Parallel Debugging

Bettina Krammer, Matthias Müller, Pavel Neytchev, Rainer Keller

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

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger Dijkstra

Outline

- Motivation
- Approaches and Tools
  - Memory Tracing Tools
    - **Valgrind** → see talk [5a] on Thursday
  - MPI-Analysis Tools
    - **Marmot**
  - Debuggers
    - **gdb**
    - **DDT**
    - **TotalView**

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?**
    - **portable MPI code?**
    - **32Bit/64Bit architectures**
  - Compilers, libraries etc. might be buggy themselves
  - 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

Motivation - Problems of Parallel Programming III

- Typical problems with newly parallelized programs: the program
  - does not start
  - ends abnormally
  - deadlocks
  - gives wrong results

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 points 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

Tools and Techniques to Avoid and Remove Bugs

- Programming techniques
- Static Code analysis
  - Compiler (with –Wall flag or similar), lint
- Runtime analysis
  - Memory tracing tools
  - Special OpenMP tools (assure, thread checker)
  - Special MPI tools
- Post mortem analysis
  - Debuggers

Programming Techniques

- Think about a verbose execution mode of your program
- Use a careful/paranoid programming style
  - keep it simple, don't try to be too clever
  - check invariants and pre-requisites (assert(m>=0), assert(v<c))
  - comment your code
  - Use descriptive names for routines, variables, etc.
  - write portable code

Static Code Analysis – Compiler Flags

- Use the debugging/assertion techniques of the compiler
  - use debug flags (-g), warnings (-Wall)
  - array bound checks in Fortran
  - use memory debug libraries (-lefence)

skipped

Avoiding Debuggers

- Write portable programs
  - it avoids future problems
    - **architectures/platforms have a short life**
    - **all compilers and libraries have bugs**
    - **all languages and standards include implementation defined behavior**
  - running on different platforms and architectures significantly increases the reliability
- Use verification tools for parallel programming like assure

MARMOT

Bettina Krammer

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

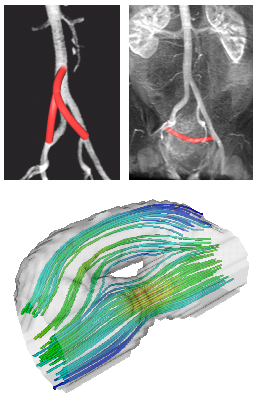
What is MARMOT?

- MPI analysis and checking tool to verify at runtime if an application conforms to the MPI standard.
- Library written in C++ that will be linked to the application.
- No source code modification of the application is required.
- Additional process working as debug server, i.e. the application will have to be run with `mpirun` for n+1 instead of n processes.
- Implementation of C and Fortran language binding of MPI-1.2 standard.
- Environment variables for tool behaviour and output (report of errors, warnings and/or remarks, trace-back, etc.).
- After the execution of the program the user can read a logfile to check for potential problems.

Availability of MARMOT

- Tests on different platforms, using different compilers and MPI implementations, e.g.
  - IA32/IA64 clusters (Intel, Gnu compiler) with mpich or lam
  - IBM Regatta
  - NEC SX5 and later
- Download and further information
  http://www.hlrs.de/organization/tsc/projects/marmot/

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_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-stenosis)

- Tube-stenosis geometry: just a tube with varying radius
- Without MARMOT:

```
mpirun -np 3 B_Stream 500. tube-stenosis
```

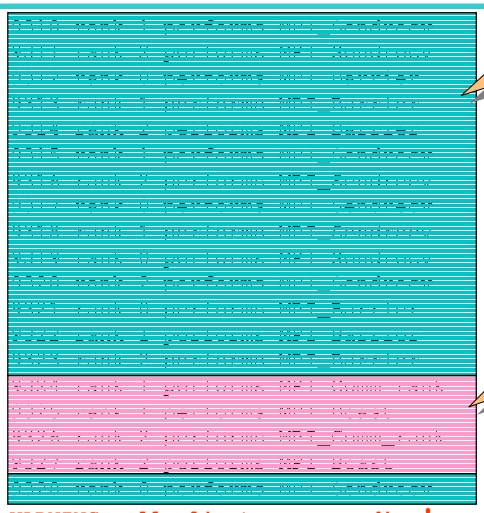
- Application is hanging

- With MARMOT:

```
mpirun -np 4 B_Stream_marmot 500. tube-
stenosis
```

- Deadlock found

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



Iteration step: Calculate and exchange results with neighbors

Communicate results among all procs

WARNING: all clients are pending!

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

```
Node 0
timestamp= 9321: MPI_Barrier(comm = MPI_COMM_WORLD)
timestamp= 9328: MPI_Sendrecv(*sendbuf, sendcount = 7220, sendtype =
   MPI_DOUBLE, dest = 2, sendtag = 1, *recvbuf, recvcount = 7220,
   recvtype = MPI_DOUBLE, source = 1, recvtag = 1, comm = self-defined
   communicator, *status)
Node 1
timestamp= 9318: MPI_Sendrecv(*sendbuf, sendcount = 7220, sendtype =
   MPI_DOUBLE, dest = 2, sendtag = 1, *recvbuf, recvcount = 7220,
   recvtype = MPI_DOUBLE, source = 0, recvtag = 1, comm = self-defined
   communicator, *status)
timestamp= 9322: MPI_Barrier(comm = MPI_COMM_WORLD)
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)
Node 2
timestamp= 9320: MPI_Sendrecv(*sendbuf, sendcount = 7220, sendtype =
   MPI_DOUBLE, dest = 0, sendtag = 1, *recvbuf, recvcount = 7220,
   recvtype = MPI_DOUBLE, source = 1, recvtag = 1, comm = self-defined
   communicator, *status)
timestamp= 9323: MPI_Barrier(comm = MPI_COMM_WORLD)
timestamp= 9326: MPI_Comm_rank(comm = MPI_COMM_WORLD, *rank)
timestamp= 9327: MPI_Bcast(*buffer, count = 3, datatype = MPI_DOUBLE,
   root = 0, comm = MPI_COMM_WORLD)
```

Example: B_Stream (blood – Code Analysis

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

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

CalculateSomething();
// exchange results with neighbors
MPI_Sendrecv(…);

// communicate results with neighbors
MPI_Bcast(…);

Be careful if you call functions with hidden MPI calls!

skipped

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

skipped

Example: B_Stream (blood flow simulation, bifurcation)

```
9319 rank 2 performs MPI_Sendrecv
9320 rank 1 performs MPI_Sendrecv
9321 rank 1 performs MPI_Barrier
9322 rank 2 performs MPI_Barrier
9323 rank 0 performs MPI_Barrier
9324 rank 0 performs MPI_Comm_rank
9325 rank 1 performs MPI_Comm_rank
9326 rank 2 performs MPI_Comm_rank
9327 rank 0 performs MPI_Bcast
9328 rank 1 performs MPI_Bcast
9329 rank 2 performs MPI_Bcast
9330 rank 0 performs MPI_Bcast
9331 rank 1 performs MPI_Bcast
```

Parallel Debugging [05]

skipped

Example: B_Stream (blood flow simulation, bifurcation)

```
9332 rank 2 performs MPI_Bcast
9333 rank 0 performs MPI_Gather
9334 rank 1 performs MPI_Gather
9335 rank 2 performs MPI_Gather
/usr/local/mpich-1.2.5.2/ch_shmem/bin/mpirun: line
   1: 10163
Segmentation fault
   /home/rusbetti/B_Stream/bin/B_Stream_marmot
"500." "bifurcation"
   9336 rank 1 performs MPI_Sendrecv
9337 rank 2 performs MPI_Sendrecv
9338 rank 1 performs MPI_Sendrecv
WARNING: all clients are pending!
```

skipped

Example: B_Stream (blood flow simulation, bifurcation)

```
1. Last calls  on node 0:
   timestamp= 9327: MPI_Bcast(*buffer, count = 3, datatype =
   MPI_DOUBLE, root = 0, comm = MPI_COMM_WORLD)
   timestamp= 9330: MPI_Bcast(*buffer, count = 3, datatype =
   MPI_DOUBLE, root = 0, comm = MPI_COMM_WORLD)
   timestamp= 9333: MPI_Gather(*sendbuf, sendcount = 266409,
   sendtype = MPI_DOUBLE, *recvbuf, recvcount = 266409,
   recvtype = MPI_DOUBLE, root = 0, comm = MPI_COMM_WORLD)
Last calls on node 1:
   timestamp= 9334: MPI_Gather(*sendbuf, sendcount = 258336,
   sendtype = MPI_DOUBLE, *recvbuf, recvcount = 258336,
   recvtype = MPI_DOUBLE, root = 0, com  = MPI_COMM_WORLD)
   timestamp= 9336:                      sendcount =
   13455, sendtype =                     sendtag = 1,
   *recvbuf, recvcou                     MPI_DOUBLE,
   source = 2, recvt                     ined
   communicator, *st
   timestamp= 9338: MPI_Sendrecv(*sendbuf, sendcount =
   13455, sendtype = MPI_DOUBLE, dest = 2, sendtag = 1,
   *recvbuf, recvcount = 13455, recvtype = MPI_DOUBLE,
   source = 0, recvtag = 1, comm = self-defined
   communicator, *status)
```

ERROR: Root 0 has different counts than rank 1 and 2

skipped

Example: B_Stream (blood flow simulation, bifurcation)

```
Last calls on node 2:
   timestamp= 9332: MPI_Bcast(*buffer, count = 3, datatype
   = MPI_DOUBLE, root = 0, comm = MPI_COMM_WORLD)
   timestamp= 9335: MPI_Gather(*sendbuf, sendcount =
   258336, sendtype = MPI_DOUBLE, *recvbuf, recvcount =
   258336, recvtype = MPI_DOUBLE, root = 0, comm =
   MPI_COMM_WORLD)
   timestamp= 9337: MPI_Sendrecv(*sendbuf, sendcount =
   13455, sendtype = MPI_DOUBLE, dest = 1, sendtag = 1,
   *recvbuf, recvcount = 13455, recvtype = MPI_DOUBLE,
   source = 0, recvtag = 1, comm = self-defined
   communicator, *status)
```

skipped

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

skipped

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!

skipped

Example: B_Stream (serial/parallel code in one file)

It is good to keep a working serial version, e.g. with

```
#ifdef PARALLEL
     {parallel code}
#else
     {serial code}
#endif
```


skipped

Example: B_Stream (B_Stream.cpp)

```
#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

skipped

Example: B_Stream – compile errors

- Compile error on our NEC Xeon EM64T cluster with voltaire_icc_dfl mpi:

```
/opt/streamline/examples/B_Stream/src/B_Stream.cpp(
  272): warning #181: argument is incompatible with
  corresponding format string conversion
        printf(" Physical_Viscosity =%lf, Porosity
  =%d \n",Mju,Porosity);
```

- Other compilers don't care

```
double Mju, Porosity;
printf(" Physical_Viscosity =%lf, Porosity =%d \n",
  Mju,Porosity);
```

- Have a look at compiler warnings: a warning on one platform can be an error on another platform!

skipped

Example: B_Stream - running

- Running the application

```
mpirun -np <np> B_Stream <Reynolds> <geometry-file>
```

  - With `10 <= Reynolds <= 500`
  - `geometry-file` = `tube` or `tube-stenosis` or `bifurcation` (read in the files `tube.conf` and `tube.bs` etc.)
- For example

```
mpirun -np 3 B_Stream 500. tube
```

skipped

Example: B_Stream (read in commandline parameters)

- Read in command line parameters

```
int main (int argc, char **argv){
      double Reynolds;
      MPI_Init(&argc, argv);
      // Getting of arguments from user
      if (argc != 1) {
          Reynolds = atof (argv[1]);
…}…
```

- Not safe! Better to do it with a Bcast from rank 0 to everyone:

```
if (rank == 0 && argc != 1) {
          Reynolds = atof (argv[1]);
}
Bcast(&Reynolds, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```


skipped

Example: BStream – start problem

- On our Nocona Xeon EM64T cluster with voltaire mpi:

```
mpirun_ssh -np 3 -V 3  -hostfile $PBS_NODEFILE ./B_Stream_dfl 500.
tube

mpirun_ssh: Starting all 3 processes...          [OK]
mpirun_ssh: Accepting incomming connections...
mpirun_ssh: Accepted connection 1....
mpirun_ssh: 1.  Reading rank...                  0
mpirun_ssh: 1.  Reading length of the data...
mpirun_ssh: 1.  Reading the data                         [OK]
mpirun_ssh: Accepted connection 2....
mpirun_ssh: 2.  Reading rank...                  2
mpirun_ssh: 2.  Reading length of the data...
mpirun_ssh: 2.  Reading the data                         [OK]
mpirun_ssh: Accepted connection 3....
mpirun_ssh: 3.  Reading rank...                  1
mpirun_ssh: 3.  Reading length of the data...
mpirun_ssh: 3.  Reading the data                         [OK]
mpirun_ssh: Writing all data...                  [OK]
mpirun_ssh: Shutting down all our connections...                [OK]
```

skipped

Example: BStream – start problem

- Code works with mpich but not with voltaire mpi
- Program exits immediately after start
- Reason: currently unknown

- This sort of problem is often caused by
  - Missing/wrong compile flags
  - Wrong versions of compilers, libraries etc.
  - Bugs in mpi implementation etc.
  - System calls in your code
- Ask your admin

skipped

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

skipped

Example: B_Stream (blood flow simulation, tube)

```
54 rank 1 performs MPI_Cart_shift
55 rank 2 performs MPI_Cart_shift
56 rank 0 performs MPI_Send
57 rank 1 performs MPI_Recv
WARNING: MPI_Recv: Use of MPI_ANY_SOURCE may cause race
   conditions!
58 rank 2 performs MPI_Recv
WARNING: MPI_Recv: Use of MPI_ANY_SOURCE may cause race
   conditions!
59 rank 0 performs MPI_Send
60 rank 1 performs MPI_Recv
WARNING: MPI_Recv: Use of MPI_ANY_SOURCE may cause race
   conditions!
61 rank 0 performs MPI_Send
62 rank 1 performs MPI_Bcast
63 rank 2 performs MPI_Recv
WARNING: MPI_Recv: Use of MPI_ANY_SOURCE may cause race
   conditions!
64 rank 0 performs MPI_Pack
65 rank 2 performs MPI_Bcast
```


skipped

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

Parallel Debuggers

Parallel Debuggers

- Most vendor debuggers have some support
- gdb has basic support for threads
- Debugging MPI programs with a “scalar” debugger is hard but possible
  - MPIch supports debugging with gdb attached to one process
  - manual attaching to the processes is possible
- TotalView is a good but expensive commercial tool
- DDT is an alternative

GNU Debugger (GDB)

What is gdb?

- gdb is the GNU free debugger.
- Features:
  - Set breakpoints
  - Single-stepping
  - examine variables, program stack, threads, etc.
- It supports C, C++, Fortran and many other programming languages.
- It supports also different memory model libraries like OpenMP and theoretically MPI (i.e. mpich).
- Ddd is a GUI for gdb
  http://www.gnu.org/software/ddd/

Gdb usage

- Compile with `-g` option
- Start gdb, for example with the command
  `gdb` or `gdb <progname>` or `gdb <progname> <corefile>`
- **OpenMP**: set the `OMP_NUM_THREADS` environment variable and start your program with `gdb <progname>`
- **MPI**: `mpirun -gdb -np 4 <progname>`
  Start the first process under gdb where possible
  If your MPI programm takes some arguments, you may have to set them explicitly in gdb with the `set args` command!
- More information:
  http://www.gnu.org/software/gdb/gdb.html

skipped

Gdb – useful commands I

`file progname`	load program from inside gdb
`run`	run program
`quit`	leave gdb
`break linenumber`	set breakpoint at the given line number
`delete breaknumber`	remove breakpoint with the given number
`info breakpoints`	list current breakpoints with some information
`list line` or `function`	lists the source code at the given line number or function name. Both of the parameters are optional
`continue`	when stopped at breakpoint, continue the program execution
`next`	when stopped at breakpoint, continuous step by step (line by line)
`step`	when stopped at breakpoint, step program until it reaches a different source line

skipped

Gdb – useful commands II

`backtrace`	prints all stack frames
`info threads`	list the IDs of the currently known threads
`thread threadnumber`	switches between threads, where threadnumber is the thread ID showed by info threads
`print varname`	print value (i.e. of variable) or expression
`set args arguments`	set arguments to use
`show args`	view arguments

Distributed Debugging Tool (DDT)

What is DDT?

- Parallel debugger
- Source level debugging for C, C++, F77, F90
- MPI, OpenMP
- SMPs, Clusters
- Available on Linux distributions and Unix
- GUI (independent of platform, based on QT libraries)
- Available on most platforms
- Commercial tool
- More information
  http://www.allinea.com/
  http://www.hlrs.de/organization/tsc/services/tools/debugger/ddt/

Parallel Debugging [05]

skipped

Availability of DDT

- Linux:
  - Linux IA32 (Intel and AMD)
  - Linux IA64
  - Linux Opteron
- Unix
  - PowerPC (AIX)
  - SGI Altix
  - SGI Irix
  - SUN Sparc
  - PA-Risc and Itanium Superdome

skipped

Availability of DDT at HWW

- Installed on
  - cacau
  - strider
  - ia64
  - volvox
- Usage: set your environment by running

```
module load ddt
```

More information:

http://www.hlrs.de/organization/tsc/services/tools/debugger/ddt/

Availability of DDT at University Stuttgart

- 512-user/512-proc Floating License for University Stuttgart:
  1. Download Software from http://www.allinea.com
  2. Copy the campus license into the **Licence** file.

More information about campus licenses available at

http://www.hlrs.de/organization/tsc/services/tools/campus/#ddt

DDT usage

- Compile with -g compiler switch
- Command name: **ddt**
- To start debugging with DDT simply type:

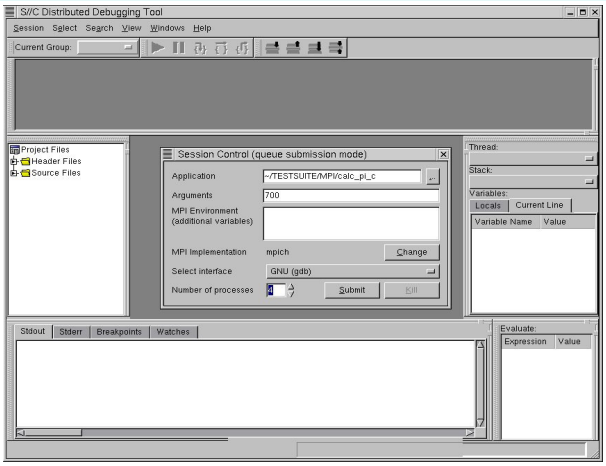
```
% $DDT myprog arguments to myprog
```


DDT Look & Feel

DDT Main Window

Configuration Window

and all belonging Panes (Thread, Stack, Output, Source code, etc.)



DDT Main/Process Window

MPI Groups

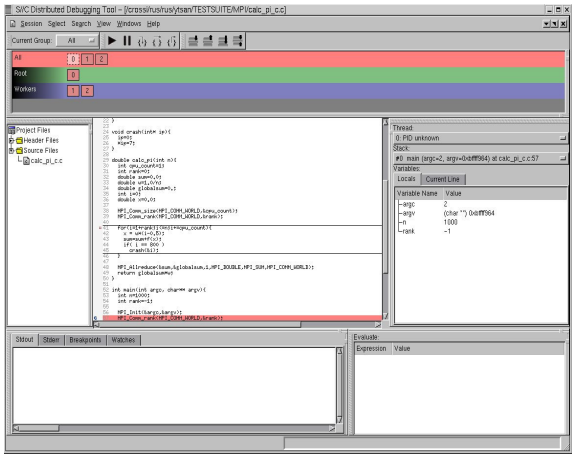
File browse and Source pane

Output,

Breakpoints,

Watch

Pane



Thread,

Stack,

Local and Global Variables

Pane

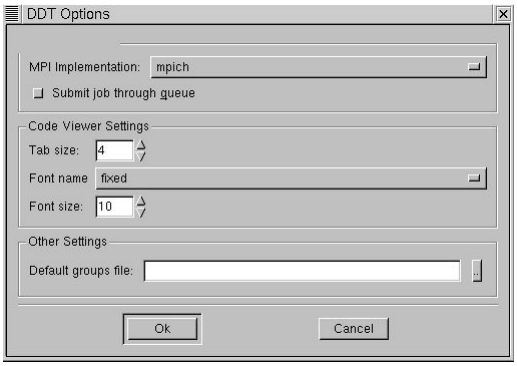
Evaluation window

skipped

DDT Options Window

The Options window: here is the MPI library implementation selected

The program can also be submitted through some batch system

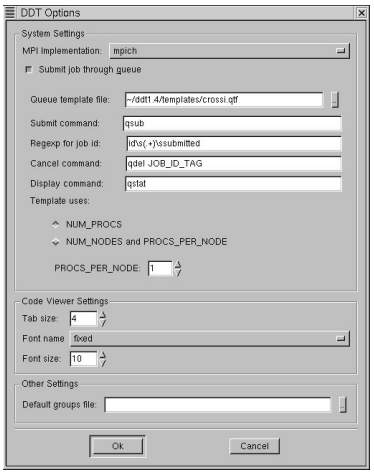


skipped

DDT Options Window (Queue)

The Options windows:
This program uses mpich as MPI implementation and starts the program through PBS batch system

For more information about the Template files and the commands, please read the DDT Users Manual.



Parallel Debugging [05]

H L R S

skipped

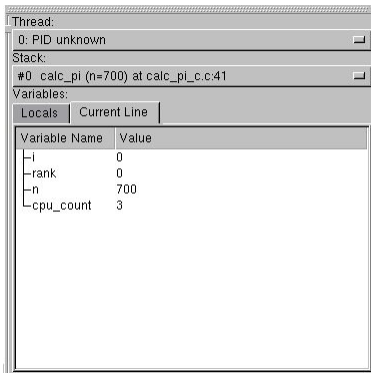
DDT Thread and Stack Window

Thread Pane:
Switch between all program threads

Stack Pane:
Switch between the functions in the selected thread

Variables Pane:
Local variables:
Shows the variables value for the function in which ddt is currently stopped

Current Line:
Shows every variable value between the two lines selected by the user



H L R S

DDT Source Pane

With the right mouse button Set or Remove a breakpoint at the selected line

When the program is running and stopped at a breakpoint, the line is coloured in red (by OpenMP programs) and red, blue or green by programs using MPI

```
 3 //  Simple OpenMP Program to calculate PI
 4 //
 5 //
 6 //  Author: Matthias Mueller <mueller@hlrs.de>
 7 //  Thu Dec 28 15:12:09 CET 2000
 8 //
 9 //  Usage: calc_pi [iterations]
10 //  the program will crash if iterations >= 800
11 //
12 //  This is the intended behavior. Because the program
13 //  test a debugger.
14 */
15
16 #include <stdio.h>
17 #include <stdlib.h>
18
19 double f(double a){
20   return 4.0/(1.0 + a*a);
21 }
22
23 void crash(int* ip){
24   ip=0;
25   *ip=7;
26 }
27
28 double calc_pi(int n){
29   double sum=0.0;
30   double w=1.0/n;
31   int i=0;
32   double x=0.0;
33 #pragma omp parallel for reduction(+:sum)
34   for(i=1;i<n;i++){
35     x = w*(i-0.5);
36     sum=sum+f(x);
37     if( i == 800 )
38       crash(&i);
39   }
40   return sum*w;
41 }
42
43 int main(int argc, char** argv){
44   int n=1000;
45   if(argc>2){
46     fprintf(stderr," Usage : %s [iterations] \n",argv[0
47     return 1;
48   }
49   if(argc==2){
50     n=atoi(argv[1]);
51   }
52   printf(" Pi = %g\n",calc_pi(n));
53   return 0;
54 }
```


Parallel Debugging - Philosophy

- By default, DDT places processes in groups
  - All Group - Includes parent and all related processes
  - Root/Workers Group - Only processes that share the same source code
- Command can act on single process or group
  - stop process , stop group
  - next step process , next step group
  - go process, go group

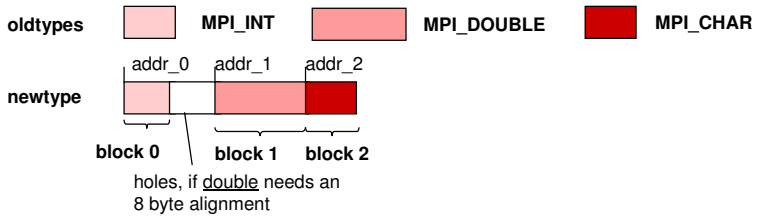
Example (see also MPI-1 standard, p. 79)

```
#include <mpi.h>

   int main(int argc, char **argv)
   {
       struct Partstruct
       {
           int    class;  /* particle class */
           double d;   /* particle coordinates */
           char   b;   /* some additional information */
       };

       struct Partstruct particle;
       int i, dest, rank, size;
       MPI_Status  status;

       /* build datatype describing structure */
       MPI_Datatype Particletype;
       MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
       int blocklen[3] = {1, 1, 1};
       MPI_Aint disp[3];
       int base;

       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       MPI_Comm_size(MPI_COMM_WORLD,&size);
```


Example (see also MPI-1 standard, p. 79)

```
       /* compute displacements of structure components */
       MPI_Address( &particle, disp);
       MPI_Address( &particle.d, disp+1);
       MPI_Address( &particle.b, disp+2);
       base = disp[0];
       for (i = 0; i < 3; i++) disp[i] -= base;

       MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
       MPI_Type_commit( &Particletype);

       if (rank == 0)
       {
           for (i = 1; i < size; i++)
               MPI_Recv( &particle, 1, Particletype, MPI_ANY_SOURCE,
                         MPI_ANY_TAG, MPI_COMM_WORLD, &status );
       }
       else
       {
           MPI_Ssend( &particle, 1, Particletype, 0, 17, MPI_COMM_WORLD );
       }

       MPI_Type_free(&Particletype);
       MPI_Finalize();

       return 0;
   }
```

Example MPI – Struct Datatype




```
int MPI_Type_struct(int count, int *array_of_blocklens,
              MPI_Aint * array_of_displacements,
              MPI_Datatype *array_of_types,
              MPI_Datatype  *newtype)
```

count = 3
array_of_blocklengths = (1, 1, 1)
array_of_displacements = (0, addr_1 – addr_0, addr_2 – addr_0)
array_of_types = (MPI_INT, MPI_DOUBLE, MPI_CHAR)

Example - continued

- `$mpicc -g -O0 -o typestruct typestruct.c`
- The example runs on one cluster but e.g. not on our Xeon EM64T cluster:

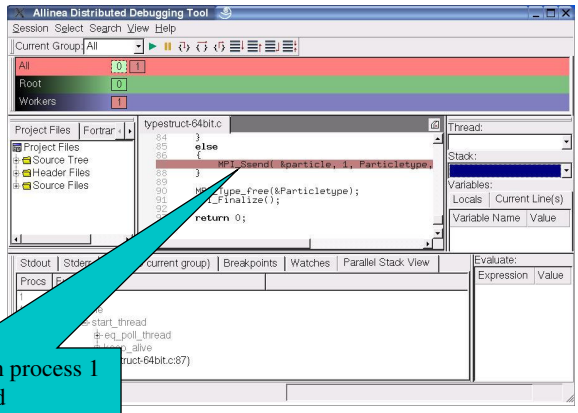
```
$mpirun_ssh -hostfile hostfile -np 2 ./typestruct

sh: line 1: 22828 Segmentation fault
  /usr/bin/env MPIRUN_HOST=noco076.nec
  MPIRUN_PORT=55790
  MPIRUN_PROCESSES='noco076.nec:noco077.nec:'
  MPIRUN_RANK=1 MPIRUN_NPROCS=2 MPIRUN_ID=1040
  MPIRUN_TIMEOUT=60 DISPLAY=cacau1:19.0
  ./typestruct
Rank 1 exited without finalize.
Cleaning up all processes ...
done.
```

How do I find and resolve this bug?

- Start your program with a parallel debugger:

```
$ddt
  ./typestruct
```

and set breakpoints or hit „Run“ and wait where the program terminates



Segmentation violation in process 1
Location: Ssend

How do I find and resolve this bug?

- Something bad happens in MPI_Ssend – tag, count, source/dest, comm arguments seem to match with the MPI_Recv, is there something wrong with our self-defined datatype?

```
if (rank == 0)
    {
        for (i = 1; i < size; i++)
            MPI_Recv( &particle, 1, Particletype, MPI_ANY_SOURCE,
                      MPI_ANY_TAG, MPI_COMM_WORLD, &status );
    }
    else
    {
        MPI_Ssend( &particle, 1, Particletype, 0, 17, MPI_COMM_WORLD );
    }
```

- You **believe** you constructed it correctly – but is that really so?
  - Check all statements (e.g. loops) and variables
  - Yes, that is tiresome but there is not much choice

How do I find and resolve this bug?

Set a breakpoint after calculating base and run to there

Not as expected! disp[0] != base

How do I find and resolve this bug?

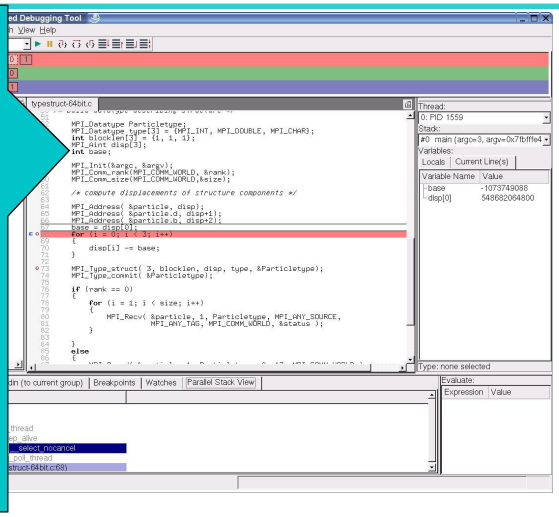
Set a breakpoint after calculating the displacements and run to there

Not as expected! We expect disp[0]==0,...

How do I find and resolve this bug?

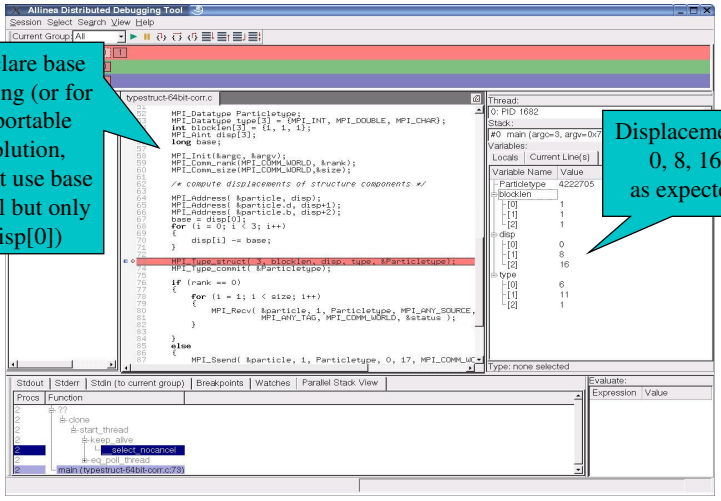
Let‘s have a look at the declarations of base and disp:
int base
MPI_Aint disp[0]

Is it possible that MPI_Aint is not the same as an int? MPI_Aint holds an address on a 64bit machine! Int is 32bit!



How do I find and resolve this bug?

Declare base as long (or for a portable solution, don‘t use base at all but only disp[0])



Displacements 0, 8, 16 as expected

Conclusion

- **Debuggers** help you find out **where** your program crashed.
- **You** have to find out **why.**
- A debugger is a very convenient tool to set breakpoints, examine variables, stacks, etc. **without** adding **printf** statements everywhere in your code (and later on removing them again).
- A debugger is great help to understand your program better.

Ok, I found the bug....

- ... but aren't there any easier ways to find out?
- ...sometimes!
- In this case, have a look at compiler warnings, e.g. with Intel compiler (9.0)

```
$mpicc -Wall -g -O0 -o typestruct typestruct-
  64bit.c
typestruct-64bit.c(67): remark #1682: implicit
  conversion of a 64-bit integral type to a
  smaller integral type (potential portability
  problem)
    base = disp[0];
    ^
```

int

MPI_Aint
long on 64bit
int on 32bit

- e.g. with GNU compiler (3.4.3)

```
$mpicc -Wall -g -O0 -o typestruct typestruct-
  64bit.c
```

No warning!

TOTALVIEW

What is TotalView?

- Parallel debugger
- Source level debugging for C, C++, F77, F90, HPF
- **MPI, OpenMP**, Pthreads, PVM, shmem
- SMPs, MPPs, PVPs, Clusters
- Available on all major Unix Platforms and most Supercomputers
  - available at HLRS on NEC SX-8
- GUI (independent of platform)
- Currently available: Totalview 6.8
- More information:
http://www.hlrs.de/organization/tsc/services/tools/debugger/totalview

TotalView usage

- Compile with -g compiler switch
- command name: **totalview**

On a new process:

```
% totalview myprog -a arguments to myprog
```

To debug MPI programs:

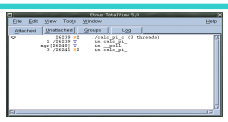
```
% totalview mpirun -a -nprocs 3 myprog
% mpirun -tv -np 3 myprog
```

To debug IBM POE programs:

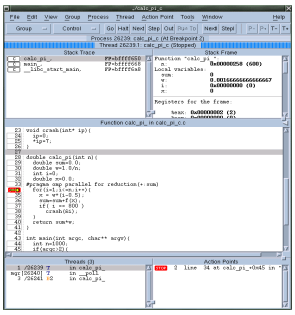
```
% totalview poe -a myprog [args]
```

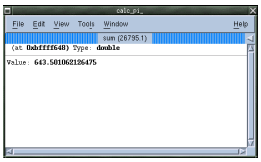

TotalView Look & Feel

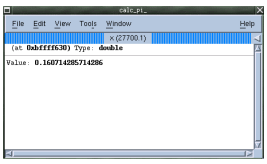
Root Window



Process Window



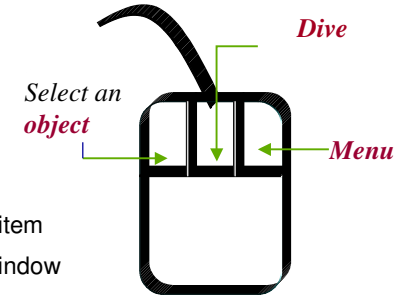




Data Windows

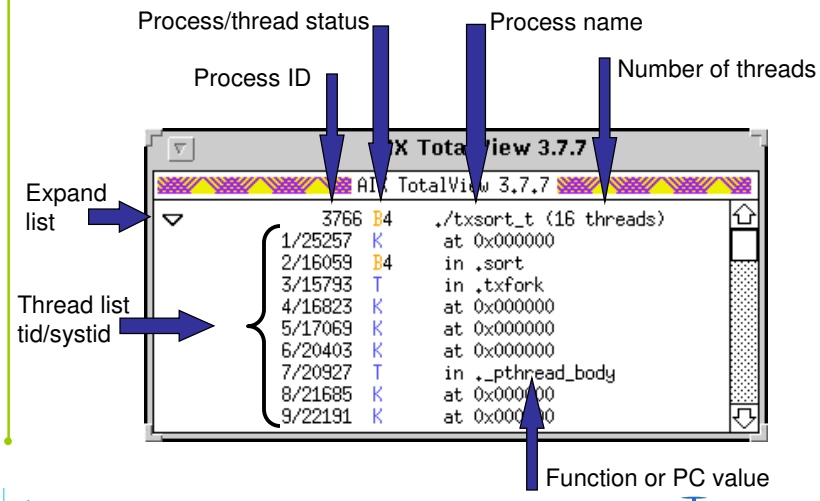
TotalView Mouse Buttons

- **Left** button is **Select**:
  - Chooses an item of interest,
  - Starts editing a item
- **Middle** button is **Dive**:
  - Gets more information about an item
  - **Shift+Dive** forces open a new window
- **Right** button is **Menu**:
  - Raises a menu of actions
  - All menus have a **Help** (^?) entry

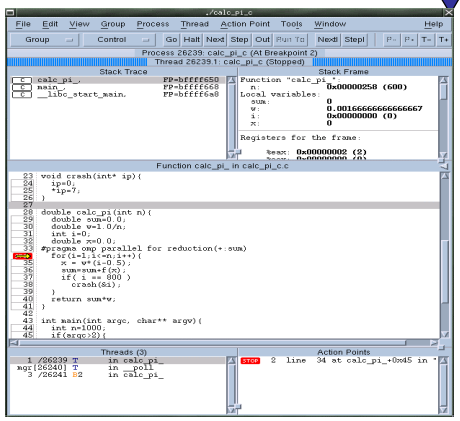


skipped

TotalView Main Window



TotalView Process Window



skipped

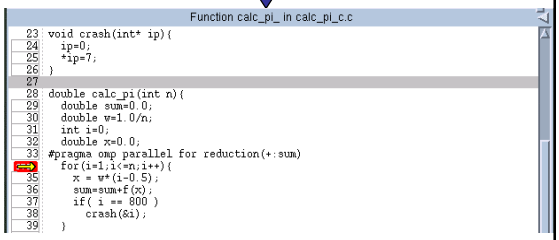
TotalView Source Pane

Gridded box is a possible site for a breakpoint

Current function and source file

Select to set one

Current point of execution



- **Dive** on a source word to get more information
- **Select** a line to use **Run to selection** command
- **Select** or **dive** on a line number to set an action point

Parallel Debugging - Philosophy

- By default, TotalView places processes in groups
  - Program Group - Includes parent and all related processes
  - Share Group - Only processes that share the same source code

- Command can act on single process or share group
  - halt process (h) , halt group (H)
  - next step process (n), next step group (N)
  - go process (g), go group (G)

Parallel Debugging [05]

skipped

TotalView MPI Message Queue Window

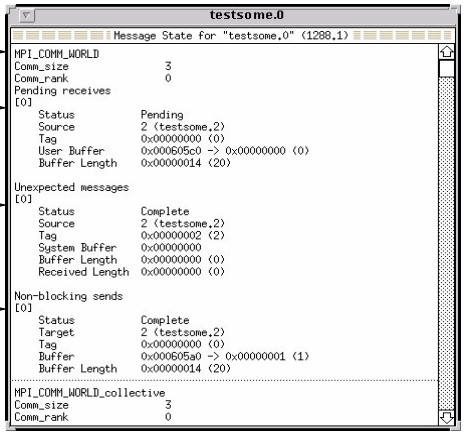
Communicator name and info

Non-blocking receive operations

Unmatched incoming messages

Non-blocking send operations

- Dive on source or target to refocus Process window
- Dive on buffer to see message contents



Testing can only prove the presence of bugs, not their absence.

Edsger Dijkstra

Exercise - Login from PC

Login at PC

Start Win32

- Icon on your screen (or menu Start →Programs→ Communication → X11 → X-Win32 → X-Win32)
- cancel the pop-up-menu

Start first first SSH window with Putty:

- Start -> Programme ->Communications ->ssh,telnet->Putty
- Tunnel -> choose X11-Forwarding

Login to asama.hww.de, user rzvmpi13

- **cd ~/MPI/#nr**

Compilation:

- **mpif90 –g –O0** –o *my_prog my_prog.f*
- **mpicc –g –O0** –o *my_prog my_prog.c*

login slides

Exercise - DDT on asama

- Set your path:
  export PATH=/opt/streamline/ddt/1.10/bin:$PATH
- Start ddt, e.g.
  ddt ./myprog

- Choose the number of processes in the left field and hit "**Run**" to start your debugging session (Note: use "generic" MPI, to change click on the "Advanced" button.)

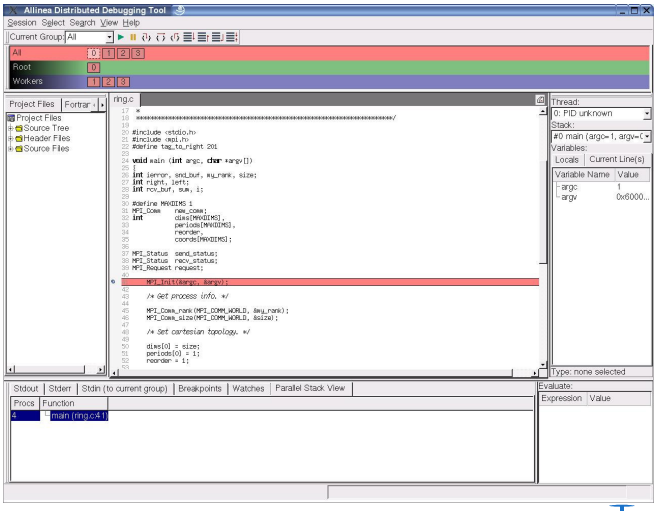
Login & Commands – Page 3

Parallel Debugging Krammer
Slide 87 of 89 Höchstleistungsrechenzentrum Stuttgart

H L R S

login slides

DDT on asama – start your debugging session



Parallel Debugging Krammer
Slide 88 of 89 Höchstleistungsrechenzentrum Stuttgart

H L R S

login slides

Exercise

- Copy the example
  /nfs/home1/rus/rzv/mpi13/ddt/typestruct-64bit.c
  in your working directory and run it with DDT

- Run your other examples with DDT