

Message-Passing Parallel Programming with MPI

What is MPI?

MPI (Message-Passing Interface) is a message-passing library specification that can be used to write parallel programs for parallel computers, clusters, and heterogeneous networks.

Portability across platforms is a crucial advantage of MPI. Not only can MPI programs run on any distributed-memory machine and multicomputer, but they can also run efficiently on shared-memory machines. OpenMP programs can only run efficiently on machines with hardware support for shared memory.

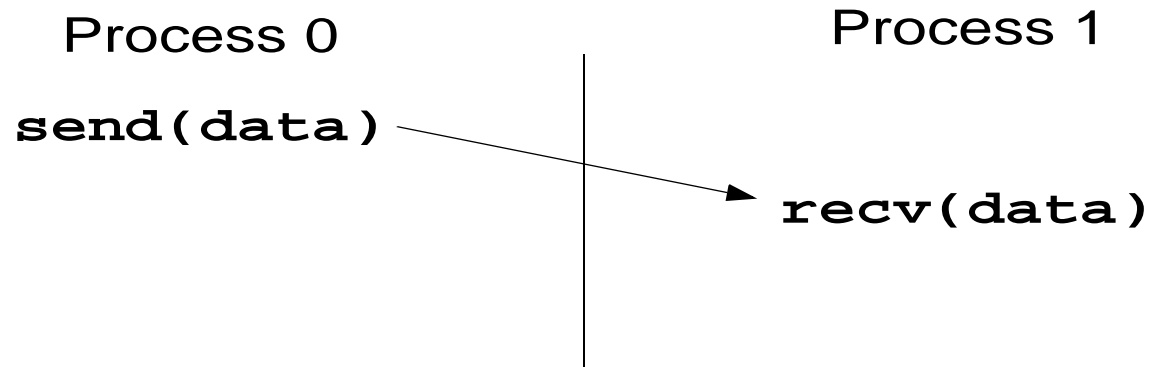
Like OpenMP, MPI was designed with the participation of several computer vendors (IBM, Intel, Cray, Convex, Meiko, Ncube) and software houses (KAI, ParaSoft). Furthermore, several Universities participated in the design of MPI.

Cooperative operations

Message-passing is an approach that makes the exchange of data cooperative.

Data must both be explicitly sent and received.

An advantage is that any change in the receiver's memory is made with the receiver's participation.

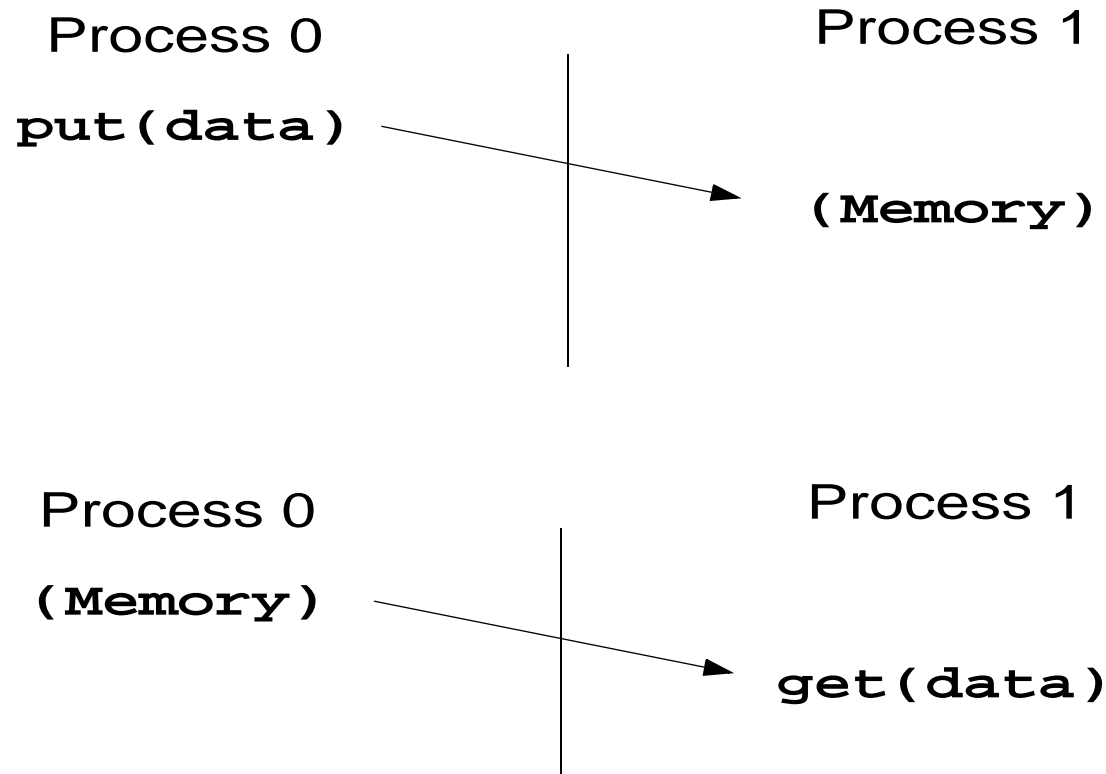


(From W. Gropp's transparencies: Tutorial on MPI. <http://www.mcs.anl.gov/mpi>).

One-sided operations

One-sided operations between parallel processes include remote memory reads and writes.

An advantage is that data can be accessed without waiting for another process.



(From W. Gropp's transparencies: Tutorial on MPI. <http://www.mcs.anl.gov/mpi>).

Comparison

One-sided operations tend to produce programs that are easy to read. With one sided operations only the processor using the data has to participate in the communication.

- Example: The following code would execute the statement $a=f(b,c)$ in processor 1 of a multicomputer if a and c are in the memory of processor 2.

```
in processor 1:      receive (2,x)
                    y = f(b,x)
                    send (2,y)
```

```
in processor 2:      send (1,c)
                    receive(1,a)
```

In a shared-memory machine or a machine with a global address space, the code would be simpler because the program running on processor 2 does not need to participate in the computation:

```
in processor 1:

                    x := get(2,c)
                    y = f(b,x)
                    put(2,a) := y
```

- Example: To execute the loop

```
do i=1,n
    a(i) = x(k(i))
end do
```

in parallel on a message passing machine could require complex interactions between processors.

- Pointer chasing across the whole machine is another case where the difficulties of message-passing become apparent.

Features of MPI

MPI has a number of useful features beyond the send and receive capabilities.

- *Communicators.* A subset of the active processes that can be treated as a group for collective operations such as broadcast, reduction, barriers, sending or receiving. Within each communicator, a process has a *rank* that ranges from zero to the size of the group minus one. There is a default communicator that refers to all the MPI processes that is called `MPI_COMM_WORLD`.
- *Topologies.* A communicator can have a topology associated with it. This arranges a communicator into some layout. The most common layout is a cartesian decomposition.

- *Communication modes.* MPI supports multiple styles of communication, including blocking and non-blocking. Users can also choose to use explicit buffers for sending or allow MPI to manage the buffers. The nonblocking capabilities allow the overlap of communication and computation.
- *Single-call collective operations.* Some of the calls in MPI automate collective operations in a single call. For example, there is a single call to sum values across all the processes to a single value.

(From K. Dowd and C. Severance. High Performance Computing. O'Reilly 1998).

Writing MPI Programs

```
include 'mpif.h'

integer rank, size

call MPI_INIT(ierr)

call MPI_COMM_RANK(MPI_COMM_WORLD,rank)

call MPI_COMM_SIZE(MPI_COMM_WORLD,size)

print *, "hello world I'm ", rank, " of ", size

call MPI_FINALIZE(ierr)

end
```

(From W. Gropp's transparencies: Tutorial on MPI. <http://www.mcs.anl.gov/mpi>).

Typical output on a SMP

```
hello world I'm 1 of 20
hello world I'm 2 of 20
hello world I'm 3 of 20
hello world I'm 4 of 20
hello world I'm 5 of 20
hello world I'm 6 of 20
hello world I'm 7 of 20
hello world I'm 9 of 20
hello world I'm 11 of 20
hello world I'm 12 of 20
hello world I'm 10 of 20
hello world I'm 8 of 20
hello world I'm 14 of 20
hello world I'm 13 of 20
hello world I'm 15 of 20
hello world I'm 16 of 20
hello world I'm 19 of 20
hello world I'm 0 of 20
hello world I'm 18 of 20
hello world I'm 17 of 20
4.03u 0.43s 0:01.88e 237.2%
```

Commentary

- `include 'mpif.h'` provides basic MPI definitions and types
- `call MPI_INIT` starts MPI
- `call MPI_FINALIZE` exits MPI
- `call MPI_COMM_RANK(MPI_COMM_WORLD,rank)` returns the rank of the process making the subroutine call. Notice that this rank is within the default communicator.
- `call MPI_COMM_SIZE(MPI_COMM_WORLD,size)` returns the total number of processes involved in the execution of the MPI program.

(From W. Gropp's transparencies: Tutorial on MPI. <http://www.mcs.anl.gov/mpi>).

Send and Receive Operations in MPI

The basic (blocking) send operation in MPI is

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

where

- **(buf, count, datatype)** describes **count** occurrences of items of the form **datatype** starting at **buf**.
- **dest** is the rank of the destination in the group associated with the communicator **comm**.
- **tag** is an integer to restrict receipt of the message.

(From W. Gropp E. Lusk, and A. Skejellum. Using MPI. MIT Press 1996).

The receive operation has the form

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

where

- **count** is the size of the receive buffer
- **source** is the id of source process, or **MPI_ANY_SOURCE**
- **tag** is a message tag, or **MPI_ANY_TAG**
- **status** contains the source, tag, and count of the message actually received.

Broadcast and Reduction

The routine `MPI_BCAST` sends data from one process to all others.

The routine `MPI_REDUCE` combines data from all processes (by adding them in the example shown next), and returning the result to a single program.

Second MPI Example: PI

```
program main
include 'mpif.h'
double precision  PI25DT
parameter          (PI25DT = 3.141592653589793238462643d0)

double precision  mypi, pi, h, sum, x, f, a
integer n, myid, numprocs, i, rc
c                                     function to integrate
f(a) = 4.d0 / (1.d0 + a*a)

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
print *, "Process ", myid, " of ", numprocs, " is alive"

sizetype   = 1
sumtype    = 2

10  if ( myid .eq. 0 ) then
      write(6,98)
98   format('Enter the number of intervals: (0 quits)')
      read(5,99) n
99   format(i10)
endif
```



```

    call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c                                     check for quit signal
    if ( n .le. 0 ) goto 30
c                                     calculate the interval size
    h = 1.0d0/n

    sum = 0.0d0
    do 20 i = myid+1, n, numprocs
        x = h * (dble(i) - 0.5d0)
        sum = sum + f(x)
20    continue
    mypi = h * sum

c                                     collect all the partial sums
    call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
$      MPI_COMM_WORLD,ierr)

c                                     node 0 prints the answer.
    if (myid .eq. 0) then
        write(6, 97) pi, abs(pi - PI25DT)
97    format(' pi is approximately: ', F18.16,
+         ' Error is: ', F18.16)
    endif
    goto 10
30    call MPI_FINALIZE(rc)
    stop
end

```

(From W. Gropp's transparencies: Tutorial on MPI. <http://www.mcs.anl.gov/mpi>).