MPI_WTIME()

```
double MPI_Wtime(void)

DOUBLE PRECISION MPI_WTIME()
```

{`double MPI::Wtime()` *(binding deprecated, see Section 15.2)* }

MPI_WTIME returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past.

The "time in the past" is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage. One would use it like this:

```
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n",endtime-starttime);
}
```

The times returned are local to the node that called them. There is no requirement that different nodes return "the same time." (But see also the discussion of MPI_WTIME_IS_GLOBAL).

MPI_WTICK()

```
double MPI_Wtick(void)

DOUBLE PRECISION MPI_WTICK()
```

{`double MPI::Wtick()` *(binding deprecated, see Section 15.2)* }

MPI_WTICK returns the resolution of MPI_WTIME in seconds. That is, it returns, as a double precision value, the number of seconds between successive clock ticks. For example, if the clock is implemented by the hardware as a counter that is incremented every millisecond, the value returned by MPI_WTICK should be 10^{-3}.

8.7 Startup

One goal of MPI is to achieve *source code portability*. By this we mean that a program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This explicitly does *not* say anything about how an MPI program is started or launched from the command line, nor what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed

before other MPI routines may be called. To provide for this, MPI includes an initialization routine MPI_INIT.

MPI_INIT()

```
int MPI_Init(int *argc, char ***argv)

MPI_INIT(IERROR)
    INTEGER IERROR

{void MPI::Init(int& argc, char**& argv) (binding deprecated, see Section 15.2) }

{void MPI::Init() (binding deprecated, see Section 15.2) }
```

ticket313.

[All MPI programs must contain exactly one call to an MPI initialization routine: MPI_INIT or MPI_INIT_THREAD. Subsequent calls to any initialization routines are erroneous. The only MPI functions that may be invoked before the MPI initialization routines are called are MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED.] ticket313. Each MPI process must contain exactly one call to an MPI initialization routine, MPI_INIT or MPI_INIT_THREAD. Subsequent calls by the process to any initialization routine are erroneous. The only MPI functions that may be invoked by a process before the MPI initialization routine completed are MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED.

The version for ISO C accepts the argc and argv that are provided by the arguments to main or NULL:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    /* parse arguments */
    /* main program    */

    MPI_Finalize();     /* see below */
}
```

The Fortran version takes only IERROR.

Conforming implementations of MPI are required to allow applications to pass NULL for both the argc e argv arguments of main in C. [and C++. In C++, there is an alternative binding for MPI::Init that does not have these arguments at all.] ticket313. ticket313.

> *Rationale.* In some applications, libraries may be making the call to MPI_Init, and may not have access to argc and argv from main. It is anticipated that applications requiring special information about the environment or information supplied by mpiexec can get that information from environment variables. (*End of rationale.*)

MPI_FINALIZE()

```
int MPI_Finalize(void)
```

```
MPI_FINALIZE(IERROR)
    INTEGER IERROR
```

{`void MPI::Finalize()` *(binding deprecated, see Section 15.2)* }

ticket313. This routine cleans up all MPI state. [Each process must call MPI_FINALIZE before it exits.] Before each process exits, the process must call MPI_FINALIZE. Unless there has been a call to MPI_ABORT, each process must ensure that all pending nonblocking communications are (locally) complete before calling ticket313. MPI_FINALIZE. [Further, at the instant at which the last process calls MPI_FINALIZE, all pending sends must be matched by a receive, and all pending receives must be matched by a send.

For example, the following program is correct] Further when the last process calls MPI_FINALIZE, all non-local MPI calls at each process have been matched by MPI calls at the other processes that are needed to complete the relevant operation: For each send, the matching receive has occurred, each collective operation has been invoked at all involved processes, etc. The following examples illustrates these rules

Example 8.3 The following code is correct

```
Process 0               Process 1
---------               ---------
MPI_Init();             MPI_Init();
MPI_Send(dest=1);       MPI_Recv(src=0);
MPI_Finalize();         MPI_Finalize();
```

Example 8.4 Without a matching receive, the program is erroneous

```
Process 0               Process 1
-----------             -----------
MPI_Init();             MPI_Init();
MPI_Send (dest=1);
MPI_Finalize();         MPI_Finalize();
```

ticket313. [deleted in April Since MPI_FINALIZE is a collective call, a correct MPI program will naturally ensure that all participants in pending collective operations have made the call before calling MPI_FINALIZE.

A successful return from a blocking communication operation or from MPI_WAIT or MPI_TEST tells the user that the buffer can be reused and means that the communication is completed by the user, but does not guarantee that the local process has no more work to do. A successful return from MPI_REQUEST_FREE with a request handle generated by an MPI_ISEND nullifies the handle but provides no assurance of operation completion. The MPI_ISEND is complete only when it is known by some means that a matching receive has completed. MPI_FINALIZE guarantees that all local actions required by communications the user has completed will, in fact, occur before it returns.

MPI_FINALIZE guarantees nothing about pending communications that have not been completed (completion is assured only by MPI_WAIT, MPI_TEST, or MPI_REQUEST_FREE ticket313. combined with some other verification of completion).]

[

Example 8.5 This program is correct HEADER SKIP ENDHEADER

```
rank 0                              rank 1
=====================================================
...                                 ...
MPI_Isend();                        MPI_Recv();
MPI_Request_free();                 MPI_Barrier();
MPI_Barrier();                      MPI_Finalize();
MPI_Finalize();                     exit();
exit();
```

Example 8.6 This program is erroneous and its behavior is undefined: HEADER SKIP ENDHEADER

```
rank 0                              rank 1
=====================================================
...                                 ...
MPI_Isend();                        MPI_Recv();
MPI_Request_free();                 MPI_Finalize();
MPI_Finalize();                     exit();
exit();
```

]

ticket313.

Example 8.7 This program is erroneous: The MPI_Isend call is not guaranteed to be locally complete before process 0 calls MPI_Finalize

```
Process 0                   Process 1
---------                   ---------
MPI_Isend();                MPI_Recv();
MPI_Request_free();         MPI_Barrier();
MPI_Barrier();              MPI_Finalize();
MPI_Finalize();
```

ticket313.

[If no MPI_BUFFER_DETACH occurs between an MPI_BSEND (or other buffered send) and MPI_FINALIZE, the MPI_FINALIZE implicitly supplies the MPI_BUFFER_DETACH.

Example 8.8 This program is correct, and after the MPI_Finalize, it is as if the buffer had been detached. HEADER SKIP ENDHEADER

```
rank 0                              rank 1
=====================================================
...                                 ...
buffer = malloc(1000000);           MPI_Recv();
MPI_Buffer_attach();                MPI_Finalize();
MPI_Bsend();                        exit();
MPI_Finalize();
free(buffer);
exit();
```

] While the user must ensure that communications can complete before MPI is finalized, it needs not free resources allocated by MPI (buffers, windows, requests, communicators, etc.); the MPI_FINALIZE function will do so. ticket313.

Example 8.9 This program is correct, and after the MPI_Finalize, it is as if the buffer had been detached.

```
Process 0                       Process 1
---------                       ---------
buffer = malloc(1000000);       MPI_Recv();
MPI_Buffer_attach();            MPI_Finalize();
MPI_Bsend();                    exit();
MPI_Finalize();
free(buffer);
exit();
```

ticket313. [

Example 8.10 In this example, MPI_Iprobe() must return a FALSE flag. MPI_Test_cancelled() must return a TRUE flag, independent of the relative order of execution of MPI_Cancel() in process 0 and MPI_Finalize() in process 1.

The MPI_Iprobe() call is there to make sure the implementation knows that the "tag1" message exists at the destination, without being able to claim that the user knows about it.

HEADER SKIP ENDHEADER

```
rank 0                          rank 1
========================================================
MPI_Init();                     MPI_Init();
MPI_Isend(tag1);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Iprobe(tag2);
MPI_Barrier();                  MPI_Barrier();
                                MPI_Finalize();
                                exit();
MPI_Cancel();
MPI_Wait();
MPI_Test_cancelled();
MPI_Finalize();
exit();
```

ticket313.]

Example 8.11 This program is correct. The cancel operation must succeed, since the send cannot complete normally.

```
Process 0                    Process 1
---------                    ---------
MPI_Isend(tag1);             MPI_Finalize();
MPI_Cancel();
MPI_Wait();
MPI_Finalize();
```

[

Advice to implementors. An implementation may need to delay the return from MPI_FINALIZE until all potential future message cancellations have been processed. One possible solution is to place a barrier inside MPI_FINALIZE (*End of advice to implementors.*)

]

Advice to implementors. An implementation may need to delay the return from MPI_FINALIZE on a process even if all communications related to MPI calls by that process have completed; the process may still receive cancel requests for messages it has completed receiving. One possible solution is to place a barrier inside MPI_FINALIZE (*End of advice to implementors.*)

Once MPI_FINALIZE returns, no MPI routine (not even MPI_INIT) may be called, except for MPI_GET_VERSION, MPI_INITIALIZED, and MPI_FINALIZED. Each process must complete any pending communication it initiated before it calls MPI_FINALIZE. If the call returns, each process may continue local computations, or exit, without participating in further MPI communication with other processes. MPI_FINALIZE is collective over all connected processes. If no processes were spawned, accepted or connected then this means over MPI_COMM_WORLD; otherwise it is collective over the union of all processes that have been and continue to be connected, as explained in Section 10.5.4 on page 358.

Advice to implementors. Even though a process has completed all the communication it initiated, such communication may not yet be completed from the viewpoint of the underlying MPI system. E.g., a blocking send may have completed, even though the data is still buffered at the sender. The MPI implementation must ensure that a process has completed any involvement in MPI communication before MPI_FINALIZE returns. Thus, if a process exits after the call to MPI_FINALIZE, this will not cause an ongoing communication to fail. (*End of advice to implementors.*)

Although it is not required that all processes return from MPI_FINALIZE, it is required that at least process 0 in MPI_COMM_WORLD return, so that users can know that the MPI portion of the computation is over. In addition, in a POSIX environment, they may desire to supply an exit code for each process that returns from MPI_FINALIZE.

Example 8.12 The following illustrates the use of requiring that at least one process return and that it be known that process 0 is one of the processes that return. One wants code like the following to work no matter how many processes return.