CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **15**:93–100 (DOI: 10.1002/cpe.705)

MPI-CHECK: a tool for checking Fortran 90 MPI programs

Glenn Luecke*,†, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva and Yan Zou

High Performance Computing Group, Iowa State University, Ames, IA 50011, U.S.A.

SUMMARY

MPI is commonly used to write parallel programs for distributed memory parallel computers. MPI-CHECK is a tool developed to aid in the debugging of MPI programs that are written in free or fixed format Fortran 90 and Fortran 77. MPI-CHECK provides automatic compile-time and run-time checking of MPI programs. MPI-CHECK automatically detects the following problems in the use of MPI routines: (i) mismatch in argument type, kind, rank or number; (ii) messages which exceed the bounds of the source/destination array; (iii) negative message lengths; (iv) illegal MPI calls before MPI_INIT or after MPI_FINALIZE; (v) inconsistencies between the declared type of a message and its associated DATATYPE argument; and (vi) actual arguments which violate the INTENT attribute. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: MPI; tool; bounds checking; argument checking; data type checking

1. INTRODUCTION

MPI [1] is commonly used to write parallel programs for distributed memory parallel computers. Writing and debugging MPI programs is often difficult and time consuming. MPI-CHECK [2] is a tool developed to help make this process easier and faster. MPI-CHECK provides compile-time and run-time checking that automatically finds many of the errors made when writing MPI programs. MPI-CHECK runs on both shared memory and distributed memory parallel computers.

While MPI-CHECK was being developed, a similar project was being carried out at Lawrence Livermore National Laboratories where they were developing an MPI tool named Umpire [3]. Umpire is a tool for detecting MPI errors at run-time that monitors the MPI operations of an application by interposing itself between the application and the MPI run-time system using the MPI profiling layer. Umpire then checks the application's MPI behavior for certain errors. Umpire's initial collection

*Correspondence to: Glenn Luecke, High Performance Computing Group, Iowa State University, Ames, IA 50011, U.S.A.
†E-mail: grl@iastate.edu

Received 4 December 2001
Revised 18 April 2002

of programming errors detected includes deadlock detection, mismatched collective operations and resource exhaustion. Unlike MPI-CHECK, Umpire has a central manager that controls the execution of the MPI program and collects the MPI call information. Currently, Umpire only runs on shared memory machines and not on distributed memory parallel computers. Umpire may be used with Fortran, C and C++ MPI programs.

Compile-time checking of MPI-CHECK is discussed in Section 2. Section 3 discusses the run-time checking done by MPI-CHECK and Section 4 contains our conclusions and future directions.

MPI-CHECK may be obtained from [2].

2. COMPILE-TIME CHECKING

When writing a C/C++ program with calls to MPI routines, one is required to add the statement #include <mpi.h> which supplies needed MPI constants and the prototypes for each MPI function. When writing a Fortran program with calls to MPI routines, one is required to add the statement include 'mpif.h' which supplies needed MPI constants but not the Fortran 90 interface blocks since Fortran 77 compilers do not recognize interface blocks. The MPI-2 standard allows one to use the statement **use mpi** instead of **include 'mpif.h'** when compiling with a Fortran 90 compiler. We have created a module that not only supplies all the information in mpif.h but also contains the interface blocks for all MPI routines. This allows the Fortran 90 compiler to check each MPI routine for

- the data type of each argument,
- the intent of each argument, and
- the number of arguments.

MPI_CHECK automatically inserts the statement **use MPI** in each routine and supplies the necessary module. All **include 'mpif.h'** and **use mpi** statements in the original program are removed to avoid conflicts with our MPI module.

At Argonne National Laboratories, one can obtain an MPI module, (see ftp://ftp.mcs.anl.gov/pub/mpi). However, there were two problems with using this module in MPI-CHECK: (i) the module does not handle messages that are arrays of more than two dimensions, and (ii) the module does not contain the intent information for the arguments of the MPI routines. Some vendors, e.g. Silicon Graphics Incorporated (SGI), do provide the MPI module with intent information, but not all vendors do this. The intent information for all MPI-1 and MPI-2 routines can be found on the MPI Forum Web site [4]. Adding the intent information from the MPI Forum Web site to the MPI module obtained from Argonne National Laboratories was relatively easy, but adding all possible combinations of message array dimensionalities cause the module to become too large to compile, and caused the compiled modules to become unmanageably large. To avoid this problem, MPI-CHECK creates modules only for the MPI routines used in the MPI program to be checked. These modules are then compiled and linked with the executable. Creating these modules dynamically requires little time. MPI-CHECK accepts arrays up to dimension seven, the maximum allowed by Fortran.

The following example illustrates the usefulness of checking argument intent information.

```
use MPI
integer parameter :: p = 2, n = 5
real :: A(n)
```

```
...
call random_number(A) ! initialize array A
...
call mpi_comm_size(mpi_comm_world, p, ierror)
...
q=3
call mpi_send(A,q,mpi_real,2,1,mpi_comm_world, ierror)
```

Note that there are two errors in this program. In the call to mpi_comm_size, *p* is a constant and cannot have intent(out). Note also that the variable *q* has been initialized but its type has not been declared. Therefore *q* takes the default REAL type and hence is not used properly in the call to mpi_send. A Fortran 90 compiler with the MPI module in MPI-CHECK will flag both errors.

3. RUN-TIME CHECKING

MPI-CHECK accomplishes run-time checking by instrumenting the original program and then compiling and linking the resulting program to produce an instrumented executable. When the resulting executable is run, error/warning messages are automatically reported. The run-time portion of MPI-CHECK consists of a parser, a file containing MPI routine information (called the MPI Knowledge Base) and the code instrumentation portion. In this section, we first discuss each of these components of MPI-CHECK and then list the MPI problems that are automatically detected. MPI-CHECK run-time checking will check all MPI-1 and MPI-2 routines for the problems listed in Section 3.3. MPI-2 added the MPI_IN_PLACE option for most of the collective routines. MPI-CHECK does not recognize MPI_IN_PLACE as a valid argument for a buffer and issues an error message. Note that MPI-CHECK does not abort execution after issuing an error unless it detects that the following MPI call will generate an array out-of-bounds condition.

3.1. The MPI-CHECK parser

The MPI-CHECK parser assumes that the Fortran 90 syntax is correct and that there is at most one call to an MPI routine on each line. MPI-CHECK supports programs written in free or fixed format Fortran 90.

The MPI-CHECK parser copies all lines without MPI calls in the source program to the instrumented code without any change. When the parser detects an MPI call, it collects the following information: the routine name, the number of arguments, the name of each argument, the dimension of each argument (if it exists), and line number scope of an MPI call. The MPI-CHECK parser stores this information in the FUNCTION structure.

3.2. The MPI Knowledge Base

The information in the MPI Knowledge Base is used to determine what instrumentation is needed for each MPI routine. The MPI Knowledge Base contains information about each MPI routine. This includes the number of arguments, the data type of each argument, etc. The language bindings used to generate the MPI-CHECK Knowledge Base can be found on the MPI Forum Web site [4].

If more MPI routines are added to the MPI standard or if arguments of some MPI routines change in future versions of MPI, then it would be easy to update MPI-CHECK since only the MPI Knowledge Base will need to be updated.

3.3. The MPI-CHECK instrumentation

After parsing the MPI source code, MPI-CHECK inserts **use MPI_CHECK** (see Section 2) as the first line of the instrumented MPI code for the main program and for each function and subroutine. For each MPI call, MPI-CHECK retrieves the entry of this MPI routine from the MPI Knowledge Base, interprets the function of each argument, and then inserts additional statements before and/or after the call to the MPI routine into the instrumented MPI code. This section shows how MPI-CHECK instruments a program to detect MPI-specific problems.

3.3.1. Buffer data type inconsistency

MPI-CHECK compares the Fortran data type of each message in an MPI call with the corresponding MPI data type argument. If they are inconsistent, MPI-CHECK issues an error message. For example, for the following code:

```
real*8 :: A(m1,m2)
...
call mpi_send(A, m, mpi_integer, dest, tag, mpi_comm_world)
```

MPI-CHECK will issue the message:

```
Error: [File = filename, line = 30, argument = 3] data type mismatch, expecting MPI_REAL8
call mpi_send(A, m, mpi_integer, dest, tag, mpi_comm_world)
```

Although it is possible to write a correct MPI program that mixes data types, we consider this to be poor programming style that can easily lead to execution errors when running the MPI program on different machines. In the above example, MPI-CHECK will also issue an error message if the MPI data type in the call to mpi_send is MPI_REAL4. Matching data types between the Fortran type statement and the MPI call is necessary when writing portable MPI programs. The comparison of data types is simple for MPI_REAL, MPI_INTEGER, etc. To match an MPI data type such as MPI_REAL4 or MPI_DOUBLE_PRECISION, we check that the Fortran 90 type of the buffer is REAL and has the correct extent. This consistency check is not done for MPI_PACKED and MPI_BYTE because these types can be used with any of the other types.

This data type checking is accomplished by creating a generic interface named type_check. The generic interface for type_check defines specific interfaces not only for all the type-kind combinations but for all degrees of dimensionality including scalars and arrays of dimensionality from one to seven, the maximum allowed by Fortran.

MPI-CHECK checks buffers that are MPI derived types as follows. The parser scans for mpi_type_commit and stores the names of the committed derived data types in an array. If the MPI data type used in the MPI routine is neither a primitive MPI data type nor one of the committed data types, MPI-CHECK issues an error message.

The data consistency check is performed by the subroutine type_check. See statement 6 in the example in Figure 1, Section 3.3.6.

3.3.2. *Buffer out of bounds*

Another problem occurs when attempting to send or receive elements that are not within the declared bounds of an array. This can be checked using the Fortran 90 intrinsic functions LBOUND and UBOUND. If one calls an MPI routine as in the following example:

call mpi_send (A, m, mpi_real, dest, tag, comm, ierror)

no bounds checking for the starting address need be done, so MPI-CHECK does no checking in this case. In all other cases, MPI-CHECK checks the starting address. To illustrate how this is done, suppose we have the following MPI call:

call mpi_send (A(m1), m, mpi_real, dest, tag, comm, ierror)

then the starting address of the send buffer is inside the declared bounds if the following condition is satisfied:

$$\mathrm{lbound}(A, 1) \leqslant \mathrm{m1} \leqslant \mathrm{ubound}(A, 1) \tag{1}$$

where lbound and ubound are the Fortran 90 intrinsic functions that return the lower and upper bounds of an array. The ending address of the message is within the bounds of the declared message buffer if

$$\mathrm{lbound}(A, 1) \leqslant \mathrm{m1} + \mathrm{m} - 1 \leqslant \mathrm{ubound}(A, 1) \tag{2}$$

Note that this assumes that the datatype of A is the same as the MPI data type in the call and this may not be true. By using mpi_type_extent, MPI-CHECK removes this assumption. MPI detects an out-of-bounds condition for higher-dimensional arrays in a similar fashion. This check is performed by the subroutine buffer_check. See statement 7 in the example in Figure 1, Section 3.3.6.

3.3.3. *Improper placement of MPI_INIT*

MPI requires that the first call to an MPI routine must be to mpi_init with the possible exception of mpi_initialized. MPI-CHECK inserts a call to subroutine init_check prior to each MPI call except for mpi_initialized and mpi_init. This subroutine uses mpi_initialized to determine if mpi_init has been called. See statements 2 and 8 in the example in Figure 1, Section 3.3.6.

3.3.4. *Illegal message length*

MPI requires that the message length count is greater than or equal to zero. This checking is performed by calling the count_check subroutine, see statement 5 in the example in Figure 1, Section 3.3.6.

3.3.5. Invalid MPI rank

Let p be the number of processes involved in a communicator **comm** as given by the MPI call mpi_comm_size(comm,$\boldsymbol{p}$,ierr). Then the **rank** of a process involved in a communication must satisfy $0 \leqslant \mathbf{rank} \leqslant \boldsymbol{p} - 1$. MPI requires that the **root** and **dest** arguments to MPI routines are the **rank** of the calling processor, and that the **source** argument must either be a valid **rank** for the communicator or be the MPI defined constant MPI_ANY_SOURCE.

These conditions are checked by calling the rank_check subroutine, see statement 9 in the example in Figure 1, Section 3.3.6.

3.3.6. An example

We next present an example to show how to use MPI-CHECK and to illustrate how MPI-CHECK instruments a program. We assume the example program is stored in the file test.f90. To use MPI-CHECK to find MPI errors at either compile time or run-time, we issue the command

mpicheck f90 [compiler options] -o prog program.f90 –lmpi

This creates the instrumented executable prog. We now run the executable as usual, i.e.

mpirun –np 4 prog.

The following example MPI program has a number of MPI usage errors:

```
1.  implicit real*8 (a-h, o-z)
2.  integer, parameter :: n = 3
3.  real*8 A(n)
4.  include "mpif.h"
5.  integer p, rank, status(mpi_status_size)
6.  call mpi_comm_size(mpi_comm_world, p, ierror)
7.  call mpi_init(ierror)
8.  call mpi_comm_rank(mpi_comm_world, rank, ierror)
9.  if (rank == 0) then
10. do i = 1, n
11. A(i) = float(i)
12. enddo
13. do i = 1, p
14. call mpi_send(A, p, mpi_real4, i, 1, mpi_comm_world, ierror)
15. enddo
16. endif
17. if (rank > 0) then
18. call mpi_recv(A, p, mpi_real8, 0, 1, mpi_comm_world, status, ierror)
19. print *, 'On processor ', rank, 'A= ', A
20. endif
21. call mpi_finalize(ierror)
22. end
```

The following shows the instrumentation produced by MPI-CHECK only for lines 6, 7 and 14 of this program.

```
use MPI_CHECK
call init_check('?tl.f90',6,6,'mpi_comm_size')
call mpi_comm_size(mpi_comm_world, p, ierror)
call mpi_init(ierror)
    ...
call count_check('/tl.f90',14, 14,p>=0, 'p' 1,                  &
      'count should not be less than 0')
call type_check('/tl.f90',14, 14,A, mpi_real4, 2, .true. )
call buffer_check('/tl.f90',14, 14, extent_mpi(mpi_real4) *p+   &
  extent_fortran(A) *mpicheck_startarray()<=extent_fortran(A)*get_size(A),&
  'A',0,'message size exceeds the bounds of this array, please&
 &' check the message size.')
call init_check('/tl.f90',14, 14, 'mpi_send')
call rank_check('/tl.f90',14, 14,i>=0,and. i<get_commsize(mpi_comm_world) &
  ,'i', 3,'rank should be between 0 and p-1')
call mpi_send(A, p, mpi_real4, i, 1, mpi_comm_world, ierror)
      ...
```

Figure 1. An example of an instrumental program.

The first error in the program is that in line 6 of the original MPI program, **mpi_comm_size** is called before **mpi_init**. This error will be found by the subroutine **init_check** and MPI-CHECK will issue the message:

```
  Error: [File=/t1.f90, Line= 6 ] mpi_comm_size must be called after
calling mpi_init
         call mpi_comm_size(mpi_comm_world, p, ierror)
```

The second error is that A is declared to be of type **real*8** in the Fortran program but in the call to mpi_send at line 14, **mpi_real4** is used as the MPI data type. This error will be found by the subroutine **type_check**. MPI_CHECK issues the message:

```
   Error: [File=/t1.f90, Line= 14 Argument= 3 ] datatype mismatch,
expecting mpi_real8
             call mpi_send(A,p,mpi_real4,i,1,mpi_comm_world, ierror)
```

The third error, using p as the count instead of n in the mpi_send call, is an error only when $p > n$. When $p > n$, the error is found by the **buffer_check** subroutine and MPI_CHECK issues the message:

```
  Error: [File=/t1.f90, Line= 14 , Argument= 1 ] A, message size
exceeds the bounds of this array, please check the message size.
             call mpi_send(A,p,mpi_real4,i,1,mpi_comm_world, ierror)
```

The fourth error is that the fourth argument of mpi_send is the rank of the destination process and will not be between 0 and $p - 1$ when $i = p$. The subroutine **rank_check** finds this problem and reports the following message:

```
     Error: [File=/t1.f90, Line= 14 , Argument= 4 ] i, rank should be
between 0 and p-1
       call mpi_send(A,p,mpi_real4,i,1,mpi_comm_world, ierror)
```

4. MPI-CHECK OVERHEAD

MPI-CHECK introduces an additional overhead for both the compile and the execution phases of an MPI program. To illustrate how much overhead is introduced by MPI-CHECK, we compiled and executed the NAS LU Parallel Benchmark [5] using the class A problem size with four processors on an SGI Origin 2000. The compile time using –O3 without MPI-CHECK was 39 s and with MPI-CHECK the compile time was 214 s. 108 of the 214 s went to the generation of support files for MPI-CHECK. After the support files are generated, the compile time with MPI-CHECK is only 106 s; so additional compiles with MPI-CHECK only require 106 s. The execution time without MPI-CHECK was 192 s and 207 s with MPI-CHECK, about 8% slower. MPI-CHECK does not add any tables or arrays so additional memory requirements are negligible. The class A NAS LU Parallel Benchmark required an additional 5 MBytes of disk space for the MPI-CHECK support.

5. CONCLUSIONS AND FUTURE DIRECTIONS

MPI-CHECK is a tool developed to aid in the debugging of MPI programs that are written in free or fixed format Fortran 90. MPI-CHECK provides automatic compile-time and run-time checking of MPI programs. MPI-CHECK automatically detects the following problems:

- mismatch in argument type, kind, rank or number;
- messages which exceed the bounds of the source/destination array;
- negative message lengths;
- illegal MPI calls before MPI_INIT or after MPI_FINALIZE;
- inconsistencies between a message's declared type and the MPI data type used as an argument in the MPI routine; and
- actual arguments, which violate the INTENT, attribute for that dummy argument.

Currently, we are working on improvements to MPI-CHECK. The improvements will be in the areas of adding support for C++ and detecting deadlocks.

REFERENCES

1. Gropp W, Lusk E, Skjellum A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (2nd edn). MIT Press: Cambridge, MA, 1999.
2. MPI-CHECK. http://www.hpc.iastate.edu/MPI-CHECK.htm.
3. Vetter JS, Supinski BR. Dynamic software testing of MPI applications with Umpire. *Proceedings Supercomputing 2000*. IEEE Computer Society Press: Dallas, 2000.
4. MPICH. http://www.mpi-forum.org/docs/docs.html.
5. NAS Parallel Benchmarks. http://webserv.gsfc.nasa.gov/neumann/hive_comp/bmarks/node5.html.