Chapter 10

Message Passing and Threads

Ian Foster, William Gropp, and Carl Kesselman

In this chapter we examine two fundamental, although low-level, approaches to expressing parallelism in programs. Over the years, numerous different approaches to designing and implementing parallel programs have been developed (e.g., see the excellent survey article by Skillicorn and Talia [870]). However, over time, two dominant alternatives have emerged: *message passing* and *multithreading.*

These two approaches can be distinguished in terms of how concurrently executing segments of an application share data and synchronize their execution. In message passing, data is shared by explicitly copying ("sending") it from one parallel component to another, while synchronization is implicit with the completion of the copy. In contrast, the multithreading approach shares data implicitly through the use of shared memory, with synchronization being performed explicitly via mechanisms such as locks, semaphores and condition variables.

As with any set of alternatives, there are advantages and disadvantages to each approach. Multithreaded programs can be executed particularly efficiently on computers that use physically shared memory as their communication architecture. However, many parallel computers being built today do not support shared memory across the whole computer, in which case the message-passing approach is more appropriate.

From the perspective of programming complexity, the implicit sharing provided by the shared-memory model simplifies the process of converting existing sequential code to run on a parallel computer. However, the need for explicit synchronization can result in errors that produce nondeterministic race conditions that are hard to detect and correct. On the other hand, converting a program to use message passing requires more work up front, as one must extract the information that must be shared from the application data structures and explicitly move it to the desired concurrently executing program component. However, because synchronization is implicit in the arrival of the data, race conditions are generally avoided.

Both message passing and multithreading can be implemented via special purpose programming languages (and associated compilers) or through libraries that are linked with an application written in an existing programming language, such as Fortran or C. In this chapter, we focus on the most common library-based implementations, specifically the *Message Passing Interface* [881] (MPI) for message-passing programs and the POSIX standard thread library [503] (Pthreads) for multithreading programs. We also discuss popular alternatives, notably Parallel Virtual Machine [374] (PVM).

10.1 Message-Passing Programming Model

Message passing is by far the most widely used approach to parallel computing, at least on large parallel systems. (Multithreading dominates on small shared-memory systems.) In the message-passing model, a computation comprises one or more processes that communicate by calling library routines to send and receive *messages*. Communication is *cooperative*: data is sent by calling a routine, and the data is not received until the destination process calls a routine to receive the data. This is in contrast to other models, such as one-sided or remote-memory operations, where communication can be accomplished by a single process.

The message-passing model has two great strengths. The most obvious to users is that programs written using message passing are highly portable. Virtually any collection of computers can be used to execute a parallel program that is written using message passing; the message-passing programming model does not require any special hardware support for efficient execution, unlike, for example, shared-memory programming models. The second strength is that message passing provides the programmer with explicit control over the location of memory in a parallel program, specifically, the memory used by each process. Since memory access and placement often determine performance, this ability to manage memory location can allow the programmer to achieve high performance. The major disadvantage of message passing is that the programmer is *required* to pay attention to such details as the placement of memory and the ordering of communication.

Numerous different message-passing models and libraries have been proposed. At their core, most support the same basic mechanisms. For *point-to-point* communication, a *send* operation is used to initiate a data transfer between two concurrently executing program components and a matching *receive* operation is used to extract that data from system data structures into application memory space. In addition, *collective* operations such as broadcast and reductions are often provided; these implement common global operations involving multiple processes.

Specific models and libraries also differ from one another in a variety of ways.

For example, the send calls may be blocking or nonblocking, as can the receives. The means by which a send and receive are matched up may also vary from implementation to implementation. The exact details of how message buffers are created, filled with data and managed also vary from implementation to implementation.

Variations in the specifics of a message-passing interface can have a significant impact on the performance of programs written to that interface. There are three major factors that influence performance: bandwidth and latency of actual message passing and the ability to overlap communication with computation. On most modern parallel computers, latency is dominated by the message setup time rather than the actual time of flight through the communication network. Thus the software overhead of initializing message buffers and interfacing with the communication hardware can be significant. The bandwidth achieved by a specific message-passing implementation is often dominated by the number of times the data being communicated must be copied when transferring data between application components. Poorly designed interfaces can result in extra copies, reducing the overall performance of an application. The final performance concern is that of overlapping communication and computation. Nonblocking sending semantics enable the sender to continue execution even if the data has not been accepted by the receiver. Furthermore, nonblocking receives enable a receiver to anticipate the next incoming data elements, while still performing valuable work. In both situations, the performance of the resulting application is improved.

While message passing can be, and indeed has been, provided as primitives in a programming language (e.g., Occam [508], Concurrent C [375]), it is more typically implemented by library routines. In the following we focus on the two most widely used message-passing libraries: Message Passing Interface (MPI) and Parallel Virtual Machine (PVM).

10.1.1 The Message Passing Interface Standard

While the message-passing model is portable in the sense that it is easily implemented on any parallel platform, a specific program is portable only if the system that supports it is widely available. For example, programs written in C and Fortran are portable because C and Fortran compilers are widely available. In the early history of parallel computing, there were numerous different implementations of the message-passing model, with many being specific to individual vendor systems or written by research groups. This diversity of message-passing implementations prevented applications from being portable to a wide range of parallel computers.

Recognizing that a single clearly and precisely specified message-passing interface would benefit users (by making their programs portable to a larger set of machines) and vendors (by enlarging the set of applications that can run on any machine), a group of researchers, parallel computer vendors, and users came together to develop a standard for message passing. Following in the footsteps of the High Performance Fortran Forum, this group developed a standard called the

Message Passing Interface [669, 882], and the group was called the MPI Forum. The standard itself is available on the web at `http://www.mpi-forum.org`.

MPI defines a library of routines that implement the message-passing model. Rather than simply codifying features of the message-passing libraries that were in common use when MPI was defined, such as PVM, Express, and p4 [133, 158], the MPI Forum incorporated best practice across a variety of different message-passing libraries. The result is a rich and sophisticated library that includes a wide variety of features, many of which are important in implementing large and complex applications.

Although MPI is a complex and multifaceted system, just six of its functions are needed to solve a wide range of problems. We focus our attention here on those functions; for a more complete tutorial discussion of MPI, see [342, 409, 410, 732].

The six MPI functions that we describe here are used to initiate and terminate a computation, identify processes, and send and receive messages:

`MPI_INIT`	:	Initiate an MPI computation.
`MPI_FINALIZE`	:	Terminate a computation.
`MPI_COMM_SIZE`	:	Determine number of processes.
`MPI_COMM_RANK`	:	Determine my process identifier.
`MPI_SEND`	:	Send a message.
`MPI_RECV`	:	Receive a message.

Function parameters are detailed in Figure 10.1. In this and subsequent figures, the labels `IN`, `OUT`, and `INOUT` indicate whether the function uses but does not modify the parameter (`IN`), does not use but may update the parameter (`OUT`), or both uses and updates the parameter (`INOUT`).

MPI defines "language bindings" for C, Fortran, and C++ (bindings for other languages can be defined as well). Different language bindings have slightly different syntax, reflecting language peculiarities. Sources of syntactic difference include the function names themselves, the mechanism used for return codes, the representation of the *handles* used to access specialized MPI data structures such as communicators, and the implementation of the `status` datatype returned by `MPI_RECV`. (The use of handles hides the internal representation of MPI data structures.)

For example, in the C language binding, function names are as in the MPI definition but with only the `MPI` prefix and the first letter of the function name capitalized. Status values are returned as integer return codes. The return code for successful completion is `MPI_SUCCESS`; a set of error codes is also defined. Compile-time constants are all in upper case and are defined in the file "`mpi.h`", which must be included in any program that makes MPI calls. Handles are represented by special defined types, defined in `mpi.h`. These will be introduced as needed in the following. Function parameters with type `IN` are passed by value, while parameters with type `OUT` and `INOUT` are passed by reference (that is, as pointers).

All but the first two calls take a *communicator handle* as an argument. A communicator identifies the *process group* and *context* with respect to which

`MPI_INIT(int *argc, char ***argv)`
Initiate a computation.
`argc`, `argv` are required only in the C language binding,
where they are the main program's arguments.

`MPI_FINALIZE()`
Shut down a computation.

`MPI_COMM_SIZE(comm, size)`
Determine the number of processes in a computation.

`IN`	`comm`	communicator (handle)
`OUT`	`size`	number of processes in the group of `comm` (integer)

`MPI_COMM_RANK(comm, pid)`
Determine the identifier of the current process.

`IN`	`comm`	communicator (handle)
`OUT`	`pid`	process id in the group of `comm` (integer)

`MPI_SEND(buf, count, datatype, dest, tag, comm)`
Send a message.

`IN`	`buf`	address of send buffer (choice)
`IN`	`count`	number of elements to send (integer ≥ 0)
`IN`	`datatype`	datatype of send buffer elements (handle)
`IN`	`dest`	process id of destination process (integer)
`IN`	`tag`	message tag (integer)
`IN`	`comm`	communicator (handle)

`MPI_RECV(buf, count, datatype, source, tag, comm, status)`
Receive a message.

`OUT`	`buf`	address of receive buffer (choice)
`IN`	`count`	size of receive buffer, in elements (integer ≥ 0)
`IN`	`datatype`	datatype of receive buffer elements (handle)
`IN`	`source`	process id of source process, or `MPI_ANY_SOURCE` (integer)
`IN`	`tag`	message tag, or `MPI_ANY_TAG` (integer)
`IN`	`comm`	communicator (handle)
`OUT`	`status`	status object (status)

Figure 10.1: Basic MPI. These six functions suffice to write a wide range of parallel programs. The arguments are characterized as having mode `IN` or `OUT` and having type integer, choice, handle, or status. These terms are explained in the text.

the operation is to be performed. As we shall see later, communicators provide a mechanism for identifying process subsets when we are developing modular programs. They also ensure that messages intended for different purposes are not confused. For now, it suffices to provide the default value MPI_COMM_WORLD, which identifies *all* processes involved in a computation. Other arguments have type integer, datatype handle, or status.

The functions MPI_INIT and MPI_FINALIZE are used to initiate and shut down an MPI computation, respectively. MPI_INIT must be called before any other MPI function and must be called exactly once per process. No further MPI functions can be called after MPI_FINALIZE.

The functions MPI_COMM_SIZE and MPI_COMM_RANK determine the number of processes in the current computation and the integer identifier assigned to the current process, respectively. (The processes in a process group are identified with unique, contiguous integers numbered from 0.) For example, consider the following program, expressed in pseudocode rather than Fortran or C.

```
program main
begin
  MPI_INIT()                            Initiate computation
  MPI_COMM_SIZE(MPI_COMM_WORLD, count)  Find # of processes
  MPI_COMM_RANK(MPI_COMM_WORLD, myid)   Find my id
  print("I am", myid, "of", count)      Print message
  MPI_FINALIZE()                        Shut down
end
```

The MPI standard does not specify how a parallel computation is started. However, a typical mechanism might be a command line argument indicating the number of processes that are to be created: for example, "`myprog -n 4`", where `myprog` is the name of the executable. Additional arguments might be required, for example to specify processor names in a networked environment.

Once a computation is initiated, each of the processes created will normally execute the same program. Hence, execution of the program above gives something like the following output.

```
I am 1 of 4
I am 3 of 4
I am 0 of 4
I am 2 of 4
```

The order in which the output from the four processes appears is not defined; we assume here that the output from individual print statements is not interleaved.

Finally, we consider the functions MPI_SEND and MPI_RECV. These are used to send and receive messages, respectively. A call to MPI_SEND has the following general form.

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

It specifies that a message containing `count` elements of the specified `datatype` starting at address `buf` is to be sent to the process with identifier `dest`. As will be explained in greater detail subsequently, this message is associated with an *envelope* comprising the specified `tag`, the source process's identifier, and the specified communicator (`comm`). An MPI datatype is defined for each C datatype: `MPI_CHAR`, `MPI_INT`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`, and so forth. MPI provides similar datatype names for Fortran and C++. In addition, MPI allows the user to define new datatypes that represent noncontiguous buffers (such as constant stride vectors or index scatter/gathers).

A call to `MPI_RECV` has the following general form.

```
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

It attempts to receive a message with an envelope corresponding to the specified `tag`, `source`, and `comm`, blocking until such a message is available. When the message arrives, elements of the specified `datatype` are placed into the buffer at address `buf`. This buffer is guaranteed by the user to be large enough to contain at least `count` elements. The `status` variable can be used subsequently to inquire about the size, tag, and source of the message.

The six functions just described can be used to express a wide variety of parallel computations. Figure 10.2 shows a simple Fortran program that sends data from the process with rank zero to the process with rank one. Note that in the message-passing model, each process is separate from all other processes. The variables in this program represent *different* memory locations for each process and may, in fact, be on computer hardware (executing on processors) located miles apart. In other words, the variable `buf` in process zero is a different memory location than the variable `buf` in process one.

Nonblocking Communication Operations

The MPI communication routines just described are *blocking*; when the routine returns, the user can use the data buffer. In the case of `MPI_Send`, the user can immediately change the value of data in the buffer (for example, execute `abuf = 2`). In the case of `MPI_Recv`, the user can immediately use the value in the buffer, as we do in our example. This is a simple and easy to use model, but it has two major drawbacks. Consider the program where the two processes exchange data:

```
...
if (rank .eq. 0) then
    call mpi_send( abuf, n, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr )
    call mpi_recv( buf, n, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, &
                   status, ierr )
else if (rank .eq. 1) then
    call mpi_send( abuf, n, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr )
    call mpi_recv( buf, n, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, &
                   status, ierr )
endif
```

```
program main
use mpi
integer ierr, size, rank
integer abuf, buf, status(mpi_status_size)


call mpi_init(ierr)
call mpi_comm_size( mpi_comm_world, size, ierr )
call mpi_comm_rank( mpi_comm_world, rank, ierr )
if (size .lt. 2) then
   print *, 'Error - must have at least 2 processes'
   call mpi_abort( mpi_comm_world, ierr )
endif
if (rank .eq. 0) then
   abuf = 10
   call mpi_send( abuf, 1, MPI_INTEGER, 1, 0, mpi_comm_world, &
                  ierr )
elseif (rank .eq. 1) then
   buf = -1
   print *, 'Buf before recv = ', buf
   call mpi_recv( buf, 1, MPI_INTEGER, 0, 0, mpi_comm_world, &
                  status, ierr )
   print *, 'Buf after recv = ', buf
endif
call mpi_finalize(ierr)
end
```

Figure 10.2: Simple MPI: Using the six basic functions to write a program that communicates data from process 0 to process 1.

For this program to execute correctly, at least one of the processes must complete the `MPI_Send` call so that the matching `MPI_Recv` will be executed. For the `MPI_Send` calls to complete, the data in `abuf` must be copied into a system buffer. This introduces two problems. First, a data copy hurts performance; it is an operation that doesn't achieve anything other than allowing the operation to complete. Second, and more serious, this requires that there be enough buffer space to copy the data into. In the example above, this is (usually) not a problem if `n` is 1, 10, or even 1000; but if `n` is very large (e.g., 100,000,000), there may not be enough space in the buffer. In this case, the program will never complete; it will *deadlock*.

It is always possible to reorder the send and receive operations to avoid this deadlock, but particularly for complex communication patterns, this can be so difficult as to be impractical. Hence, MPI also provides for nonblocking variants of the send and receive routines: `MPI_Isend` and `MPI_Irecv`. These calls do not wait for the communication to complete before they return. (The user can ensure that they have completed by calling `MPI_Wait` or any of several variants.) These routines are necessary for the *correct* operation of message-passing programs. For more discussion on this point, see [409, Chapter 4].

Communicators

In the MPI interface, the message tag is used to match a send operation with a receive operation. The tag provides a simple means of separating messages at the receiver. This can be used to identify the purpose of the message or make sure one part of a program doesn't receive a message that was not intended for it. For example, a different tag value can be used for data values and message-containing configuration options for the program.

In early message-passing libraries, the message tag was the only mechanism provided to distinguish messages. This made it difficult to write reusable program libraries. Even preallocating ranges of tags to a library is not sufficient to ensure that messages intended for one piece of code were not intercepted by another (if `MPI_ANY_TAG` is used). MPI addressed this problem by introducing the concept of a communication context. Each MPI communicator contains a separate communication context; this defines a separate virtual communication space. As discussed above, each MPI send and receive operation takes a communicator as an argument, and thus we can isolate a application library by providing it its own communicator when it is invoked.

Communicators do more than scope a tag namespace. They also define a namespace of processes, i.e., the size and rank used for named endpoints of send and receive operations. Starting with `MPI_COMM_WORLD`, which contains all the processes in a parallel computation, new communicators are formed by either including or excluding processes from an existing communicator. Within a communicator, processes are assigned rank from zero to one minus the size of the communicator. Thus `MPI_Comm_size` returns the number of processes within the specified communicator, while `MPI_Comm_rank` returns the identifier of the current process within the scope of the specified communicator.

Collective Operations

Parallel algorithms often call for coordinated communication operations involving multiple processes. For example, all processes may need to cooperate to transpose a distributed matrix or to sum a set of numbers distributed one per process. Clearly, these global operations can be implemented by a programmer using the send and receive functions introduced previously. For convenience, and to permit optimized implementations, MPI also provides a suite of specialized *collective communication* functions that perform commonly used operations of this type. These functions include the following.

- Barrier: Synchronize all processes.
- Broadcast: Send data from one process to all processes.

- Gather: Gather data from all processes to one process.
- Scatter: Scatter data from one process to all processes.
- Reduction operations: addition, multiplication, etc. of distributed data.

These operations are all executed in a collective fashion, meaning that each process in a process group calls the communication routine.

10.1.2 Parallel Virtual Machine

At this point, it would be instructive to discuss the major message-passing library that was in common use prior to the development of MPI. The Parallel Virtual Machine, or PVM [374], was a widely used library that ran on a large number of different parallel computing platforms, including networks of workstations. While there was a public domain distribution of this library, there were also vendor-supported versions on computers such as the Cray T3E and the IBM SP2. Figure 9.2 in Chapter 9 illustrates the use of PVM in a simple problem.

The message-passing primitives of PVM differ from MPI in a number of significant ways. For example, there is no notion of communicators. Also, buffers are implicit to the the send operation, rather then being explicitly provided by a function argument. However, PVM is still important. In addition to providing communication operations, it also provided a set of operations for manipulating the computing environment to dynamically construct a "parallel virtual machine," from which the library takes its name.

Unlike MPI, which assumes a fixed set of processes in `MPI_COMM_WORLD`, PVM applications can dynamically change the set of processes over which communication operations take place. Rather than sending to a process of a specific rank, PVM applications use *task identifiers (tid)* to specify the endpoints of a communication operation. While many applications can fit into the static process model of MPI, there are a number of significant applications that cannot. Addressing the needs of these more dynamic applications was one of the motivations that lead to the development of MPI-2, discussed in the next section.

10.1.3 Extensions to the Message Passing Interface

Even though the MPI specification contains a large number of routines, many users found that it lacked needed features. To remedy this, the MPI Forum developed a set of extensions to the MPI specification, called MPI-2 [408, 672]. These new features fell into three major categories. In the original MPI specification (henceforth MPI-1), the number of processes in an MPI application remains fixed between the call to `MPI_Init` and `MPI_Finalize`. As experience with PVM showed, it is sometimes helpful to be able to change the number of processes available during the execution of a parallel computation. In MPI-2, the function `MPI_Comm_spawn` allows the user to create new processes and connect them to an MPI application; other routines allow two MPI programs to connect to each other.

Another requested feature, stimulated both by research into other parallel computing models such as active messages [969] and bulk synchronous processes (BSP) [474] and in particular by the success of the Cray `shmem` library, was one-sided or remote-memory operations. MPI-2 provides routines that allow a process to put or get directly into another process's memory.

Perhaps the most requested feature, however, was for parallel I/O. The MPI approach to I/O is covered later in this book in Chapter 11.

In addition, MPI-2 added bindings for C++ and Fortran 90; defined how MPI programs that make use of C, Fortran 90, and C++ communicate; and added a number of more minor (but necessary) extensions. These new features are covered in more detail in [408, 410].

10.1.4 Summary

The Message Passing Interface has been a tremendously successful parallel programming system. While low level, it has provided a standard notation for parallel program design and has permitted the development of high-performance implementations. MPI-1 implementations are available on almost every parallel computer, and large numbers of applications and libraries have been developed that achieve high performance on small and large parallel systems. MPI-2 implementations are also beginning to appear. The I/O part of MPI-2 is already widely available, and several supercomputer vendors have complete MPI-2 implementations.

Equally important, MPI support for modular program construction has facilitated, almost for the first time, the development of reusable parallel program components. (The Connection Machine's library was one important precursor.) Hence we see the development of important reusable code bases. An example is the PETSc library for scientific computing [71], which allows application programmers to develop complex parallel programs without writing any explicitly parallel code.

In summary, MPI is particularly suited for applications where portability, both in space (across different systems that exist now) and in time (across generations of computers), is important. MPI is also an excellent choice for task-parallel computations and for applications where the data structures are dynamic, such as unstructured mesh computations.

10.2 Multithreaded Programming

We now turn our attention to the multithreaded programming model. As we described above, the message-passing model assumes that each concurrently executing program component has a separate, independent address space and that data is moved explicitly between these address spaces via send and receive operations. The shared-memory programming model is the exact opposite, in that we start with the assumption that concurrently executing program components all share a single, common address space. There is no need to provide special operations for copying, as program components can exchange information simply by reading and writing to memory using normal variable assignment operations. Because the concurrent elements of the program do not have distinct address spaces, it is not correct to refer to them as processes. Instead, we say that the program consists of many independent "threads of control," or threads for short. Hence the use of the name multithreaded programming.

Because communication in shared-memory programs is implicit, it is generally up to the hardware platform on which the program is executing to ensure that the latest value of a variable updated in one thread is used when that same variable is accessed in another thread. This is the so-called memory coherence problem, which can be difficult to solve for a variety of reasons. Modern computer architectures tend to copy data in local, high-speed memory in order to decrease access time. Some of this memory (i.e., registers) is manipulated explicitly by the compiler, while other memory takes the form of one or more levels of cache. These are transparent to the compiler and application (except in terms of performance). If shared-memory programs are to function properly, both the hardware and the compilers used to generate and execute the multithreaded program need to make sure that the various pieces of memory in the computer remain consistent. Hardware support typically takes the form of a *cache-coherency protocol*. Many different consistency models and associated coherency protocols have been proposed, often with slightly different sharing semantics [7, 555]. Yet in spite of these sophisticated protocols, physical constraints limit the number of processors that can share memory. As a rule, shared-memory programs do not scale as well as message-passing programs.

Software-only shared-memory systems have been proposed as well. These so-called *distributed shared-memory* (DSM) systems generally try to exploit the hardware support for implementing virtual memory in order to determine when changes to memory are made and to propagate those changes to other processors [89, 103, 555, 710]. These systems typically use a message-passing layer as the means for communicating updated values throughout the system. Because memory updates are done on a page basis with hundreds or thousands of memory locations, arbitrary writing of variables can be very costly. For this reason, DSM systems typically try to enhance the locality of modifications, restrict which memory locations can be shared, or introduce specialized coherency semantics in order to reduce the number of pages that must be sent from one processor to another.

While the operations for sharing data are straightforward, multithreaded programs introduce the new problem of controlling access to a shared-memory location while values are being updated. In the message-passing model, the success of a receive operation indicates that all the data had been transferred from one process to another and that the data is now available for use. Without the intervention of some explicit synchronization operation, interleaved execution of multiple threads can result in incorrect results. For example, in many situations, partially modified data structures should not be used, and a reading thread should be delayed until the writing thread has finished updating all of the data-structure fields. Likewise, having two threads attempt to update the contents of a data structure at the same time can result in disaster, with the result being an arbitrary combination of the fields written by the competing threads.

Over the years, many different proposals have been made concerning synchronization of multithreaded programs. Some of these limit the ways in which variables can be written (e.g., functional and single-assignment languages [4, 352, 660]). Others define special synchronization operations, including monitors, locks, critical sections, condition variables, semaphores, and barriers (see [41] for an overview of these operations). There are some fundamental concepts (for example, some notion of an atomic test and set operation) that underlie all of these primitives. But in many cases, one set of synchronization operations can be implemented in terms of other primitives and the choice of the set to use comes down to programming convenience and performance.

Methods for development of multithreaded programs are similar to methods for development of message-passing systems: one can produce special purpose languages, extend existing languages or provide purely library-based approaches. In the following, we will examine two different approaches. The first is POSIX threads and is completely library based. The second, OpenMP, combines compiler support with library calls to implement its multithreading model.

10.2.1 POSIX Threads

The multithreading programming model has important uses outside of parallel programming. Multithreading proves to be a very effective programming model when a program has to respond to asynchronous requests. Such requests occur, for example, when interacting with slow I/O systems or when implementing network based systems such as client/server architectures (specifically the server). For this reason, the set of real-time extensions defined for the POSIX operating system interface includes a thread library. Because of its wide range of intended uses, the POSIX standard thread library [503], or *pthreads* as it is often called, includes many features that are not of interest to parallel program developers. In the following, we provide a brief overview of those facilities that are germane to parallel programs.

A Pthread program starts its life with a single thread of control, just like any sequential program. New threads of control must be created explicitly by calling the `pthread_create` function and specifying the function to run in the newly created thread. This call creates a thread record, which is initialized by allocating a call stack and creating an initial stack frame, and sets up the call of the function to be executed in the thread. There is no restriction on the number of threads that can be created by a program. In particular, the number of threads can exceed the number of processors available in the system. To support the creation of more threads than processors, thread records are generally handed to a scheduler, which arranges to execute the thread when a processor becomes available. It is interesting to note that from the perspective of concurrency, the pthread model is fundamentally dynamic, while the MPI model is basically static.

A thread terminates when the function being executed by the thread completes or when an explicit thread exit function is called. Threads can also be explicitly terminated by specifying the the thread identifier returned by a `pthread_create` call as an argument to the `pthread_kill` function.

In the POSIX model, all the memory associated with a thread is shared, including the thread's call stack, dynamically allocated heap memory, and obviously global variables. This can cause programming difficulties. Often, one needs a variable that is global to the routines called within a thread but not shared between between threads. A set of pthreads functions are used to manipulate *thread local storage* to address these requirements.

Synchronization Operations

The pthreads library provides two types of synchronization primitives, one associated with the control structure (i.e., threads) and the second to synchronize access to data structures. The control-oriented synchronization enables one thread to block, waiting for the completion of a second thread. Unfortunately, this facility does not generalize to more then two threads, meaning that it cannot be used to implement the popular join/fork concurrency model (discussed below).

The data-oriented synchronization routines are based on the use of a *mutex*, short for mutual exclusion. A mutex is a dynamically allocated data structure that can be passed as an argument to the routines `pthread_mutex_lock` and `pthread_mutex_unlock`. Once a `pthread_mutex_lock` call is made on a specific mutex, subsequent `pthread_mutex_lock` calls will block until a call is made to `pthread_mutex_unlock` with that mutex.

Locking is generally used for fine-grain access control. To minimize the response time to an unlock operation, locks are often implemented by "spinning," repeatedly testing the lock to see if it has been released. This has the downside of consuming 100% of the resources of the processor executing the lock operation. More coarse-grain synchronization is provided by *condition variables*, which allow a thread to wait until a Boolean predicate that depends on the contents of one or more shared-memory locations becomes true.

A condition variable associates a mutex with the desired predicate. Before the program makes its test, it obtains a lock on the associated mutex. Then it evaluates the predicate. If the predicate evaluates to false, the thread can execute a `pthread_cond_wait` operation, which atomically suspends the calling thread, puts the thread record on a waiting list that is part of the condition variable, and releases the mutex. The thread scheduler is now free to use the processor to execute another thread. If the predicate evaluates to true, the thread simply releases its lock and continues on its way.

If a thread changes the value of any shared variables associated with a condition variable predicate, it needs to cause any threads that may be waiting on this condition variable to be rescheduled. The `pthread_cond_signal` causes one of the threads waiting on the condition variable to become unblocked, returning from the `pthread_cond_wait` that caused it to block in the first place. The mutex is automatically reobtained as part of the return from the wait, so the thread is in the position to reevaluate the predicate immediately.

Because condition variables enable atomic evaluation of arbitrary predicates, they are quite flexible and can be used to implement a range of different synchronization structures, including semaphores, barriers, critical sections, etc.

10.2.2 OpenMP

As we discussed above, support for parallel programming was only one of the factors that was considered in the design of pthreads. Not surprisingly, compromises were made that affect both programmability and performance. The OpenMP interface [725] is an alternative multithreading interface specifically designed to support high-performance parallel programs.

OpenMP differs from pthreads in several significant ways. Where pthreads is implemented purely as a library, OpenMP is implemented as a combination of library calls and a set of compiler directives or pragmas. These directives instruct the compiler to create threads, perform synchronization operations and manage shared memory. OpenMP does require specialized compiler support in order to understand and process these directives. However, an increasing number of vendors are producing OpenMP versions of their Fortran, C, and C++ compilers.

In pthreads, there is almost no structure or a priori relationship between threads, short of the ability of one thread to wait for the termination of one other thread. While such lack of structure is essential to the design of servers, which typically consist of a number of independent threads, the need for structure in the design of programs is well understood. Because OpenMP was designed specifically for parallel applications, the use of threads is highly structured, following what is known as the fork/join model. This is a block-structured approach to introducing concurrency. A single thread of control splits into some number of independent threads (the fork), and the end of the block is reached when all the threads have completed execution of their specified tasks (the join). In OpenMP, a fork/join block is indicated by a *parallel region*, indicated by PARALLEL and END PARALLEL directives. The number of threads assigned to a region is defined by the user, either globally or on a region-by-region basis, and each thread executes the code enclosed by the PARALLEL directives. Because of the way OpenMP is defined, a PARALLEL region does not imply the creation of a new thread. Thus if an existing thread is available, it can be used to execute a parallel region, allowing the OpenMP implementation to amortize thread start-up costs. Parallel blocks can be nested, with the implementation figuring out when the threads in each nested region have completed. Clearly, the use of parallel regions can significantly decrease the potential for error.

The PARALLEL region enables a single task to be replicated across a set of threads. However, in parallel programs it is very common to distribute different tasks across a set of threads, for example parallel iteration over the index set of a loop. To support this common need, a PARALLEL region can be augmented with an additional set of directives that enable each thread to execute a different task. This is called *worksharing.* The most useful form of worksharing is a parallel `do` loop, which is specified by inserting a DO directive in front of a do loop and enclosing the entire thing in a PARALLEL declaration. During execution, the parallel region creates some number of independent threads, and the worksharing declaration causes the compiler to generate code that distributes the iterations (which can be greater in number then the number of threads) to the threads for execution.

OpenMP synchronization primitives are also more application-oriented than the pthread synchronization primitives. OpenMP synchronization primitives are:

- *critical sections*, which ensure that only one thread at a time executes the enclosed code (critical sections are similar to pthread mutexes);
- *atomic updates*, which behave like critical sections, but can be optimized on some hardware platforms;
- *barriers*, which synchronize all threads in a parallel region; and
- *master selection*, which ensures that the enclosed code only executes in one thread, even if the code is part of a parallel region.

This selection of synchronization primitives makes it easier to write parallel programs. Certainly, each of these operations can be implemented in terms of pthread mutex and condition variables. However, by including these as basic OpenMP operations, it is possible for an OpenMP implementation to generate code that is more efficient than the equivalent pthread code.

The final place where OpenMP differs from pthreads is in its treatment of shared memory. Recall that in pthreads, all memory is shared with the exception of thread local storage. Unlike pthreads, OpenMP does not allow sharing of stack variables. Again, this makes it possible to generate better optimized code. OpenMP also provides thread local storage. However, because of the block-structured nature of PARALLEL regions, OpenMP threads can access local variables that are defined by the routine or block in which the PARALLEL region occurs (this is not an issue in pthreads because pthreads has no concept of nesting). Sometimes it is advantageous for each thread to have its own private copy of these variables, and OpenMP allows variables to be annotated as being PRIVATE.

In summary, OpenMP does not define a new programming language, but rather consists of a set of annotations that are interpreted by an OpenMP-enabled compiler, or preprocessor. OpenMP annotations can be included in Fortran programs, as directives, or in C and C++ programs as pragmas. By focusing specifically on the needs of parallel programs, OpenMP can result in a more convenient and higher-performance implementation of multithreaded parallel programs.

10.3 Combining Message Passing and Multithreading

In the preceding sections, we have discussed the message-passing and multi-threaded approaches to parallel programming in isolation. Given that many recent large-scale parallel computers are built as clusters of multiprocessor (i.e., shared-memory) nodes, there is a strong motivation to combine message passing and multithreading in a single application. The hybrid approach has the potential to optimize program performance by using multithreaded structures within a multiprocessor node and message-passing primitives for communication between nodes.

In principle, there is no reason why message passing and parallel programming cannot be combined in a single application program. In practice, we find that many implementations of message passing are not threadsafe; that is, it is not safe for two threads to make message-passing calls at the same time. This can be caused by a number of different factors: the message-passing interface may have been designed so as to preclude a threadsafe implementation (for example, requiring that state be stored in the implementation), or an otherwise thread-safe API may not have been implemented in a threadsafe manner, or the low-level interfaces to the message-passing hardware may not be threadsafe. These caveats aside, the MPI interface was designed to be threadsafe and threadsafe implementations of MPI exist. These implementations can be combined with multithreading approaches such as pthreads or OpenMP to produce hybrid programs.

There are a variety of approaches that can be used to merge the message-passing and multithreading styles. The most common is to use a single thread (the main thread) for all MPI communication and to use other threads for computational tasks; this is the case when OpenMP is used to parallelize loops that do not contain any MPI calls. Most MPI implementations (even those that are not threadsafe) may be used in this way with threads. Another method is to perform sends from any thread, but receives within a single receiver thread. This thread would be responsible for integrating receive buffers into shared memory. It would then either notify waiting threads or create new threads to process the new data. Alternatively, one could perform receives from an arbitrary thread. This can be a good approach if there is a collection of worker threads, and any one of them can process the data from an incoming message. A modification is to perform send and receive operations between specific thread pairs [327]. For this to work, one must assume that threads are long lived, as performing a send to a thread that no longer exists would result in an error. MPI communicators can be very helpful in implementing this type of hybrid model. A final approach is that taken by special purpose communication libraries, such as Nexus [348], in which the arrival of data causes the automatic creation of a new thread of control to process that data.

While tools and techniques that support both message passing and multithreading are important, the significant challenge lies not in the tools but in the application itself. For these hybrid techniques to be useful, one must be able to exploit the heterogeneous characteristics, in terms of bandwidth and latency, found in shared-memory and message-passing systems [325, 583, 614, 909].