provided by the message manager is provided in the Appendix in Section A.3.

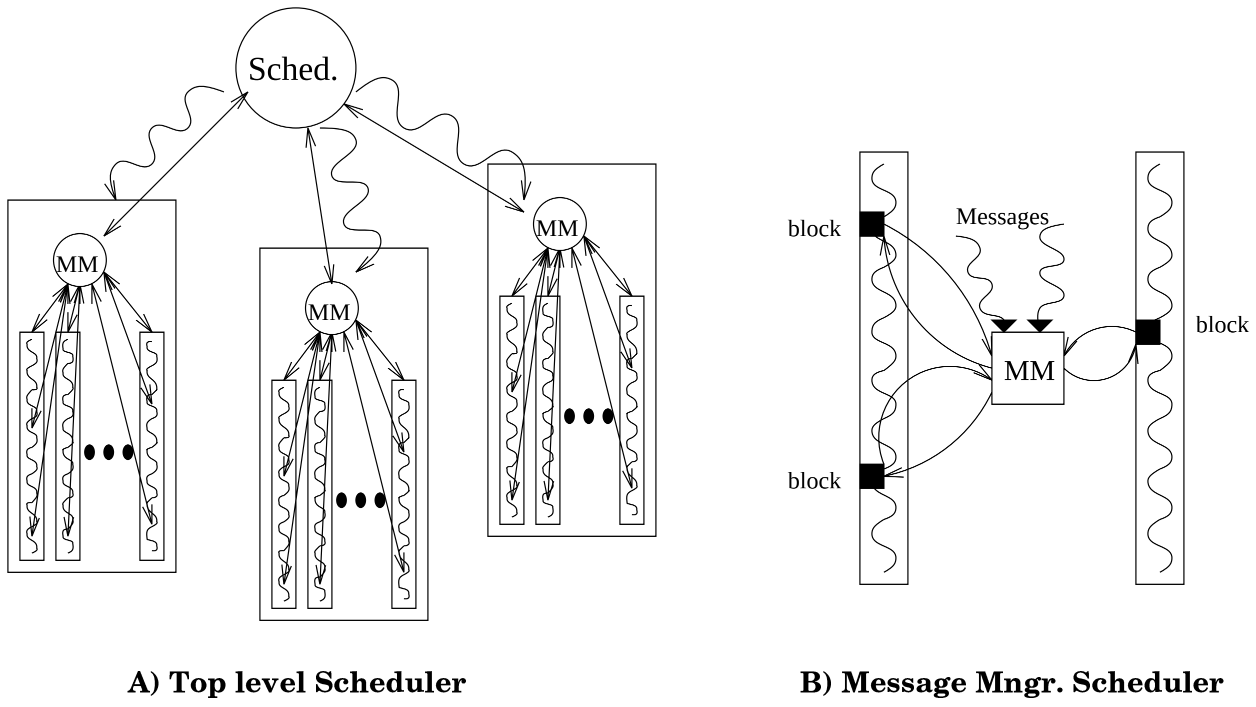


Figure 5: Message managers

4 Thread Objects

Most programs have a single thread of control. This is reflected in them having a single stack and a single program counter. However many complex programs are difficult to express in a single threaded manner. This is particularly true for programs that involve asynchronous events, internal or external (such as receipt of messages) . Many such programs can be expressed or thought of easily as a collection of multiple objects, each one with its thread of control and making progress at a rate independent of that 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 inspite of the interleaving of control among them. Expressing a program in this fashion is facilitated by the threads packages.

A threads package typically consists of these components

- An ability to freeze or suspend the execution or running of an object and to resume the execution of a previously suspended object.
- A scheduler that manages the transfer of control among the objects.
- A concurrency control mechanism such as locks.
- Process management if the threads are created on multiple processors.

Many thread packages have been developed in the past few years. A standard for threads — Posix threads or pthreads — 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 an interoperable run time system. The particular scheduling strategy provided by the threads package may not be appropriate for the problem at hand. We suggest that the capabilities of thread packages be modularly separated. In particular, we propose the definition and implementation of a *thread object* [Thr95] 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.

4.1 Thread object functionality

The modular abstraction of the thread object encapsulates only the stack, program counter and registers. It contains the following fields **(a)** an integer id **(b)** a stack_top **(c)** a pointer to the resumer and **(d)** state buffers for holding state of the caller and of this thread.

The switching of threads 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. The basic calls provided are those to (a) **Create** a thread, (b) **Resume** a thread, (c) **Yield** control to scheduler (d) **Suspend** current thread and (e) **Exit** current thread. We note that, in this model, when a thread yields, suspends or exits, control returns to the caller that last resumed the thread. This feature allows *hierarchical* creation and flexible scheduling of threads.

Making the functionality for these calls available separately allows one to create a number of threads and schedule them using different schedulers including the message manager scheduler discussed earlier.

4.2 Synchronization mechanisms

Two types of synchronization mechanisms are supported — locks (or mutexes) and conditional variables.

4.2.1 Locks

Locks may be implemented by having queues attached to each lock. The thread trying to obtain a lock continues if the lock can be obtained. If not, it places itself in a queue (perhaps prioritized) for the lock, and control is returned to the thread's scheduler. A thread which releases the lock causes the shifting of ownership of the lock to the first thread in this queue and moves this thread to its scheduler's ready queue so that it can continue executing when it is scheduled. The information about a thread's scheduler is available from the thread's data area.

4.2.2 Condition variables

Condition variables allow semantics where several threads block on a single condition. This case might occur in a messaging environment for instance, when several threads are blocked waiting for a particular information to become available through the arrival of a message.

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 of all the threads that are waiting on the condition variable.

Queue implementations of locks and condition variables remove the thread from the control of its scheduler after causing it to block. Therefore the implementation needs to differentiate between a normally yielding thread which gets put right away on the (perhaps prioritized) ready queue, and a thread which gets blocked on a lock or condition variable. Another type of thread suspension is when a thread suspends while requesting a message. This causes the thread to be registered with the message manager and the thread returns to the ready queue when the requested message arrives.

4.3 Compatibility with POSIX threads

The functionality outlined above is an extendible subset of the functionality provided by other thread packages like **pthreads** and the **PORTS0** interface, the only notable difference being that the scheduler is separated out. This allows one to have different implementations of thread schedulers, and further allows a certain amount of freedom in the manipulation of thread objects in the user program when so desired.

The extra calls to be exported are the calls to **resume** a thread and to **suspend** a thread without the thread being placed on the ready queue of a scheduler. These calls are used internally in the scheduler of any thread package.

A description of the function calls associated with threads can be found in the Appendix in Section A.4.

5 MMI: A Minimal Machine Interface

The Minimal Machine Interface defines a minimal interface between the machine independent part of the runtime such as the scheduler, language dependent libraries, etc., 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. E.g. MPI provides a "receive" call based on context, tag and source processor. The overhead of maintaining messages indexed for such retrieval is unnecessary for message driven programs. The interface we propose to develop is minimal, yet it is possible to provide an equally efficient MPI style retrieval on top of this interface. We have implemented this interface on a variety of distributed as well as shared memory machines.

Following are the minimum facilities to be provided by the machine-layer. Each of these functions could be provided as a macro in a standard file such as `machine.h` or could be inlined to avoid the overhead of a function call. Not all the functions described here are exported to the user of this framework. Some are used internally by the other components of the run-time such as scheduler and load balancing.

Process Creation: The machine layer should allow creation of processes initially on each processor in the machine (or a network of machines) either from command-line (for host-less architectures such as SPMD) or from a program (for machines that need a host).

Distributed Memory Machines: Currently there are two main classes of distributed memory machines. We describe the process creation primitives needed for both these classes below.

SPMD Architectures: Most SPMD architectures allow creation of processes on multiple processors from command line. Typically processes are created on all the processors in the partition. Partition creation and modification is done external to the run-time system. Therefore, it is not a requirement for Minimal Machine Interface. However, the interface should allow for each process to assign a unique id (starting with 0) to itself. This ID could be set at startup (e.g. in `main`) and could be returned in `McMyPeNum()`. Each process should also be able to know the total number of processes in the system (in SPMD, this is equal to total number of processors in partition). But this does not present a significant difficulty because it could be passed as a command-line parameter and could be extracted at startup time.

Networks of Workstations: Launching processes on remote workstations through mechanisms such as `rsh` is a key requirement for network of workstations. Total number and addresses of nodes could be read from a `nodes` file. Parallel programs on networks of workstations can be launched on each node specified in `nodes` file from a special process on the user's workstation. Capability to terminate a process is also required if one plans to allow user interrupts. Since there is no way to determine the logical node ID of a process on such machines, the process which spawns remote processes also has to do some initial hand-holding to communicate this necessary information to the nodes. However, this can be done using the send and receive capability of the MMI.

Shared Memory Machines Shared memory machines should allow creation of processes or threads; at least one on each processor in the machine. Dynamic creation and destruction of threads is not necessary. The main program (which is launched from command line) should be able to start a number of threads running. Also there should be a way to determine that all threads have terminated. On most systems, this could be implemented using `Fork` and `Join` constructs. Each thread should have a unique ID. (IDs need not start from 0 because system developer can set up a translation table to convert them. But it would be more efficient to have thread identifiers in the range `0..nproc-1`.)

Communication Minimal communications interface differs significantly for two main types of machines; those with shared and distributed memory.

Distributed Memory Machines The underlying architecture should provide facilities for setting up and carrying out point-to-point communication between any two processes. In particular, a process should be able to send a message (an array of bytes) of any length across processes. The run-time system could take advantage of the availability of asynchronous sends to make programs more efficient. On the receiving side, a process should be able to check for incoming messages and also to check for the length of arrived message. With this ability of the machine layer, a simple blocking receive suffices. The run-time should be able to receive a message irrespective of its source. Therefore, the underlying architecture should allow source parameter in the `probe` or `receive` call to be wildcarded.

In most SPMD machines, a call such as *Probe* serves the purpose along with a blocking receive. However, in many machines, asynchronous receives serve both purposes and make runtime more efficient. Having a tag attached to each message is an extra overhead because our run-time does not use it. If the message-passing library requires use of additional system-level tags, they should be set to a constant for all messages.

The run-time system would be more efficient if the system allows invoking a handler on the arrival of a message. In this case, the handler should just buffer the arrived message and return it upon a receive request from the run-time.

Shared Memory Machines The machine programming interfaces (usually in the form of libraries) should allow for creation, locking and unlocking of spinlocks to handle critical sections. Other primitives for creating and maintaining critical sections are also acceptable as long as they allow for the following three calls: *InitCriticalSection*,*EnterCriticalSection* and *LeaveCriticalSection*.

Miscellaneous The MMI should provide calls to read the timer value. A timer value could be an integer (or a long integer) representing milliseconds (or some unit which can be easily converted to milliseconds). Absolute value of this timer is not important but the difference should correctly represent the time elapsed between two calls to timer functions.

Any process on the machine should be able to determine its processor number (starting from 0). For machines with hosts, any process should also know the processor number of the Host. Capability of input and output (in the form of scanf and printf) from any node is not essential. But the runtime can take advantage of this if such facility is available.

5.1 EMI: Extended Machine Interface

Functionality commonly used in developing parallel programs such as multicasts over process groups is not required in the MMI because they could be built on top of it. However, it is possible to produce a customized implementation of these functions for specific machines. We include such functions in an Extended Machine Interface (EMI). For any specific machine the implementor is free to use either the generic EMI implementation or to develop a customized implementation.

6 Summary and Future Work

The different modules of the interoperable framework described in this paper can be seen in Figure 6. Once an interface has been defined between these modules, it is easy to see that we can use different implementations of each module in the framework as long as these implementations adhere to the interface specifications.

Some issues remain to be worked out in the interoperable framework we have described. These include :

- A standardization of support for message priorities, including priority field formats.
- Support for pre-emptive, interrupt-driven scheduling
- Dynamic Load balancing needs to be coordinated across all languages (because the load in one module affects load balancing decisions in the other). Therefore support for language-independent dynamic load balancing is necessary. Also, as applications differ in their characteristics one should allow selection of a load balancing module at link time from a suite of libraries. Our current intention is to use the load-balancing framework from Charm with necessary extensions.
- Support for objects and first class continuations Object management functions are implemented differently for different languages; they may not be necessary for languages which are not object based. This module is responsible for looking up object data areas using object ids. The format of the object id may be language dependent. This module may, for example, also be responsible for object migration services : messages to migratable objects can be non-schedulable, so they are immediately delivered to the language-specific module. If the object to which a message is directed has been migrated to a different processor, the message is simply redirected on to that processor, otherwise it is enqueued into the scheduler queue. First class continuations are necessary for objects in one language to request a reply from objects in another language. First class continuations include an object id and a method id. We will develop a standard format for continuations which will enable the run-time system to extract relevant information from the continuation structure.

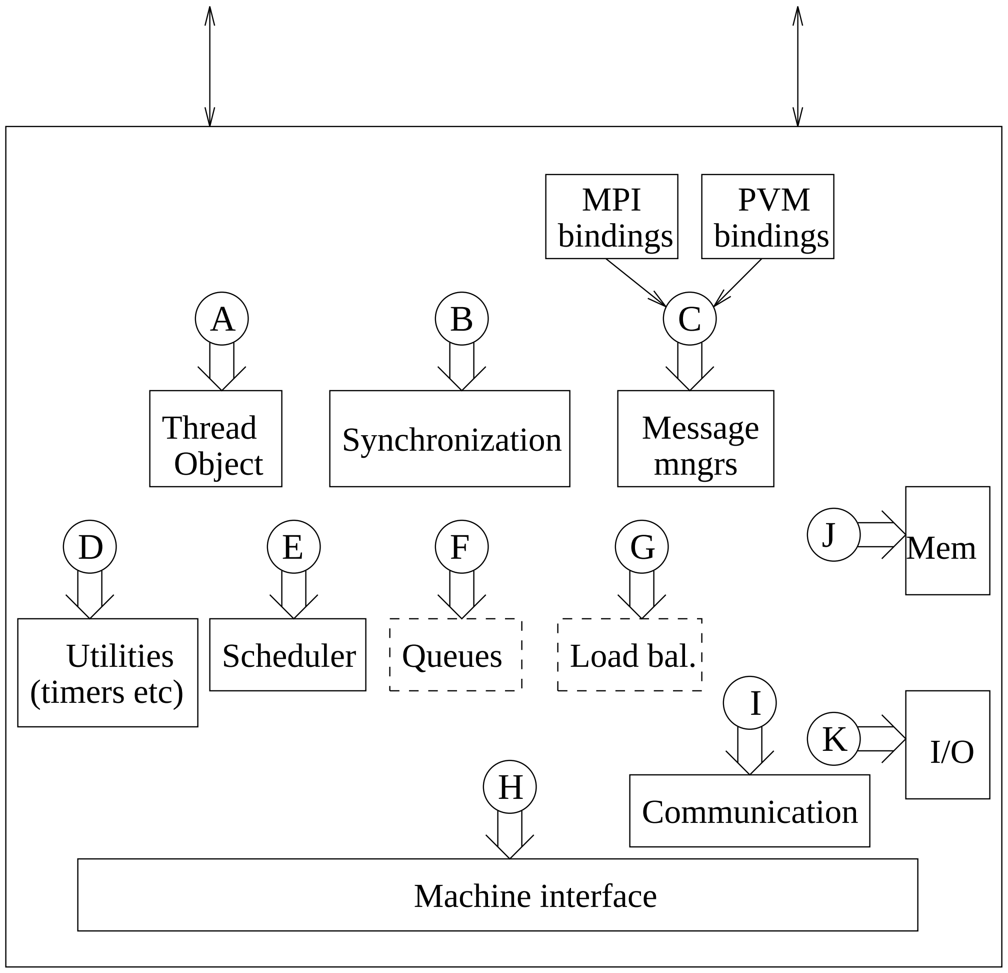


Figure 6: Modules in the framework

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[BBG+92] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic ideas for an object parallel language, 1992.

[BMH95] Pete Beckman, Bernd Mohr, and Matt Haines. A proposal for ports1. Technical report, PORTS Consortium, February 1995.

[Chi93] A. Chien. *Concurrent Aggregates.* MIT Press, 1993.

[CK92] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. Technical Report Caltech-CS-TR-92-13, Department of Computer Science, California Institute of Tech nology, 1992.

[Gri93] A. Grimshaw. Easy to use object oriented parallel processing with Mentat. *IEEE Computer*, May 1993.

[GS91] 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.

[Gur94] Attila Gursoy. *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance.* PhD thesis, University of Illinois, June 1994. also, Report No. UIUCDCS-R-94-1852.

[HCM94] M. Hainer, D. Cronk, and P. Mehrotra. On the design of chant: A talking threads package. In *Proceedings of Supercomputing 94*, November 1994.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification (Draft)*, 1.0 edition, January 1993.

[HKT92] S. Hiranandani, K. Kennedy, and C. Tseng. *Compiler support for machine independent parallel programming in Fortran-D.* Elsevier Science Publishers B.V., 1992.

[Kal90] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Proces sing*, August 1990.

[KK93] L.V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[Mes93] Message Passing Interface Forum. *Document for a Standard Message-Passing Interface*, November 1993.

[SK90] 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.

[SK93] Amitabh 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.

[Thr95] Thread objects. Technical report, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana, February 1995.

[Yon90] A. Yonezawa. *ABCL: An Object Oriented Concurrent System.* MIT Press, 1990.

A Function Call Interface

This appendix describes all the library calls provided by various modules in the described framework. Actual names of the calls may be changed for compatibility with the Posix or PORTS standards.

A.1 Message format macros

- GetLanguageIndex(message)
- SetLanguageIndex(message)
- GetQueueCategory(message)
- SetQueueCategory(message)

- GetPriorityFlag(message)
- SetPriorityFlag(message)

A.2 Core runtime library calls

The communication calls provided by the core runtime system to support interoperability include :

- void BasicSendMessage(void *MsgBuf, int MsgSize, int DestPE) : this assumes that the message buffer has been initialized with information for all the core-runtime's fields. This call is directly routed to the underlying machine interface.
- void SendData(void *Buf, int MsgSize, int DestPE, int DestLang, int QCat, int NewWorkFlag, int HasPriorityFlag) : this call assumes that the space for the core runtime system's fields has not been allocated ; it may allocate a new message buffer which includes those fields and copy the data from Buf, or it may use more efficient mechanisms if provided by the underlying machine layer.
- int BlockingReceive(int Language, int MsgBuf, int Size) : this call blocks the processor until a message for the specified language has been received. No other application code will execute until this call returns. The call copies the received message into MsgBuf ; a maximum of Size bytes are copied. The call returns the size of the received message.
- int NonBlockingReceive(int Language, int MsgBuf, int Size) : this call returns with a message for the specified language if one is available; if not, it returns -1 immediately.

The message-send functions perform a broadcast when the DestPE field is -1.

The scheduler related calls provided by the core runtime are :

- int Scheduler(int Niterations) : When modules in explicit and implicit models need to interact, this call allows the explicit module to give control over to the scheduler for the implicit modules. This call hands over control to the scheduler of the implicit modules to handle a certain number of messages, and the corresponding processing. A value of -1 for Niterations causes the scheduler to keep handling messages for the implicit modules until the application code specifies a termination condition.
- int ThreadBlockingReceive(int Language, int MsgBuf, int Size) : this call blocks only the calling thread until a message for the specified language has been received. Other threads in the program may execute before control returns to the calling thread.
- void RelinquishControl() : this is called by a thread when it wishes to relinquish control to the scheduler. The calling thread will be rescheduled at a later time.
- int RegisterHandler(FUNCTION_PTR handler) : this is called by an application module to register a language-specific handler with the core runtime. (Note : this must be called only once per language, even if there are more than one module of a language). It returns a language index which can be used to specify the language a message must be delivered to upon receipt.

A.3 Message manager calls

The message manager needs to export the following functions :

- MMNGR *create_mmngr(int num_indices, int *offset_array);
  creates and returns a pointerr to a message manager which will receive and store message indexed upon `num_indices` indices which are to found at the offsets in the message denoted by the **offset_array**.

- int insert(MMNGR *mmngr, void *msg);
  inserts the message into the message manager `mmngr`.

- int b_recv(MMNGR *mngr, index *index_array, void **msg, int *size);
  returns when a message is available at the message manager `mngr` that has the attributes stored in the array `index_array`.

- int probe(MMNGR *mngr, index *index_array);
  returns 1 if such a message is available or 0 if no such message is available in the message manager `mngr`.

- int nb_recv(MMNGR *mngr, index *index_array, void **msg, int *size);
  returns immediately with a return value of 1 and `*msg` set to the message if the requested message is available with the message manager. If the message is not available, the call returns a value of 0.

A.4 Thread Function Calls

A.4.1 Thread Object Functionality

- THREAD ck_thr_create(THRFN fn, void *args);

  (AND)

  THREAD ck_thr_create_of_stack(THRFN fn, void *args, int stacksize);
  are calls to create a thread. This function takes a function pointer and a void pointer as an argument to the function. The second function call can be used if a stacksize other than STACKSIZE is to be allocated for the thread.

- int ck_thr_resume(THREAD thread);
  is used to start a newly created thread or one that has run earlier and suspended. Returns the value -1 if the thread which was resumed has exited, 0 if it has yielded and is ready to run, and 1 if the thread has suspended but is not yet ready.

- int ck_thr_yield();
  can be called by a thread to yield control of the processor back to its scheduler. A thread that yields may be immediately added to the ready queue of the scheduler.

- int ck_thr_suspend();
  can be called by a thread to suspend itself. This call is made when the suspending thread is not to be placed immediately on the ready queue.

- int ck_thr_exit();
  can be called by a thread which is exiting.

- THREAD ck_thr_self();
  returns the id of the currently executing thread.

A.4.2 Scheduler Functions

- int ck_add_to_q(THREAD thr);
  is to be provided by a scheduler so that a thread which has been waiting on synchronization mechanisms can be placed on the ready queue.

A.4.3 Synchronization Mechanisms

- **Mutexes (or locks)**
  - int ck_mutex_init(MUTEX *mutex);
    to initialize and return a mutex variable.
  - int ck_mutex_destroy(MUTEX *mutex);
    to destroy the mutex variable. No further use of the mutex is allowed.
  - int ck_mutex_lock(MUTEX *mutex);
    a blocking call which returns when the mutex has been obtained by the calling thread.
  - int ck_mutex_trylock(MUTEX *mutex);
    a nonblocking call which returns with 1 if the mutex is locked already or 0 if the mutex is obtained.
  - int ck_mutex_unlock(MUTEX *mutex);
    Unlock a mutex which the thread currently holds.
- Condition variables
  - int ck_condn_init(CONDN *condn);
    to initialize and return a condition variable.
  - int ck_condn_destroy(CONDN *condn);
    to destroy the condition variable. No further use of the condition variable is allowed.
  - int ck_condn_wait(CONDN *condn, MUTEX *mutex);
    automatically releases the mutex, and returns when the condition variable being waited on is released by a ck_condn_signal() or a ck_condn_broadcast().
  - int ck_condn_signal(CONDN *condn);
    unblocks one thread that is blocked on the condition variable pointed to by the condition variable.
  - int ck_condn_broadcast(CONDN *condn);
    unblocks all the threads that are waiting on the condition variable.

A.5 MMI Calls

This is the list of functions that are needed for the minimal machine interface.

Distributed Memory machines: Following functions are required by the run-time on distributed-memory machines.

`void McInit(int argc, char*argv[]):` Initializes MMI.

`void *McGetMsg(void):` Returns with a message addressed to calling process and removes the message from the pending queue. Memory for receiving this message has to be allocated inside this function. If no message is pending, this function returns `NULL`.

`void McSyncSend(int destPE, int size, char *msg)::` Send message `msg` of size `size` to processor `destPE` synchronously. (A blocking send is acceptable.)

`void McAsyncSend(int destPE, int size, char *msg):` Send message `msg` of size `size` to processor `destPE` asynchronously. This function need not copy the message to a local buffer. Also, keep the communication handle for future reference. (It is needed for `McReleaseSentMessages`, `McAllAsyncMsgsSent`)

`Boolean McAllAsyncMsgsSent(void):` Returns `TRUE` if all the asynchronous message sends have been completed. Returns `FALSE` otherwise.

`void McReleaseSentMessages(void):` Removes asynchronously sent messages for which communication is complete from the internal list. Also frees the data area associated with these messages.

Broadcast calls: The MMI should provide capability to broadcast a message to all the processes belonging to a parallel program. In addition, all processes except for the calling process should be able to receive this message with `McGetMsg`. The following functions should be provided by the MMI:

`void McBroadcast(int size, char *msg, Boolean Free, Boolean Self):` Send message `msg` with size `size` to all processes. If `Self` is FALSE, do not send this message to the calling process.If `Free` is TRUE, free the message after communication is complete. This function can be implemented using `McSyncSend` in the absence of specialized broadcast functionality from the system.

`void McAsyncBroadcast(int size, char *msg):` Broadcast asynchronously to all the processes except for the calling process. Could be implemented using `McAsyncSend` in absence of specialized broadcast facility in the system. Do not free the message.

Anticipatory receive calls: A feature to be included in MMI is anticipatory receives to avoid copying. There are three forms of such receives :

`int McAsyncReceive(int LanguageIndex, ...Filter..., ...Action... )` : where the filter identifies the message (for example, using a tag and source processor number) whereas the action specifies where to copy which parts of the message. Details for specifying the filter and action fields are yet to be finalized. The call returns an id which can be used to query if the receive has completed.

`McExpect(int LanguageIndex, ...Filter..., ...Action... )` : this is similar to the above except that it causes a token message to be delivered to the scheduler when the message is received.

`McBlockingReceive(int LanguageIndex, ...Filter..., ...Action... )` : this call blocks until the message is received.

Shared Memory Machines: Following functions are required by the run-time on shared-memory machines.

`void Fork(int nprocs, void *(func)()):` Forks `nprocs` processes/threads which execute `func`.

`void Wait(int nprocs):` Waits for `nprocs` threads to terminate.

`void McSpinInit(Lock l):` Initializes spin lock `l`.

`void McSpinLock(Lock l):` Locks spinlock `l`. If it is already locked, blocks until it is released. Busy waiting is allowed if exactly one process is created on each processor.

`void McSpinUnlock(Lock l):` Unlocks spinlock `l`. Activates any processes waiting on that lock.

Common: These functions are required by the run-time on both shared and distributed memory machines.

Input/Output: Following three primitives for input/output should be provided by the machine-layer.

`int McFlushPrintfs(void)`: Flushes any pending print requests from current node. On most systems where I/O is allowed from any node, this will correspond to `fflush(stdout)`. On systems where I/O has to be implemented using messages to the main PE, this function will have to check that all printf messages sent asynchronously have been completed.

`void McPrintf(char *format, arg1,arg2,...)` This can be `#defined` to `printf` if the underlying environment allows printing from any node of the system. Otherwise, it typically converts the argument list to a character string to be output on the host and sends it as an asynchronous message to the main PE. Also keeps track of all the printf messages sent. (See `McFlushPrintfs`).

`void McScanf(char *format, arg1, arg2,...)` This can be `#defined` to `scanf` if the underlying environment allows input on any node of the system. Otherwise, it must send the format string as a synchronous message to host PE and block on reply from the host (or the main PE). On reply from host, it should fill in the arguments and return.

Timer Calls: Following calls that provide both per-process and wallclock timer should be implemented in the machine-layer.

`void McTimerInit(void)`: Initializes a per-process timer and a wall-clock timer. Both timers return values in milliseconds since start of program. Thus both of them should be initialized to zero. If a more accurate timer is available, it should be used to serve `McUTimer` call and should be initialized in this function.

`unsigned int McTimer(void)`: Returns current value of wall-clock timer in milliseconds.

`unsigned int McUTimer(void)`: Returns current value of wall-clock timer in microseconds.

`int McHTimer(void)`: Returns current value of wall-clock timer in hours truncated to integer. Should return (-1) if not implemented.

`unsigned int McPTimer(void)`: Returns current value of per-process timer in milliseconds. For most systems it is the sum of system and user times spent by process since its beginning.

`int McMyPeNum(void)`: Returns index of processor on which the calling process is running.

`int McTotalNumPe(void)`: Returns total number of processors on which this parallel program is running.