MPI Application Development with MARMOT

Bettina Krammer
University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Outline

• Motivation
• Approaches and Tools
• Examples
• Future work and comparison with other tools
• Conclusion
Motivation

Motivation - Problems of Parallel Programming I

- All problems of serial programming
  - For example, use of non-initialized variables, typos, etc.
  - Is your code portable?
    - portable C/C++/Fortran code?
    - 32Bit/64Bit architectures
  - Compilers, libraries etc. might be buggy themselves
  - Hardware problems
  - Legacy code - a pain in the neck
Motivation - Problems of Parallel Programming II

- Additional problems:
  - Increased difficulty to verify correctness of program
  - Increased difficulty to debug N parallel processes
  - New parallel problems:
    - deadlocks
    - race conditions
    - Irreproducibility
- Errors may not be reproducible but occur only sometimes

Common MPI Programming Errors
Common MPI programming errors I – Collective Routines

- Argument mismatches (e.g. different send/recv-counts in Gather)
- Deadlocks: not all processes call the same collective routine
  - E.g. all procs call Gather, except for one that calls Allgather
  - E.g. all procs call Bcast, except for one that calls Send before Bcast, matching Recv is called after Bcast
  - E.g. all procs call Bcast, then Gather, except for one that calls Gather first and then Bcast

Common MPI programming errors II – Point-to-Point Routines

- Deadlocks: matching routine is not called, e.g.
  Proc0: MPI_Send(…)
  MPI_Recv(…)
  Proc1: MPI_Send(…)
  MPI_Recv(…)

- Argument mismatches
  - different datatypes in Send/Recv pairs, e.g.
    Proc0: MPI_Send(1, MPI_INT)
    Proc1: MPI_Recv(8, MPI_BYTE)
    Illegal!
Common MPI programming errors III – Point-to-Point Routines

- especially tricky with user-defined datatypes, e.g.
  - MPI_INT
  - MPI_DOUBLE
  - derived datatype 1: DER_1
  - derived datatype 2: DER_2
  - derived datatype 3: DER_3

  MPI_Send(2, DER_1), MPI_Recv(1, DER_2) is legal
  MPI_Send(2, DER_1), MPI_Recv(1, DER_3) is illegal

- different counts in Send/Recv pairs are allowed as Partial Receive
  - MPI_Send(1, DER_1), MPI_Recv(1, DER_2) is legal
  - MPI_Send(1, DER_1), MPI_Recv(1, DER_3) is legal
  - MPI_Send(1, DER_2), MPI_Recv(1, DER_1) is illegal

Common MPI programming errors IV – Point-to-Point Routines

- Incorrect resource handling
  - Non-blocking calls (e.g. Isend, Irecv) can complete without issuing test/wait call, BUT:
    - Number of available request handles is limited (and implementation defined)
  - Free request handles before you reuse them (either with wait/successful test routine or MPI_Request_free)
Common MPI programming errors V – Others

- Incorrect resource handling
  - Incorrect creation or usage of resources such as communicators, datatypes, groups, etc.
  - Reusing an active request
  - Passing wrong number and/or types of parameters to MPI calls (often detected by compiler)
- Memory and other resource exhaustion
  - Read/write from/into buffer that is still in use, e.g. by an unfinished Send/Recv operation
  - Allocated communicators, derived datatypes, request handles, etc. were not freed
- Outstanding messages at Finalize
- MPI-standard 2: I/O errors etc.

Common MPI programming errors VI – Race conditions

- Irreproducibility
  - Results may sometimes be wrong
  - Deadlocks may occur sometimes
- Possible reasons:
  - Use of wild cards (MPI_ANY_TAG, MPI_ANY_SOURCE)
  - Use of random numbers etc.
  - Nodes do not behave exactly the same (background load, …)
  - No synchronization of processes
- Bugs can be very nasty to track down in this case!
- Bugs may never occur in the presence of a tool (so-called Heisenbugs)
Common MPI programming errors VII – Portability issues

- MPI standard leaves some decisions to implementors, portability therefore not guaranteed!
  - "Opaque objects" (e.g. MPI groups, datatypes, communicators) are defined by implementation and are accessible via handles.
    - For example, in mpich, MPI_Comm is an int
    - In lam-mpi, MPI_Comm is a pointer to a struct
  - Message buffering implementation-dependent (e.g. for Send/Recv operations)
    - Use Isend/Irecv
    - Bsend (usually slow, beware of buffer overflows)
  - Synchronizing collective calls implementation-dependent
  - Thread safety not guaranteed

Approaches & Tools
(Parallel) Debuggers

- Most vendor debuggers have some support
- Debugging MPI programs with a “serial” debugger is hard but possible
  - MPIch supports debugging with gdb attached to one process
  - manual attaching to the processes is possible
- Commercial debuggers: Totalview, DDT

Special MPI tools & libraries

- MPI implementations offer (limited) support, e.g.
  - NEC Collectives Verification Library
    - Only for NEC MPI/SX, MPI/EX
  - mpich2 profiling library for collective functions
    - Checks correctness of collective calls and datatypes
    - Portable library
- Tools specially dedicated to analysis of MPI applications:
  - MPI-Check
  - Umpire
  - IMC
  - Marmot
What is MARMOT?

- Tool for the development of MPI applications
- Automatic runtime analysis of the application:
  - Detect incorrect use of MPI
  - Detect non-portable constructs
  - Detect possible race conditions and deadlocks
- MARMOT does not require source code modifications, just relinking
- C and Fortran binding of MPI -1.2 is supported, also C++ and mixed C/Fortran code
- Development is still ongoing (not every possible functionality is implemented yet…)
- Tool makes use of the so-called profiling interface
5. Parallel debugging and TotalView

Design of MARMOT

Application or Test Program

Profiling Interface

MARMOT core tool

MPI library

Debug Server (additional process)

What is the profiling interface?

- Defined in the MPI standard
- Every MPI routine can also be called as the nameshifted routine PMPI.
- This allows users to replace MPI routines by their own routines.
- Example (MARMOT): redefine the MPI calls
  
  ```c
  MPI_Send {
      doSomeChecks();
      PMPI_Send(…);
  }
  ```
Examples of Server Checks: verification between the nodes, control of program

- Everything that requires a global view
- Control the execution flow, trace the MPI calls on each node throughout the whole application
- Signal conditions, e.g. deadlocks (with traceback on each node.)
- Check matching send/receive pairs for consistency
- Check collective calls for consistency
- Output of human readable log file

Examples of Client Checks: verification on the local nodes

- Verification of proper construction and usage of MPI resources such as communicators, groups, datatypes etc., for example
  - Verification of MPI_Request usage
    - invalid recycling of active request
    - invalid use of unregistered request
    - warning if number of requests is zero
    - warning if all requests are MPI_REQUEST_NULL
  - Check for pending messages and active requests in MPI_Finalize
- Verification of all other arguments such as ranks, tags, etc.
Availability of MARMOT

- Tests on different platforms, using different compilers (Intel, GNU,...) and MPI implementations (mpich, lam, vendor MPIs,...), e.g.
  - IA32/IA64 clusters
  - Opteron clusters
  - Xeon EM64T clusters
  - IBM Regatta
  - NEC SX5,..., SX8

- Download and further information
  http://www.hlrs.de/organization/tsc/projects/marmot/
Example 1: request-reuse (source code)

```c
/* ** Here we re-use a request we didn't free before */

#include <stdio.h>
#include <assert.h>
#include "mpi.h"

int main( int argc, char **argv ) {
  int size = -1;
  int rank = -1;
  int value = -1;
  int value2 = -1;
  MPI_Status send_status, recv_status;
  MPI_Request send_request, recv_request;
  printf( "We call Irecv and Isend with non-freed requests.\n"
          );
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  printf( " I am rank %d of %d PEs\n", rank, size );

  if( rank == 0 ){
     /*** this is just to get the request used ***/
      MPI_Irecv( &value, 1, MPI_INT, 1, 18, MPI_COMM_WORLD,
                 &recv_request );
     /*** going to receive the message and reuse a non-freed request ***/
      MPI_Irecv( &value, 1, MPI_INT, 1, 17, MPI_COMM_WORLD,
                 &recv_request );
      MPI_Wait( &recv_request, &recv_status );
      assert( value = 19 );
  }
  if( rank == 1 ){
      value2 = 19;
     /*** this is just to use the request ***/
      MPI_Isend( &value, 1, MPI_INT, 0, 18, MPI_COMM_WORLD,
                 &send_request );
     /*** going to send the message ***/
      MPI_Isend( &value2, 1, MPI_INT, 0, 17, MPI_COMM_WORLD,
                 &send_request );
      MPI_Wait( &send_request, &send_status );
  }
  MPI_Finalize();
  return 0;
}
```

Example 1: request-reuse (source code continued)

```c
if( rank == 0 ){
    //** this is just to get the request used */
    MPI_Irecv( &value, 1, MPI_INT, 1, 18, MPI_COMM_WORLD,
               &recv_request );
    //** going to receive the message and reuse a non-freed request */
    MPI_Irecv( &value, 1, MPI_INT, 1, 17, MPI_COMM_WORLD,
               &recv_request );
    MPI_Wait( &recv_request, &recv_status );
    assert( value = 19 );
}
if( rank == 1 ){
    value2 = 19;
    //** this is just to use the request */
    MPI_Isend( &value, 1, MPI_INT, 0, 18, MPI_COMM_WORLD,
               &send_request );
    //** going to send the message */
    MPI_Isend( &value2, 1, MPI_INT, 0, 17, MPI_COMM_WORLD,
               &send_request );
    MPI_Wait( &send_request, &send_status );
}
MPI_Finalize();
return 0;
```
Example 1: request-reuse (output log)

We call Irecv and Isend with non-freed requests.
1 rank 0 performs MPI_Init
2 rank 1 performs MPI_Init
3 rank 0 performs MPI_Comm_size
4 rank 1 performs MPI_Comm_size
5 rank 0 performs MPI_Comm_rank
6 rank 1 performs MPI_Comm_rank
    I am rank 0 of 2 PEs
7 rank 0 performs MPI_Irecv
    I am rank 1 of 2 PEs
8 rank 1 performs MPI_Isend
9 rank 0 performs MPI_Irecv
10 rank 1 performs MPI_Isend
   ERROR: MPI_Irecv Request is still in use !!
11 rank 0 performs MPI_Wait
   ERROR: MPI_Isend Request is still in use !!
12 rank 1 performs MPI_Wait
13 rank 0 performs MPI_Finalize
14 rank 1 performs MPI_Finalize

Example 2: deadlock (source code)

/* This program produces a deadlock.
   ** At least 2 nodes are required to run the program.
   **
   ** Rank 0 recv a message from Rank 1.
   ** Rank 1 recv a message from Rank 0.
   **
   ** AFTERWARDS:
   ** Rank 0 sends a message to Rank 1.
   ** Rank 1 sends a message to Rank 0.
   */

#include <stdio.h>
#include "mpi.h"

int main( int argc, char** argv ){
    int rank = 0;
    int size = 0;
    int dummy = 0;
    MPI_Status status;
Example 2: deadlock (source code continued)

```c
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if( size < 2 ){
    fprintf(stderr," This program needs at least 2 PEs!\n" );
}
else {
    if( rank == 0 ){
        MPI_Recv( &dummy, 1, MPI_INT, 1, 17, MPI_COMM_WORLD, &status );
        MPI_Send( &dummy, 1, MPI_INT, 1, 18, MPI_COMM_WORLD );
    }
    if( rank == 1 ){
        MPI_Recv( &dummy, 1, MPI_INT, 0, 18, MPI_COMM_WORLD, &status );
        MPI_Send( &dummy, 1, MPI_INT, 0, 17, MPI_COMM_WORLD );
    }
}
MPI_Finalize();
```

Example 2: deadlock (output log)

```
$ mpirun -np 3 deadlock1
1 rank 0 performs MPI_Init
2 rank 1 performs MPI_Init
3 rank 0 performs MPI_Comm_rank
4 rank 1 performs MPI_Comm_rank
5 rank 0 performs MPI_Comm_size
6 rank 1 performs MPI_Comm_size
7 rank 0 performs MPI_Recv
8 rank 1 performs MPI_Recv
8 Rank 0 is pending!
8 Rank 1 is pending!
WARNING: deadlock detected, all clients are pending
```
Example 2: deadlock (output log continued)

Last calls (max. 10) on node 0:
- timestamp = 1: MPI_Init( *argc, ***argv )
- timestamp = 3: MPI_Comm_rank( comm, *rank )
- timestamp = 5: MPI_Comm_size( comm, *size )
- timestamp = 7: MPI_Recv( *buf, count = -1,
  datatype = non-predefined datatype, source = - 1, tag = -1, comm, *status)

Last calls (max. 10) on node 1:
- timestamp = 2: MPI_Init( *argc, ***argv )
- timestamp = 4: MPI_Comm_rank( comm, *rank )
- timestamp = 6: MPI_Comm_size( comm, *size )
- timestamp = 8: MPI_Recv( *buf, count = -1,
  datatype = non-predefined datatype, source = - 1, tag = -1, comm, *status )

Future Directions

Functionality
- More checks
- MPI-2
- OpenMP/MPI

Usability
- GUI

Performance
- Scalability

Combination with other tools
- Debugger
- Vampir
Thanks for your attention
Example - Medical Application B_Stream

- Calculation of blood flow with 3D Lattice-Boltzmann method
- 16 different MPI calls:
  - MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Pack, MPI_Bcast, MPI_Unpack, MPI_Cart_create, MPI_Cart_shift, MPI_Cart_rank, MPI_Send, MPI_Recv, MPI_Bcast, MPI_Barrier, MPI_Reduce, MPI_Sendrecv, MPI_Wtime, MPI_Finalize
- Around 6500 lines of code
- We use different input files that describe the geometry of the artery: tube, tube-stenosis, bifurcation

Example: B_Stream (blood flow simulation, tube)

- Tube geometry: simplest case, just a tube with about the same radius everywhere
- Running the application without/with MARMOT:
  mpirun -np 3 B_Stream 500. tube
  mpirun -np 4 B_Stream_marmot 500. tube
- Application seems to run without problems
Example: B_Stream (blood flow simulation, tube-stenosis)

- Tube-stenosis geometry: just a tube with varying radius
- Without MARMOT:
  ```
  mpirun -np 3 B_Stream 500. tube-stenosis
  ```
  Application seems to be hanging
- With MARMOT:
  ```
  mpirun -np 4 B_Stream_marmot 500. tube-stenosis
  ```
  Deadlock found

WARNING: all clients are pending!

Iteration step:
Calculate and exchange results with neighbors

Communicate results among all procs
Example: B_Stream (blood flow simulation, tube-stenosis)

deadlock

Node 0

timestamp= 9319: MPI_Sendrecv(*sendbuf, sendcount = 7220, sendtype = MPI_DOUBLE, dest = 1, sendtag = 1, *recvbuf, recvcount = 7220, recvtype = MPI_DOUBLE, source = 2, recvtag = 1, comm = self-defined communicator, *status)

timestamp= 9321: MPI_Barrier(comm = MPI_COMM_WORLD)

timestamp= 9322: MPI_Bcast(*buffer, count = 3, datatype = MPI_DOUBLE, root = 0, comm = MPI_COMM_WORLD)

timestamp= 9323: MPI_Barrier(comm = MPI_COMM_WORLD)

Node 1

timestamp= 9324: MPI_Comm_rank(comm = MPI_COMM_WORLD, *rank)

timestamp= 9325: MPI_Bcast(*buffer, count = 3, datatype = MPI_DOUBLE, root = 0, comm = MPI_COMM_WORLD)

timestamp= 9326: MPI_Comm_rank(comm = MPI_COMM_WORLD, *rank)

Node 2

timestamp= 9327: MPI_Bcast(*buffer, count = 3, datatype = MPI_DOUBLE, root = 0, comm = MPI_COMM_WORLD)

Example: B_Stream (blood flow simulation, tube-stenosis)

– Code Analysis

main {
  ...
  num_iter = calculate_number_of_iterations();
  for (i=0; i < num_iter; i++) {
    computeBloodflow();
  }

  writeResults();
  ...
  // communicate results with neighbors
  MPI_Bcast(...);

  if (radius < x) num_iter = 50;
  if (radius >= x) num_iter = 200;

  // ERROR: it is not ensured here that all
  // procs do the same (maximal) number
  // of iterations

  if (radius < x) num_iter = 50;
  if (radius >= x) num_iter = 200;

  // ERROR: it is not ensured here that all
  // procs do the same (maximal) number
  // of iterations

Be careful if you call functions with hidden MPI calls!
Example: B_Stream (blood flow simulation, bifurcation)

- Bifurcation geometry: forked artery
- Without MARMOT:
  mpirun -np 3 B_Stream 500. bifurcation
  ...
  Segmentation fault
  (platform dependent if the code breaks here or not)
- With MARMOT:
  mpirun -np 4 B_Stream_marmot 500. bifurcation
  Problem found at collective call MPI_Gather
MARMOT
Performance with real applications

Air pollution modelling

- Air pollution modeling with STEM-II model
- Transport equation solved with Petrov-Crank-Nikolson-Galerkin method
- Chemistry and Mass transfer are integrated using semi-implicit Euler and pseudo-analytical methods
- 15500 lines of Fortran code
- 12 different MPI calls:
  - MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Type_extent, MPI_Type_struct, MPI_Type_commit, MPI_Type_hvector, MPI_Bcast, MPI_Scatterv, MPI_BARRIER, MPI_Gatherv, MPI_Finalize.
STEM application on an IA32 cluster with Myrinet

Comparison with other approaches and Future Directions
Future Direction of MARMOT

- MARMOT will continue to be developed jointly by ZIH and HLRS

Future Directions

Functionality
- More checks
- MPI-2
- OpenMP/MPI

Usability
- GUI

Performance
- Scalability

Combination with other tools
- Debugger
- Vampir
Online vs. Offline Checking

- Offline Checking (Intel Message Checker):
  + History information available
  + Less intrusive at runtime
  - Large file space needed for trace file
  - Limited scalability of analysis
- Online Checking (Marmot):
  - Reduced performance of running application
  - Only limited history/time line information
  + No limit for the runtime of the program
  + The program is still "alive" when the error is detected, further online analysis is possible

MPI implementation with checking

+ Scalability and performance
+ Avoids duplicating internal management tasks
+ Avoids re-implementation of existing functionality
- No history/time line information
- Performance and reliability is always more important than checks
- Portability problems are not a major focus
Comparison of systems

<table>
<thead>
<tr>
<th></th>
<th>Online</th>
<th>Offline</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checks</td>
<td>++</td>
<td>++</td>
<td>0</td>
</tr>
<tr>
<td>Scalability and Performance</td>
<td>+−</td>
<td>+−</td>
<td>++</td>
</tr>
<tr>
<td>Runtime scalability</td>
<td>++</td>
<td>−</td>
<td>++</td>
</tr>
<tr>
<td>Timeline information</td>
<td>−</td>
<td>++</td>
<td>−</td>
</tr>
<tr>
<td>Combination with other tools</td>
<td>+</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>

Conclusion

- Parallel programming in general and the MPI standard contains enough pitfalls to raise a need for checking/correctness/confidence tools
- Different tools and approaches with different advantages and disadvantages
- MARMOT is a freely available solution that has demonstrated its usefulness with various real applications
- A combination of tools will offer the best solution for the program developer
- Not every error can be detected by tools
Thanks for your attention

Backup
Motivation - Problems of Parallel Programming I

- All problems of serial programming
  - For example, use of non-initialized variables, typos, etc.
  - Is your code portable?
    - portable C/C++/Fortran code?
    - 32Bit/64Bit architectures
  - Compilers, libraries etc. might be buggy themselves
  - Legacy code - a pain in the neck

Example: B_Stream (B_Stream.cpp)

```c
#ifdef PARALLEL
MPI_Barrier (MPI_COMM_WORLD);
MPI_Reduce (&nr_fluids, &tot_nr_fluids, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
//Calculation of porosity
if (ge.me == 0)
{
  Porosity = ((double) tot_nr_fluids) / (ge.global_dim[0] * ge.global_dim[1] * ge.global_dim[2]);
}
#else
Porosity = ((double) tot_nr_fluids) / (ge.global_dim[0] * ge.global_dim[1] * ge.global_dim[2]);
#endif
```

ERROR: Parallel code is not executed because of typo
Example: B_Stream (communication.cpp)

src/communication.h:
    MPI_Comm topology_comm2;

src/communication.cpp:

    //--- Sends the populations of the current processor to
    // the east and receives from the west ---
    void comm::send_east(int *neighbours, int top, int* pos_x)
    {
        topology_comm2 = top;
        // Send/Receive the data
        MPI_Sendrecv(send_buffer, L[1]*L[2]*CLNBR,
                      MPI_DOUBLE, neighbours[EAST], tag,
                      recv_buffer, L[1]*L[2]*CLNBR,
                      MPI_DOUBLE, neighbours[WEST], tag,
                      topology_comm2, &status);
    ...

According to MPI standard: MPI_Comm
BUT here we pass an int

Example: B(Stream (communication.cpp)

• This code works with mpich, because in mpi.h:
  /* Communicators */
  typedef int MPI_Comm;
  #define MPI_COMM_WORLD 91
  #define MPI_COMM_SELF  92

• This code does not work with lam-mpi, because in mpi.h:
  typedef struct _comm *MPI_Comm;

Compilation error:
B_Stream/src/communication.cpp:172: invalid conversion
from `int' to `_comm'*'

Use handles to access opaque objects like communicators!
Use proper conversion functions if you want to map
communicators to ints and vice versa!
Example: BStream – summary of problems

- Different errors occur on different platforms (different compilers, different MPI implementations,…)
- Different errors occur with different input files
- Not all errors can be found with tools

Medical Application

- Calculation of blood flow with Lattice-Boltzmann method
- Stripped down application with 6500 lines of C code
- 14 different MPI calls:
  - MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Pack,
  - MPI_Bcast, MPI_Unpack,
  - MPI_Cart_create, MPI_Cart_shift,
  - MPI_Send, MPI_Recv, MPI_Barrier,
  - MPI_Reduce, MPI_Sendrecv,
  - MPI_Finalize
Medical application on an IA32 cluster with Myrinet

![Graph showing time per iteration vs. processors for native MPI and MARMOT.]

Message statistics with native MPI

![Message statistics chart for native MPI.]
Message statistics with MARMOT

Medical application on an IA32 cluster with Myrinet without barrier
Barrier with native MPI

Barrier with MARMOT