23rd Summer School on PARALLEL COMPUTING

Debugging & Profiling Techniques

Gian Franco Marras - g.marras@cineca.it

Graziella Ferini - g.ferini@cineca.it

Alessandro Marani - a.marani@cineca.it

SuperComputing Applications and Innovation Department

CINECA

Outline

Outline

Compilers documentation

- **IBM:**
  - http://publib.boulder.ibm.com/infocenter/macxhelp/v6v81/index.jsp
- **Intel:**
  - Fortran: http://software.intel.com/sites/default/files/m/5/0/2/8/5/6335-copts_for.pdf
  - C: http://www2.units.it/divisioneisi/ci/tartaglia/intel/cce/bldaps_cls.pdf
    http://marcbug.scc-dc.com/svn/repository/trunk/Compilers/c_ug_lnx_8.1.pdf
- **PGI:**
  - http://www.pgroup.com/doc/pgiug.pdf
- **GNU:**
  - http://gcc.gnu.org/

Outline

Warnings

Turns on a set of warnings for common programming problems

IBM	Intel	PGI	GNU
-qinfo=all	-warn all (Fortran) -Wall (C/C++)	-Minform=warn	-Wall -Wextra -Werror

- **-Wall** detects:
  - uninitialized variables (-Wuninitialized)
  - unused parameters (-Wunused, -Wunused-{variables|function|value|label})
  - implicit declaration function, for C and C++ (-Wimplicit-function-declaration)
- **-Wextra:** enables some extra warning flags that are not enabled by -Wall. For example, if it is used with -Wall, unused but set variables (-Wunused-but-set-variables) are detected
- **-Werror:** consider warnings to be errors, so that compilation stops.

Floating-point exceptions

- If you know that somewhere in your program, there lurks a catastrophic numerical bug that puts **NaNs** or **Infs** into your results and you want to know where it first happens, the search can be a little frustrating.
- The IEEE standard can help you; these illegal events (divide by zero, underflow or overflow, or invalid operations which cause NaNs) can be made to trigger exceptions, which will stop your code right at the point where it happens; then if you run your code through a debugger, you can find the very line where it happens.
- Add the following flags to obtain a signalling massages (**SIGFPE**) from the application if a fpe is detected at run time (i.e. If after some calculation, a NaN or an inf is generated).

Floating-point exceptions

- **divide-by-zero:** an operation on finite numbers produces infinity as exact answer.
- **overflow:** a result has to be represented as a floating-point number, but has (much) larger absolute value than the largest (finite) floating-point number that is representable.
- **underflow:** a result has to be represented as a floating-point number, but has smaller absolute value than the smallest positive normalized floating-point number (and would lose much accuracy when represented as a denormalized number).
- **inexact:** the rounded result of an operation is not equal to the infinite precision result. It may occur whenever overflow or underflow occurs.
- **invalid:** there is no well-defined result for an operation, as for 0/0 or infinity - infinity or sqrt(-1).

Floating-point exceptions, cont...

IBM	Intel	PGI	GNU
-qflttrap=enable -qflttrap=overflow: underflow:zerodivide: invalid:inexact	-fpe0 (enable) -fpe3 (disable)	-Ktrap=fp -Ktrap=divz,inv, ovf,inexact,unf	-ffpe-trap= invalid,zero, overflow (Fortran only)

- **Intel - Fortran:**
  - **-fpe0:** underflow gives 0.0; abort on other IEEE exceptions.
  - **-fpe3:** produce NaN, signed infinities, and denormal results .
- **Intel - C:** For C code, you have to actually insert a call to **feenableexcept()**, which enables floating point exceptions, and is defined in **fenv.h**
- **GNU:** the flag is for FORTRAN, because in C is default

Simbolic debug

IBM	Intel	PGI	GNU
-g -qfullpath	-g	-gopt (-g)	-g (-ggdb)

- **-g:** Produces symbolic debug information in object file. Prompts the compiler to generate debug information for the source code. You **must** specify this option if you intend to debug your code.
- **-qfullpath:** Causes the full name of all source files to be added to the debug information. This can make it easier for the debugger to find source files.
- **-gopt:** Instructs the compiler to include symbolic debugging information in the object file, and to generate optimized code identical to that generated when -g is not specified.
- **-ggdb:** Produce debugging information for use by GDB. This means to use the most expressive format available (DWARF 2, stabs, or the native format if neither of those are supported), including GDB extensions if at all possible.

Check Bounds

IBM	Intel	PGI	GNU
-C (-qcheck)	-check bounds -WB	-Mbounds	-fbounds-check

- **-C** (**-qcheck** is the long form): Checks each reference to an array element, array section, or character substring to ensure the reference stays within the defined bounds of the entity
- **-check bounds** (**-CB**) : Enables compile-time and run-time checking for array subscript and character sub-string expressions. An error is reported if the expression is outside the dimension of the array or the length of the string.
  - **-WB** Turns a compile-time bounds check into a warning.
- **-fbounds-check**: generate additional code to check that indices used to access arrays are within the declared range.

Uninitialized Variables

IBM	Intel	PGI	GNU
-qinfo=uni	-check uninit (Fortran) -check-uninit (CC++)		-Wuninitialized

- **-qinfo=uni:** Produces informational messages on uninitialized variables.
- **-check-uninit , -check unint:** Enables runtime checking for uninitialized variables. If a variable is read before it is written, a runtime error routine will be called. Runtime checking of undefined variables is only implemented on local, scalar variables. It is not implemented on dynamically allocated variables, extern variables or static variables. It is not implemented on structs, classes, unions or arrays. Note for FORTRAN: Checks for uninitialized scalar variables without the SAVE attribute.
- **-Wuninitialized:** Warn at compiling time if an automatic variable is used without first being initialized.

Strict ISO C & ISO C++

IBM	Intel	PGI	GNU
-qlanglvl=stdc89	-strict-ansi		-ansi -pedantic

- **-qlanglvl=stdc89:** Determines which language standard (or superset, or subset of a standard) to consult for nonconformance. It identifies nonconforming source code and also options that allow such nonconformances. In this case, the compilation conforms to the ANSI C89 standard.
- **-ansi -pedantic:** Use ANSI C, and reject any non-ANSI extensions. These flags help in writing portable programs that will compile on other systems.

Outline

Optimization – IBM

IBM	Intel	PGI	GNU
-O0 -O2 -O3 -O4 -O5	-O0 -01 -O2 -O3	-O0 -O2 -O3	-O0 -O1 -O2 -O3

- **-O0 :** all optimizations are disabled. Usefull for debbugging, togheter with **-g**.
- **-O2 :** optimizations that offer improved performance without an unreasonable increase in time or storage that is required for compilation.
- **-O3 :** memory-intensive optimizations, the semantics of the program can be altered. Use **-qstrict** to avoid incorrect results.
- **-O4 :** aggressive optimizations (**-qarch=auto**, **-qhot**, **-qipa**, **-qtune=auto**, **-qcache=auto**, **-qsimd=auto**).
- **-O5 :** as **-O4** with **-qipa=level=2** also.

Optimization – Intel

IBM	Intel	PGI	GNU
-O0 -O2 -O3 -O4 -O5	-O0 -01 -O2 -O3	-O0 -O2 -O3	-O0 -O1 -O2 -O3

- **-O0 :** all optimizations are disabled. Usefull for debbugging, togheter with **-g**.
- **-O1 :** enables optimizations for speed and disables some optimizations that increase code size and affect speed.
- **-O2 :** enables optimizations for speed. This is generally recommended optimization level
- **-O3 :** enables **-O2** plus more aggressive optimizations.

Optimization – Intel -O3

IBM	Intel	PGI	GNU
-O0 -O2 -O3 -O4 -O5	-O0 -01 -O2 -O3	-O0 -O2 -O3	-O0 -O1 -O2 -O3

- Automatic vectorization (use of packed SIMD instructions)
- Loop interchange (for more efficient memory access)
- Loop unrolling (more instruction level parallelism)
- Prefetching (for patterns not recognized by h/w prefetcher)
- Cache blocking (for more reuse of data in cache)
- Loop peeling (allow for misalignment)
- Loop versioning (for loop count; data alignment; runtime dependency tests)
- Memcpy recognition (call Intel's fast memcpy, memset)
- Loop splitting (facilitate vectorization)
- Loop fusion (more efficient vectorization)
- Scalar replacement (reduce array accesses by scalar temps)
- Loop rerolling (enable vectorization)
- Loop reversal (handle dependencies)

Optimization – PGI

IBM	Intel	PGI	GNU
-O0 -O2 -O3 -O4 -O5	-O0 -01 -O2 -O3	-O0 -O2 -O3	-O0 -O1 -O2 -O3

- **-O0 :** no optimization. A basic block is generated for each language statement.
- **-O1 :** local optimization. Scheduling of basic blocks is performed. Register allocation is performed.
- **-O2 :** global optimization. This level performs all **-O1** local optimization as well as **-O2** global optimization. If optimization is specified on the command line without a level, **-O2** is the default.
- **-O3 :** specifies aggressive global optimization. This level performs **-O1** and **-O2** optimizations and enables more aggressive hoisting and scalar replacement optimizations that may or may not be profitable.
- **-O4 :** performs all **-O1**, **-O2** and **-O3** optimizations and enables hoisting of guarded invariant floating point expressions.

Optimization – GNU

IBM	Intel	PGI	GNU
-O0 -O2 -O3 -O4 -O5	-O0 -01 -O2 -O3	-O0 -O2 -O3	-O0 -O1 -O2 -O3

- **-O0 :** no optimization. A basic block is generated for each language statement. If optimization is specified on the command line without a level, **-O0** is the default.
- **-O1 :** Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
- **-O2 :** Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to **-O0**, this option increases both compilation time and the performance of the generated code.
- **-O3 :** Optimize yet more. All optimizations specified by **-O2** are turned on togheter with **-finline-functions**, **-funswitch-loops**, **-fpredictive-commoning**, **-fgcse-after-reload**, **-ftree-vectorize** and **-fipa-cp-clone**.

Architecture

Use these flags to target your program to instruct the compiler to generate code specific to a particular architecture.
This allows the compiler to take advantage of machine-specific instructions that can improve performance.

IBM	Intel	PGI	GNU
-qarch=<arch> -qtune=<arch> -qcache=<subop-list>	-mtune=<arch>	-tp<arch>	-march=<arch> -mtune=<arch> -mcpu=<cpu>

- IBM: By default, the -qarch setting produces code using only instructions common to all supported architectures, with resultant settings of -qtune and -qcache that are relatively general.
  - **-qarch={auto|pwr6|qp|...}**
  - **-qtune={auto|pwr6|qp|...}**
  - **-qcache={auto|level=...|line=...|size=...|...}**
- Intel: **-mtune={generic|itanium2-p9000|...}**

Architecture

IBM	Intel	PGI	GNU
-qarch=<arch> -qtune=<arch> -qcache=<subop-list>	-mtune=<arch>	-tp<arch>	-march=<arch> -mtune=<arch> -mcpu=<cpu>

- PGI : **-tp{athlon|k8-64|nehalem|p7-64|x84}**
- GNU
  - **-march=<architecture-type> :** This specifies the name of the target ARM (Adv. RISC) architecture.
  - **-mtune=<architecture-type> :** Tune to architecture-type everything applicable about the generated code, except for the ABI and the set of available instructions.
  - **-mcpu=<cpu-type> :** You can specify either the EV style name or the corresponding chip number.

Loop optimization 1/3

The performance of certain classes of loops may be improved through vectorization or unrolling options.

- **Vectorization** transforms loops to improve memory access performance and make use of packed SSE instructions which perform the same operation on multiple data items concurrently.
- **Unrolling** replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions.

Loop optimization 2/3

IBM	Intel	PGI	GNU
-qcache=[=suboptions] -qhot=[=suboptions] -qunroll	-ax{SSE4.2|AVX} -x{SSE4.2|AVX} -unroll=n	-Mvect=sse -Munroll=c:n	-msse4.2 -mavx

- **-qcache{auto|assoc=n|cost=cycles|level=level|line=bytes|...} :** Specifies the cache configuration for a specific execution machine. The compiler uses this information to tune program performance, especially for loop operations that can be structured (or blocked) to process only the amount of data that can fit into the data cache.
- **-qhot{auto|assoc=n|line=n(byte)|size=n(Kbyte)} :** Determines whether to perform high-order transformations on loops and array language during optimization and whether to pad array dimensions and data objects to avoid cache misses.
- **-qunroll:** Specifies whether unrolling DO loops is allowed in a program. Unrolling is allowed on outer and inner DO loops.

Loop optimization 3/3

- **-ax{SSE4.2|AVX} :** directs the compiler to generate processor specific code if there is a performance benefit.
- **-x{SSE4.2|AVX} :** directs the compiler to generate specialized and optimized code for the processor that execute your program.
- **-unroll=n :** Tells the compiler the maximum number of times to unroll loops.
- **-Mvect=sse :** Generates SSE instructions
- **-Munroll=c:n :** Unrolls loops, executing multiple instances of the loop every "n" iteration (if n=1, it is done during each iteration.)

Interprocedural Analisys (IPA) 1/2

- Allows use of information across function call boundaries to perform optimizations that would otherwise be unavailable.
- For example, if the actual argument to a function is in fact a constant in the caller, it may be possible to propagate that constant into the call and perform optimizations that are not valid if the dummy argument is treated as a variable.
- A wide range of optimizations are enabled or improved by using IPA, including but not limited to data alignment optimizations, argument removal, constant propagation, pointer disambiguation, pure function detection, F90/F95 array shape propagation, data placement, vestigial function removal, automatic function inlining, inlining of functions from pre-compiled libraries, and interprocedural optimization of functions from pre-compiled libraries.

Interprocedural Analisys (IPA) 2/2

IBM	Intel	PGI	GNU
-qipa=level={0|1|2}	-ip -ipo	-Mipa	-fipa-[<sub-opt>]

- **-qipa :** Determines the amount of IPA analysis and optimization performed, where <level> can be equal to:
  - 0 : Performs only minimal interprocedural analysis and optimization.
  - 1 : Turns on inlining, limited alias analysis, and limited call-site tailoring.
  - 2 : Full interprocedural data flow and alias analysis. To generate data reorganization information, specify the optimization level -qipa=level=2 or -O5 together with -qreport.
- **-ip :** enables additional interprocedural optimizations for single file compilation.
- **-ipo:** enables interprocedural optimizations between files.

Complex numbers

IBM	Intel	PGI	GNU
	-complex-limited-range		

- **-complex-limited-range:** use the highest performance formulations of complex arithmetic operations

Fuctions – ESSL

- **IBM Engineering and Scientific Subroutine Library** (**ESSL**) is a state-of-the-art collection of high-performance subroutines providing a wide range of mathematical functions for many different scientific and engineering applications.
- Its primary characteristics are performance, functional capability, and usability.
- ESSL is provided as run-time libraries that run on the servers and processors.
- ESSL can be used with Fortran, C, and C++ programs operating under the AIX and Linux operating systems.
- The mathematical subroutines, in nine computational areas, are tuned for performance. The computational areas are:
  - Linear Algebra Subprograms
  - Matrix Operations
  - Linear Algebraic Equations
  - Eigensystem Analysis
  - Fourier Transforms, Convolutions and Correlations, and Related Computations
  - Sorting and Searching
  - Interpolation
  - Numerical Quadrature
  - Random Number Generation

Fuctions – Parallel ESSL

- **Parallel ESSL** is a scalable mathematical subroutine library that supports parallel processing applications on clusters of processor nodes optionally connected by a high-performance switch.
- PESSL supports the Single Program Multiple Data (SPMD) programming model using the Message Passing Interface (MPI) library.
- PESSL provides subroutines in the following computational areas:
  - Parallel Basic Linear Algebra Subprograms (PBLAS)
  - Linear Algebraic Equations
  - Eigensystem Analysis and Singular Value Analysis
  - Fourier Transforms
  - Random Number Generation
- For communication, PESSL includes the Basic Linear Algebra Communications Subprograms (BLACS), which use MPI.
- For computations, PESSL uses the ESSL subroutines.
- The PESSL subroutines can be called from 32-bit and 64-bit environment application programs written in Fortran, C, and C++.

Fuctions – MKL

- **Intel Math Kernel Library** (**Intel MKL**) is a computing math library of highly optimized, extensively threaded routines for applications that require maximum performance.
- Intel MKL provides comprehensive functionality support in these major areas of computation, for example:
  - BLAS and LAPACK linear algebra routines, offering vector, vector-matrix, and matrixmatrix operations.
  - ScaLAPACK distributed processing linear algebra routines for Linux* and Windows* operating systems, as well as the Basic Linear Algebra Communications Subprograms (BLACS) and the Parallel Basic Linear Algebra Subprograms (PBLAS).
  - Fast Fourier transform (FFT) functions in one, two, or three dimensions with support for mixed radices (not limited to sizes that are powers of 2), as well as distributed versions of these functions provided for use on clusters of the Linux* and Windows* operating systems.
  - Vector Math Library (VML) routines for optimized mathematical operations on vectors.
  - Vector Statistical Library (VSL) routines, which offer high-performance vectorized random number generators (RNG) for several probability distributions, convolution and correlation routines, and summary statistics functions.
- Intel MKL is optimized for the latest Intel processors, including processors with multiple cores. Intel MKL also performs well on non-Intel processors.

Fuctions

IBM	Intel	PGI	GNU
-lessl, -lesslbg	-lm (MKL)	VML (MKL)	MKL

- **-lesslbg :** Engineering and Scientific Subroutine Library (essl) optimized for BG/Q architecture
- **-lm :** for using Math Kernel Library (MKL)
- **VML (MKL):** MKL and VML can be linked using a pgi compiler
- **MKL :** MKL can be linked using a gnu compiler

Report

IBM	Intel	PGI	GNU
-qlistopt, -qreport, -qsource	-vec_report{0|1|2|3|4|5}	-Minfo	

- **-vec_report :** directs the compiler to generate the vectorization reports with different level of information as follows:
  - **-vec_report0 :** no diagnostic information is displayed
  - **-vec_report1 :** display diagnostics indicating which loops have been successfully vectorized (default) .
  - **-vec_report2 :** same as -vec_report1, plus diagnostics indicating why some loops have not successfully vectorized .
  - **-vec_report3 :** same as -vec_report2, plus additional information about any proven or assumed dependences
  - **-vec_report4 :** indicate non-vectorized loops
  - **-vec_report5 :** indicate non-vectorized loops and the reason why they were not vectorized.

Report

IBM	Intel	PGI	GNU
-qlistopt, -qreport, -qsource	-vec_report{0|1|2|3|4|5}	-Minfo	

- **-qlistopt, -qreport, -qsource :** generate a file.lst containing informations on the optimization performed by the compiler, tougheter with the source code.
- **-Minfo :** display compile-time optimization listings. When this option is used, the PGI compilers issue informational messages to stderr as compilation proceeds. From these messages, you can determine which loops are optimized using unrolling, SSE instructions, vectorization, parallelization, interprocedural optimizations and various miscellaneous optimizations. You can also see where and whether functions are inlined.

Outline

Outline

Timing

time:

The time command runs the specified program command with the given arguments;

Time writes a message to standard error giving timing statistics about this program run; These statistics consist of:

- the elapsed real time between invocation and termination;
- the user CPU time;
- the system CPU time.

```
bash-3.2$ time program.x
real     0m0.701s
user     0m0.000s
sys      0m0.002s
```

Timing

ETIME function (Fortran):

The RESULT = ETIME(TARRAY) function returns the number of seconds of runtime since the start of the process's execution as the function value.

- TARRAY(1) returns the user time;
- TARRAY(2) returns the system time;
- RESULT is equal to TARRAY(1) + TARRAY(2).

Example:

```
real, dimension(2) :: tarray1, tarray2
real :: t1,t2
t1=ETIME(tarray1)
KERNEL CODE!!!
t2=ETIME(tarray2)
write(6,*) "time =",t2-t1
write(6,*) "USER TIME",tarray2(1)-tarray1(1)
write(6,*) "SYSTEM TIME",tarray2(2)-tarray1(2)
```

Timing

SYSTEM_CLOCK routine (Fortran):

SYSTEM_CLOCK([COUNT, COUNT_RATE, COUNT_MAX])

- COUNT is a processor clock since an unspecified time in the past;
- COUNT_RATE determines the number of clock ticks per second;
- COUNT_MAX is the max value of the processor clock.

Example:

```
PROGRAM test_system_clock
  integer t1, t2, count_rate, count_max
  CALL SYSTEM_CLOCK(t1, count_rate, count_max)
  KERNEL CODE!!!
  CALL SYSTEM_CLOCK(t2, count_rate, count_max)
  write(*,*) "real time", real(t2-t1)/real(count_rate)
END PROGRAM
```

Timing

gettimeofday routine (C):

int gettimeofday(timeval *tp, NULL)) gets the time of day.

Example:

```
double mycclock()
{
  struct timeval tmp;
  double sec;
  gettimeofday( &tmp, (struct timezone *)0 );
  sec = tmp.tv_sec + ((double)tmp.tv_usec)/1000000.0;
  return sec;
}
  ...
  start = mycclock();
  KERNEL CODE!!!
  finish = cclock();
  printf("TOT = %lf",finish-start);
```

Outline

GNU Profiler – Gprof

GNU Profiler – Gprof

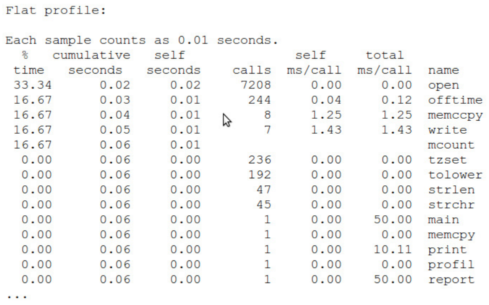
The GNU profiler **gprof** can be used to determine which parts of a program are taking most of the execution time.

gprof can produce several different output styles:

- **Flat Profile**: The flat profile shows how much time was spent executing directly in each function.
- **Call Graph**: The call graph shows which functions called which others, and how much time each function used when its subroutine calls are included.

Gprof – Flat profile

The **flat profile** shows the total amount of time your program spent executing each function.
Note that if a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called.

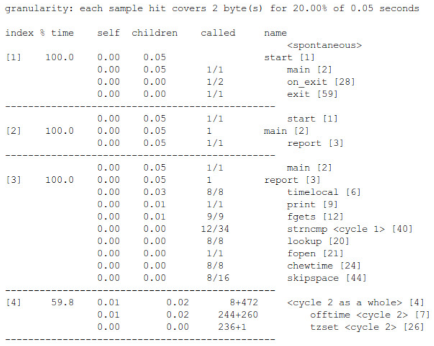


```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

Gprof – Call Graph

The **call graph** shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.

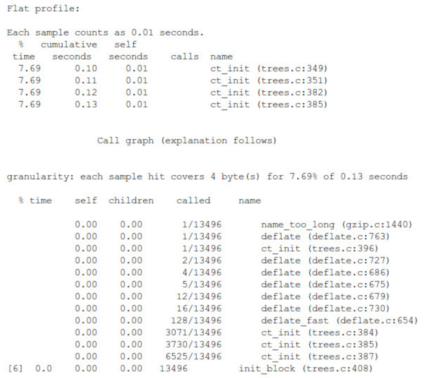


```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     100.0    0.00    0.05                 start [1]
                 0.00    0.05       1/1           main [2]
                 0.00    0.00       1/2           on_exit [28]
                 0.00    0.00       1/1           exit [59]
-----------------------------------------------
                 0.00    0.05       1/1           start [1]
[2]     100.0    0.00    0.05       1         main [2]
                 0.00    0.05       1/1           report [3]
-----------------------------------------------
                 0.00    0.05       1/1           main [2]
[3]     100.0    0.00    0.05       1         report [3]
                 0.00    0.03       8/8           timelocal [6]
                 0.00    0.01       1/1           print [9]
                 0.00    0.01       9/9           fgets [12]
                 0.00    0.00      12/34          strncmp <cycle 1> [40]
                 0.00    0.00       8/8           lookup [20]
                 0.00    0.00       1/1           fopen [21]
                 0.00    0.00       8/8           chewtime [24]
                 0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]      59.8    0.01        0.02        8+472        <cycle 2 as a whole> [4]
                 0.01        0.02      244+260            offtime <cycle 2> [7]
                 0.00        0.00      236+1              tzset <cycle 2> [26]
-----------------------------------------------
```

Gprof – Listing by line

The **-l** option enables **line-by-line** profiling, which causes histogram hits to be charged to individual source code lines, instead of functions.
This feature only works with programs compiled by older versions of the "gcc" compiler. Newer versions of "gcc" are designed to work with the "gcov" tool instead.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds    calls  name
  7.69      0.10     0.01           ct_init (trees.c:349)
  7.69      0.11     0.01           ct_init (trees.c:351)
  7.69      0.12     0.01           ct_init (trees.c:382)
  7.69      0.13     0.01           ct_init (trees.c:385)


                Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 7.69% of 0.13 seconds

  % time    self  children    called     name

            0.00    0.00       1/13496       name_too_long (gzip.c:1440)
            0.00    0.00       1/13496       deflate (deflate.c:763)
            0.00    0.00       1/13496       ct_init (trees.c:396)
            0.00    0.00       2/13496       deflate (deflate.c:727)
            0.00    0.00       4/13496       deflate (deflate.c:686)
            0.00    0.00       5/13496       deflate (deflate.c:675)
            0.00    0.00      12/13496       deflate (deflate.c:679)
            0.00    0.00      16/13496       deflate (deflate.c:730)
            0.00    0.00     128/13496       deflate_fast (deflate.c:654)
            0.00    0.00    3071/13496       ct_init (trees.c:384)
            0.00    0.00    3730/13496       ct_init (trees.c:385)
            0.00    0.00    6525/13496       ct_init (trees.c:387)
[6]   0.0   0.00    0.00   13496         init_block (trees.c:408)
```

Compiling, Running, Output files

- Compile and link the program with options: **-g -pg -qfullpath**
- Profiling files in execution directory
  - **gmon.out.<MPI Rank>** = binary files, not readable
  - The number of files depends on environment variable
    - 1 Profiling File / Process: The default setting is to generate gmon.out files only for profiling data collected on ranks 0 - 31.
    - **BG_GMON_RANK_SUBSET=N** – Only generate the gmon.out file for rank N.
    - **BG_GMON_RANK_SUBSET=N:M** – Generate gmon.out files for all ranks from N to M.
    - **BG_GMON_RANK_SUBSET=N:M:S** – Generate gmon.out files for all ranks from N to M. Skip S; 0:16:8 generates gmon.out.0, gmon.out.8, gmon.out.16
- Output files interpretation
  - **gprof** <**Binary**> **gmon.out.**<**MPI Rank**> > **gprof.out.**<**MPI Rank**>
  - Graphical utility, part of HPC Toolkit: Xprof

Using GNU profiling – Threads

- The base GNU toolchain does not provide support for profiling on threads
- Profiling threads
    - **BG_GMON_START_THREAD_TIMERS**
        - Set this environment variable to "all" to enable the SIGPROF timer on all threads created with the pthread_create() function.
        - "nocomm" to enable the SIGPROF timer on all threads except the extra threads that are created to support MPI.
    - Add a call to the **gmon_start_all_thread_timers()** function to the program, from the main thread
    - Add a call to the **gmon_thread_timer(int start)** function from the thread to be profiled: 1 to start, 0 to stop

Outline

Scalasca

- SCalable performance Analysis of LArge SCale Applications
- Developed by Juelich Supercomputer Center
- Toolset for performance analysis of parallel applications on a large scale
- Manages MPI, OpenMP, MPI+OpenMP programs
- Successfully tested on different platforms (Cray XT5/6 & XE6/XK6, Fujitsu FX10 & K computer, IBM Blue Gene/Q, IBM SP & BladeCenter clusters (AIX/Linux), NEC SX-9, SGI Altix, various Linux/Intel (x86/x64) clusters, including Intel Xeon Phi)
- Latest release 2.0; available: 1.4.2 on FERMI, 1.4.1 on PLX
- www.scalasca.org
- http://www2.fz-juelich.de/jsc/datapool/scalasca/UserGuide.pdf

scalasca

Scalasca

Scalasca can perform two kinds of analysis:

- **runtime summarization**, with aggregate performance metrics for individual function call paths (overview of the performance behavior)
- **tracing** (disabled by default): each process generates a trace file containing records for all its process-local events. Such files, after program termination, are reloaded back into main memory and analyzed in parallel with the same number of CPUs used for the target application itself. Trace analysis is useful to identify wait states that cannot be detected by runtime summarization.

Scalasca – How it works 1/3

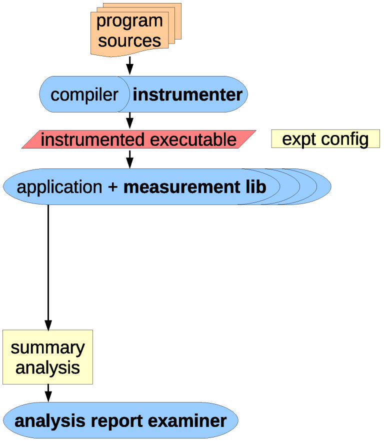
Three main steps:

- instrumentation: calls to the Scalasca measurement system are inserted into the application code, either automatically (default), semi-automatically or manually.
- execution measurement collection and analysis
- analysis report examination

Scalasca – How it works 2/3

Runtime summarization

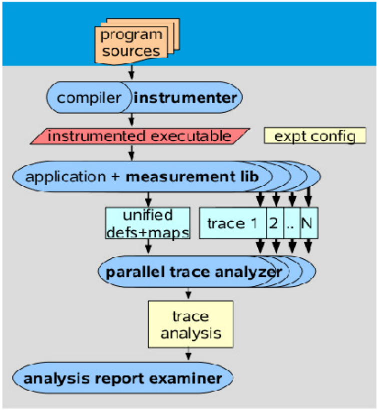
- Measurement library manages threads and events produced by instrumentation
- Measurements are summarized during execution
- Final collection of the results (unified report)



Scalasca – How it works 3/3

Event tracing

- During the measurement, there is a buffer for each thread/process
- Buffers are flushed to files at finalization
- Follow-up analysis replays events and produces extended analysis report



Scalasca – How to use

- prepare application objects and executable for measurement (automatic instrumentation)

  scalasca -instrument <compile-or-link-command>

Original command:	Scalasca instrumentation command:
mpicc -c foo.c	**scalasca -instrument mpicc -c foo.c**
mpif90 -o bar bar.f90	**skin mpif90 -o bar bar.f90**

- run application under control of measurement system

  scalasca -analyze <application-launch-command>

Original command:	Scalasca instrumentation command:
mpirun -n 4 ./myexe	**scan mpirun -n 4 ./myexe**

- post-process and explore measurement analysis report

  scalasca -examine (-s) <experiment-archive|report>

Scalasca output: Archive files 1/2

Scalasca creates an experiment archive directory:

- Pure MPI:
  scalasca -analyze mpirun –np 256 –exe <my_exe>
  ==> **epik_<myexe>_256_sum**
- MPI + OpenMP:
  scalasca -analyze mpirun –np 256 -envs OMP_NUM_THREADS=4 –exe <my_exe>
  ==> **epik_<myexe>_256x4_sum**

Archive with log files 2/2

In each epik archive there are the following files:

epik.conf	Measurement configuration when the experiment was collected
epik.log	Output of the instrumented program and measurement system
epik.path	Callpath-tree recorded by the measurement system
epitome.cube	Intermediate analysis report of the runtime summarization system
summary.cube[.gz]	Post-processed analysis report of runtime summarization

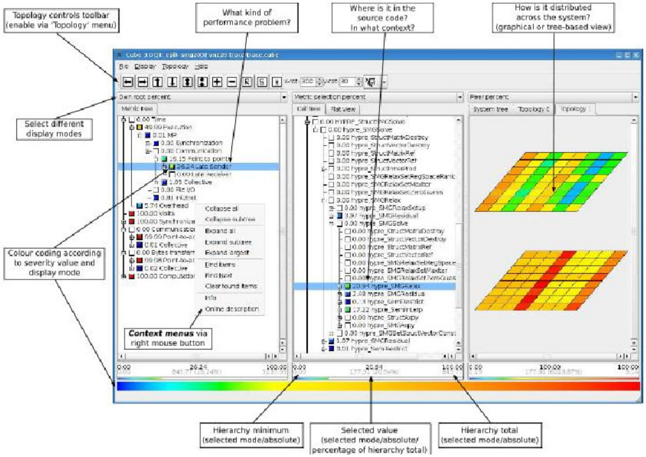
Report scoring as textual output

The -s option of square [scalasca -examine] skips display and produces textual score report (epik.score)

Region/call path classification

- **MPI** (pure MPI library functions)
- **OMP** (pure OpenMP functions/regions)
- **USR** (user-level source computation)
- **COM** (combined USR+OpenMP/MPI)
- **ANY/ALL** (aggregate of all region types)

Examination by GUI


Topology controls toolbar (enable via 'Topology' menu)
What kind of performance problem?
Where is it in the source code? In what context?
How is it distributed across the system? (graphical or tree-based view)
Select different display modes
Colour coding according to severity value and display mode
Context menus via right mouse button
Hierarchy minimum (selected mode/absolute)
Selected value (selected mode/absolute/ percentage of hierarchy total)
Hierarchy total (selected mode/absolute)
Metric tree
Call tree
Flat view
System tree
Topology 0
Topology 1
Collapse all
Collapse subtree
Expand all
Expand subtree
Expand largest
Find items
Find Next
Clear found items
Info
Online description

Display of results

Results are displayed using three coupled tree browser showing:

- Metrics (i.e. Performance properties/problems)
- Call-tree or flat region profile
- System location

Metrics 1/2

Time	Total CPU allocation time
Visits	Number of times a routine/region was executed
Synchronizations	Total number of MPI synchronization operations that were executed
Communications	The total number of MPI communication operations, excluding calls transferring no data (which are considered Synchronizations)
Bytes transferred	The total number of bytes that were sent and received in MPI communication operations. It depends on the MPI internal implementation.

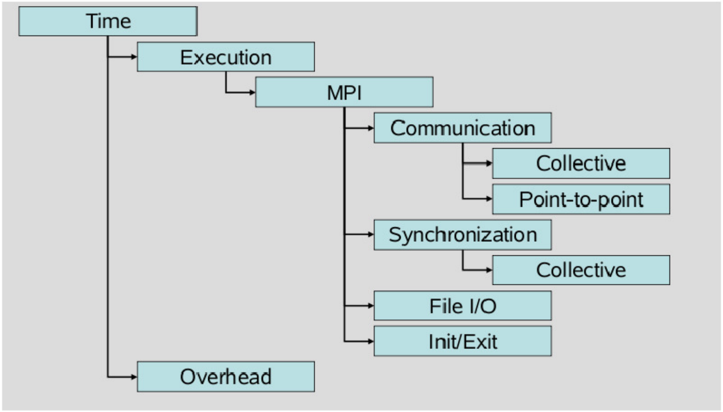
http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_patterns-1.4.html

Metrics 2/2

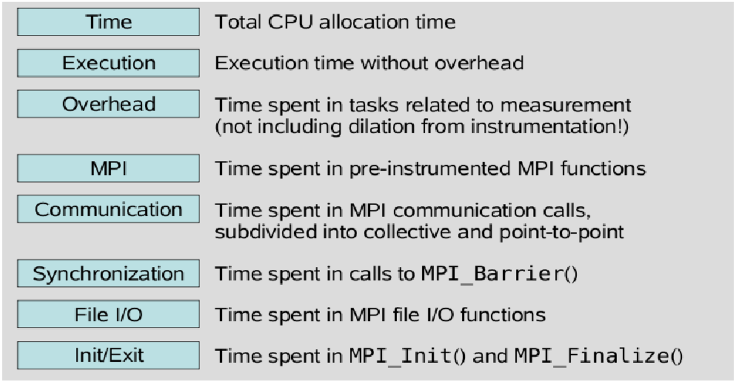
MPI file operations	Number of MPI file operations of any type.
MPI file bytes transferred	Number of bytes read or written in MPI file operations of any type.
Computational imbalance	This simple heuristic allows to identify computational load imbalances and is calculated for each (call-path, process/thread) pair.

http://www2.fz-juelich.de/jsc/datapool/scalasca/scalasca_patterns-1.4.html

Metrics – Time, pure MPI code 1/2

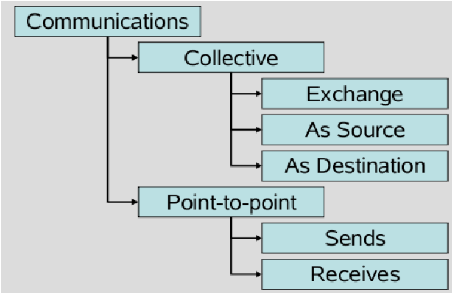


Metrics – Time, pure MPI code 2/2



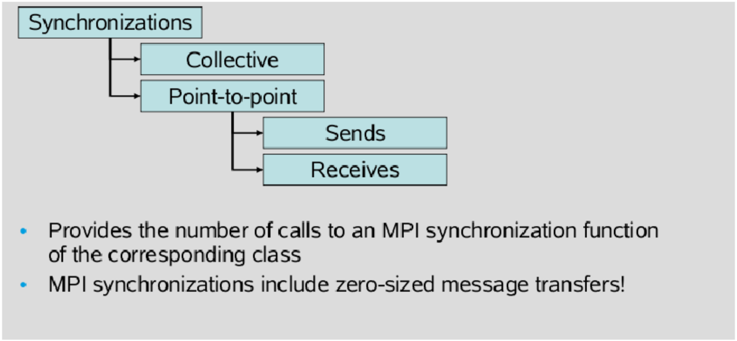
Time	Total CPU allocation time
Execution	Execution time without overhead
Overhead	Time spent in tasks related to measurement (not including dilation from instrumentation!)
MPI	Time spent in pre-instrumented MPI functions
Communication	Time spent in MPI communication calls, subdivided into collective and point-to-point
Synchronization	Time spent in calls to `MPI_Barrier()`
File I/O	Time spent in MPI file I/O functions
Init/Exit	Time spent in `MPI_Init()` and `MPI_Finalize()`

Pure MPI code – Communications



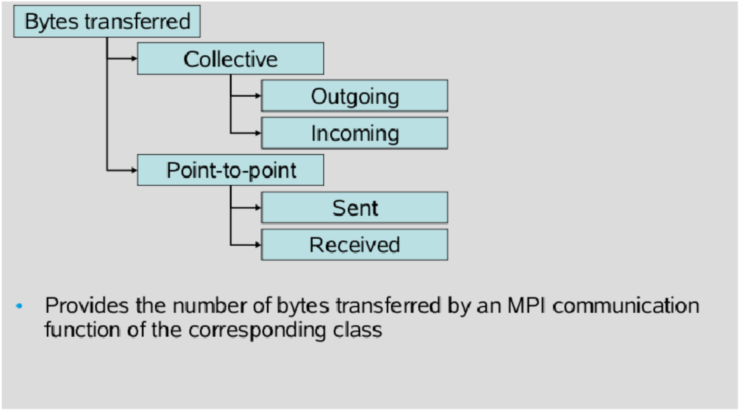
- Provides the number of calls to an MPI communication function of the corresponding class
- Zero-sized message transfers are considered *synchronization*!

Pure MPI code – Synchronizations



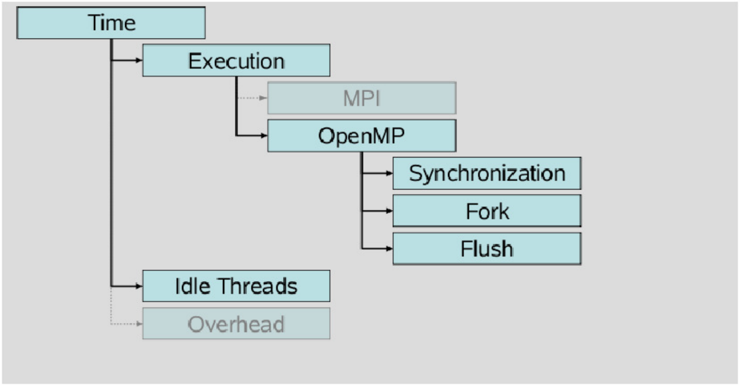
- Provides the number of calls to an MPI synchronization function of the corresponding class
- MPI synchronizations include zero-sized message transfers!

Pure MPI code – Bytes transferred

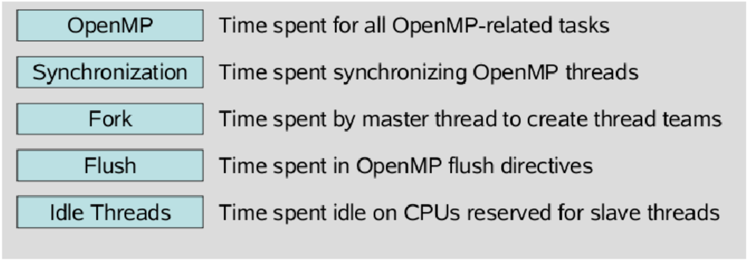


- Provides the number of bytes transferred by an MPI communication function of the corresponding class

Metrics – Time, MPI-OpenMP code



Time, OpenMP part of the code



OpenMP	Time spent for all OpenMP-related tasks
Synchronization	Time spent synchronizing OpenMP threads
Fork	Time spent by master thread to create thread teams
Flush	Time spent in OpenMP flush directives
Idle Threads	Time spent idle on CPUs reserved for slave threads

Hardware counters measurement

- Hardware counter measurement is disabled by default
- Can be enabled using:
  - the environment variable EPK_METRICS in the jobscript (scalasca -analyze)
  - scalasca -analyze -m <metric_name> <application-launch-command>
- Set EPK_METRICS to a colon-separated list of counter names, or a predefined platform-specific group
- Metric names can be chosen from the list contained in file $SCALASCA_HOME/doc/METRICS.SPEC

Manual source-code instrumentation

- Region or phase annotations manually inserted in source file can augmented or substiture automatic instrumentation, and can improve the structure of analysis reports to make them more readly comprehensible

- These annotations can be used to mark any sequence or block of statements, such as functions, phases, loop nests, etc., and can be nested, provided that every enter has matching exit

- If automatic compiler instrumentation is not used, it is typically desiderable to manually instrument at least the **main function/program and perhaps its major phases (e.g. Initialization, core/body, finalization).**

User instrumentation API – C/C++

```
#include "epik_user.h"
...
void foo(){
  ... ...
  EPIK_FUNC_START();
  ... ... // executable statements
  if(...){
    EPIK_FUNC_END();
    return;
  } else {
    EPIK_USER_REG (r_name, "region");
    EPIK_USER_START (r_name);
    ... ...
    ... ...
    EPIK_USER_END (r_name);
  }
  ... ... // executable statements;
  EPIK_FUNC_END();
  return;
}
```

User instrumentation API – Fortran

```
#include "epik_user.inc"
...
subroutine bar()
  EPIK_FUNC_REG("bar")
  EPIK_USER_REG (r_name, "region")
  ... ... ! local declarations
  EPIK_FUNC_START();
  ... ... ! executable statements
  if(...) then
    EPIK_FUNC_END()
    return
  else
    EPIK_USER_START (r_name)
    ... ...
    EPIK_USER_END (r_name)
  endif
  ... ... ! executable statements
  EPIK_FUNC_END()
  return
end subroutine bar
```

Outline

Valgrind

- Open Source Software, available on Linux for x86 and PowerPc processors.
- Interprets the object code, not needed to modify object files or executable, non require special compiler flags, recompiling, or relinking the program.
- Command is simply added at the shell command line.
- No program source is required (black-box analysis).

www.valgrind.org

Valgrind:tools

- Memcheck: a memory checker.
- Callgrind: a runtime profiler.
- Cachegrind: a cache profiler.
- Helgrind: find race conditions.
- Massif: a memory profiler.

Why should use I use Valgrind?

- Valgrind will tell you about tough to find bugs.
- Valgrind is very through.
- You may be tempted to think that Valgrind is too picky, since your program may seem to work even when valgrind complains. It is users' experience that fixing ALL Valgrind complaints will save you time in the long run.

But...
Valgrind is kind-of like a virtual x86 interpeter. So your program will run 10 to 30 times slower than normal.
Valgrind won't check static arrays.

Why should use I use Valgrind?

- Valgrind will tell you about tough to find bugs.
- Valgrind is very through.
- You may be tempted to think that Valgrind is too picky, since your program may seem to work even when valgrind complains. It is users' experience that fixing ALL Valgrind complaints will save you time in the long run.

But...
Valgrind is kind-of like a virtual x86 interpeter. So your program will run 10 to 30 times slower than normal.
Valgrind won't check static arrays.

Use of uninitialized memory:test1.c

- Local Variables that have not been initialized.
- The contents of malloc's blocks, before writing there.

```
#include <stdlib.h>
int main()
{
      int p,t,b[10];
      if(p==5)       ERROR
          t=p+1;
          b[p]=100; ERROR
      return 0;
}
```

Use of uninitialized memory:test1.c

- Local Variables that have not been initialized.
- The contents of malloc's blocks, before writing there.

```
#include <stdlib.h>
int main()
{
      int p,t,b[10];
      if (p==5)      ERROR
          t=p+1;
          b[p]=100;  ERROR
      return 0;
}
```

Use of uninitialized memory:test1.c

- Local Variables that have not been initialized.
- The contents of malloc's blocks, before writing there.

```
#include <stdlib.h>
int main()
{
     int p,t,b[10];
     if (p==5)     ERROR
         t=p+1;
         b[p]=100; ERROR
     return 0;
}
```

Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t1
```

```
==7879== Memcheck, a memory error detector.
             ....
==7879== Conditional jump or move depends on uninitialised value(s)
==7879==    at 0x8048399: main (test1.c:5)
==7879==
==7879== Use of uninitialised value of size 4
==7879==    at 0x80483A7: main (test1.c:7)
==7879==
==7879== Invalid write of size 4
==7879==    at 0x80483A7: main (test1.c:7)
==7879==  Address 0xCEF8FE44 is not stack'd, malloc'd or (recently) free'd
==7879==
==7879== Process terminating with default action of signal 11 (SIGSEGV)
==7879==  Access not within mapped region at address 0xCEF8FE44
==7879==    at 0x80483A7: main (test1.c:7)
==7879==
==7879== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 3 from 1)
==7879== malloc/free: in use at exit: 0 bytes in 0 blocks.
==7879== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==7879== For counts of detected errors, rerun with: -v
==7879== All heap blocks were freed -- no leaks are possible.
Segmentation fault
```

Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t1
```

```
==7879== Memcheck, a memory error detector.
             ....
==7879== Conditional jump or move depends on uninitialised value(s)
==7879==    at 0x8048399: main (test1.c:5)
==7879==
==7879== Use of uninitialised value of size 4
==7879==    at 0x80483A7: main (test1.c:7)
==7879==
==7879== Invalid write of size 4
==7879==    at 0x80483A7: main (test1.c:7)
==7879==  Address 0xCEF8FE44 is not stack'd, malloc'd or (recently) free'd
==7879==
==7879== Process terminating with default action of signal 11 (SIGSEGV)
==7879==  Access not within mapped region at address 0xCEF8FE44
==7879==    at 0x80483A7: main (test1.c:7)
==7879==
==7879== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 3 from 1)
==7879== malloc/free: in use at exit: 0 bytes in 0 blocks.
==7879== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==7879== For counts of detected errors, rerun with: -v
==7879== All heap blocks were freed -- no leaks are possible.
Segmentation fault
```

Illegal read/write test2.c

```
#include <stdlib.h>
int main()
{
     int *p,i,a;
     p=malloc(10*sizeof(int));
         p[11]=1; ERROR
         a=p[11]; ERROR
         free(p);
     return 0;
}
```

Illegal read/write test2.c

```
#include <stdlib.h>
int main()
{
     int *p,i,a;
     p=malloc(10*sizeof(int));
         p[11]=1; ERROR
         a=p[11]; ERROR
         free(p);
     return 0;
}
```

Illegal read/write: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t2
```

```
        .....
==8081== Invalid write of size 4
==8081==    at 0x804840A: main (test2.c:6)
==8081==  Address 0x417B054 is 4 bytes after a block of size 40 alloc'd
==8081==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8081==    by 0x8048400: main (test2.c:5)
==8081==
==8081== Invalid read of size 4
==8081==    at 0x8048416: main (test2.c:7)
==8081==  Address 0x417B054 is 4 bytes after a block of size 40 alloc'd
==8081==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8081==    by 0x8048400: main (test2.c:5)
==8081==
==8081== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 1)
==8081== malloc/free: in use at exit: 0 bytes in 0 blocks.
==8081== malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==8081== For counts of detected errors, rerun with: -v
==8081== All heap blocks were freed -- no leaks are possible.
```

Invalid free:test3.c

```c
#include <stdlib.h>
int main()
{
     int *p,i;
     p=malloc(10*sizeof(int));
         for(i=0;i<10;i++)
            p[i]=i;
         free(p);
         free(p);     ERROR
     return 0;
}
```

Invalid free:test3.c

```
#include <stdlib.h>
int main()
{
      int *p,i;
      p=malloc(10*sizeof(int));
          for(i=0;i<10;i++)
             p[i]=i;
          free(p);
          free(p);     ERROR
      return 0;
}
```

Invalid free: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t3
```

```
        .....
==8208== Invalid free() / delete / delete[]
==8208==    at 0x40231CF: free (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8208==    by 0x804843C: main (test3.c:9)
==8208==  Address 0x417B028 is 0 bytes inside a block of size 40 free'd
==8208==    at 0x40231CF: free (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8208==    by 0x8048431: main (test3.c:8)
==8208==
==8208== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
==8208== malloc/free: in use at exit: 0 bytes in 0 blocks.
==8208== malloc/free: 1 allocs, 2 frees, 40 bytes allocated.
==8208== For counts of detected errors, rerun with: -v
==8208== All heap blocks were freed -- no leaks are possible.
```

Mismatched use of functions:test4.cpp

- If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free**.
- If allocated with **new[]**, you must dealloacate with **delete[]**.
- If allocated with **new**, you must deallocate with **delete**.

```
#include <stdlib.h>
int main()
{
    int *p,i;
    p=(int*)malloc(10*sizeof(int));
        for(i=0;i<10;i++)
            p[i]=i;
        delete(p);      ERROR
    return 0;
}
```

Mismatched use of functions:test4.cpp

- If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free**.
- If allocated with **new[]**, you must dealloacate with **delete[]**.
- If allocated with **new**, you must deallocate with **delete**.

```
#include <stdlib.h>
int main()
{
     int *p,i;
     p=(int*)malloc(10*sizeof(int));
         for(i=0;i<10;i++)
            p[i]=i;
         delete(p);    ERROR
     return 0;
}
```

Mismatched use of functions:test4.cpp

- If allocated with **malloc**,**calloc**,**realloc**,**valloc** or **memalign**, you must deallocate with **free**.
- If allocated with **new[]**, you must dealloacate with **delete[]**.
- If allocated with **new**, you must deallocate with **delete**.

```
#include <stdlib.h>
int main()
{
     int *p,i;
     p=(int*)malloc(10*sizeof(int));
         for(i=0;i<10;i++)
            p[i]=i;
         delete(p);    ERROR
     return 0;
}
```

Mismatched use of functions: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t4
```

```
        .....
==8330== Mismatched free() / delete / delete []
==8330== at 0x4022EE6: operator delete(void*) (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8330==by 0x80484F1: main (test4.c:8)
==8330==Address 0x4292028 is 0 bytes inside a block of size 40 alloc'd
==8330==at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8330==by 0x80484C0: main (test4.c:5)
==8330==
==8330==ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
==8330==malloc/free: in use at exit: 0 bytes in 0 blocks.
==8330==malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==8330==For counts of detected errors, rerun with: -v
==8330==All heap blocks were freed -- no leaks are possible.
```

Invalid system call parameter:test5.c

```c
#include <stdlib.h>
#include <unistd.h>
int main()
{
      int *p;
      p=malloc(10);
       read(0,p,100); ERROR
          free(p);
      return 0;
}
```

Invalid system call parameter:test5.c

```
#include <stdlib.h>
#include <unistd.h>
int main()
{
      int *p;
      p=malloc(10);
       read(0,p,100); ERROR
          free(p);
      return 0;
}
```

Invalid system call parameter: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t5
```

```
...
==18007== Syscall param read(buf) points to unaddressable byte(s)
==18007==    at 0x4EEC240: __read_nocancel (in /lib64/libc-2.5.so)
==18007==    by 0x40056F: main (test5.c:7)
==18007==  Address 0x517d04a is 0 bytes after a block of size 10 alloc'd
==18007==    at 0x4C21168: malloc (vg_replace_malloc.c:236)
==18007==    by 0x400555: main (test5.c:6)
...
```

Memory leak detection:test6.c

```
#include <stdlib.h>
        int main()
        {
            int *p,i;
            p=malloc(5*sizeof(int));
            for(i=0; i<5;i++)
                p[i]=i;
            free(p);
                return 0;
        }
```

Summer School on PARALLEL COMPUTING

Memory leak detection:test6.c

```
#include <stdlib.h>
        int main()
        {
            int *p,i;
            p=malloc(5*sizeof(int));
            for(i=0; i<5;i++)
                p[i]=i;
            free(p);
                return 0;
        }
```

CINECA

Memory leak detection: Valgrind output

```
ruggiero@shiva:> valgrind --tool=memcheck --leak-check=full ./t6
```

```
.....
==8237== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 3 from 1)
==8237== malloc/free: in use at exit: 20 bytes in 1 blocks.
==8237== malloc/free: 1 allocs, 0 frees, 20 bytes allocated.
==8237== For counts of detected errors, rerun with: -v
==8237== searching for pointers to 1 not-freed blocks.
==8237== checked 65,900 bytes.
==8237==
==8237== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8237==    at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==8237==    by 0x80483D0: main (test6.c:5)
==8237==
==8237== LEAK SUMMARY:
==8237==    definitely lost: 20 bytes in 1 blocks.
==8237==      possibly lost: 0 bytes in 0 blocks.
==8237==    still reachable: 0 bytes in 0 blocks.
==8237==         suppressed: 0 bytes in 0 blocks.
```

What won’t Valgrind find?

```
int main()
{
    char x[10];
    x[11]='a';
}
```

- Valgrind doesn’t perform bound checking on static arrays (allocated on stack).
- Solution for testing purposes is simply to change static arrays into dinamically allocated memory taken from the heap, where you will get bounds-checking, though this could be a message of unfreed memory.

What won't Valgrind find?

```
int main()
{
    char x[10];
    x[11]='a';
}
```

- Valgrind doesn't perform bound checking on static arrays (allocated on stack).
- Solution for testing purposes is simply to change static arrays into dinamically allocated memory taken from the heap, where you will get bounds-checking, though this could be a message of unfreed memory.

sum.c: source

```c
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char* argv[])  {
    const int size=10;
    int n, sum=0;
    int* A = (int*)malloc( sizeof(int)*size);

    for(n=size; n>0; n--)
        A[n] = n;
    for(n=0; n<size; n++)
        sum+=A[n];
    printf("sum=%d\n", sum);
    return 0;
}
```

sum.c: compilation and run

```
ruggiero@shiva:~> gcc -O0 -g  -fbounds-check -ftrapv sum.c
```

```
ruggiero@shiva:~> ./a.out
```

```
sum=45
```

Valgrind:example

```
ruggiero@shiva:~> valgrind --leak-check=full --tool=memcheck ./a.out
```

```
==21579== Memcheck, a memory error detector.
...
==21791==Invalid write of size 4
==21791==at 0x804842A: main (sum.c:9)
==21791==Address 0x417B050 is 0 bytes after a block of size 40 alloc’d
at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck
.so)
==21791==by 0x8048410: main (sum.c:6)
==21791==Use of uninitialised value of size 4
==21791== at 0x408685B: _itoa_word (in /lib/libc-2.5.so)
==21791==by 0x408A581: vfprintf (in /lib/libc-2.5.so)
==21791==by 0x4090572: printf (in /lib/libc-2.5.so)
==21791==by 0x804846B: main (sum.c:12)
==21791==
==21791==Conditional jump or move depends on uninitialised value(s)
==21791==at 0x4086863: _itoa_word (in /lib/libc-2.5.so)
==21791==by 0x408A581: vfprintf (in /lib/libc-2.5.so)
==21791==by 0x4090572: printf (in /lib/libc-2.5.so)
==21791==by 0x804846B: main (sum.c:12)
==21791==40 bytes in 1 blocks are definitely lost in loss record 1 of 1
at 0x40235B5: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck
.so)
==21791==by 0x8048410: main (sum.c:6)
==21791==
```

Outline

Outline

What is gdb?

- The GNU Project debugger, is an open-source debugger.
- Protected by GNU General Public License (GPL).
- Runs on many Unix-like systems.
- Was first written by Richard Stallmann in 1986 as part of his GNU System.
- Is an Workstation Application Code extremely powerful all-purpose debugger.
- Its text-based user interface can be used to debug programs written in C, C++, Pascal, Fortran, and several other languages, including the assembly language for every micro-pro cessor that GNU supports.
- www.gnu.org/software/gdb

prime-numbers finding program

- Print a list of all primes which are less than or equal to the user-supplied upper bound *UpperBound*.
- See if *J* divides $K \leq$ *UpperBound*, for all values *J* which are
  - themselves prime (no need to try *J* if it is nonprime)
  - less than or equal to sqrt(K) (if *K* has a divisor larger than this square root, it must also have a smaller one, so no need to check for larger ones).
- *Prime[I]* will be 1 if *I* is prime, 0 otherwise.

Main.c

```c
#include <stdio.h>
#define MaxPrimes 50
int Prime[MaxPrimes],UpperBound;
int main()
{
    int N;
    printf("enter upper bound\n");
    scanf("%d",UpperBound);
    Prime[2] = 1;
    for (N = 3; N <= UpperBound; N += 2)
        CheckPrime(N);
    if (Prime[N]) printf("%d is a prime\n",N);
    return 0;
}
```

CheckPrime.c

```c
#define MaxPrimes 50
extern int Prime[MaxPrimes];
void CheckPrime(int K)
{
    int J; J = 2;
    while (1)  {
        if (Prime[J] == 1)
            if (K % J == 0)  {
                Prime[K] = 0;
                return;
            }
            J++;
    }
    Prime[K] = 1;
}
```

Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

```
enter upper bound
20
Segmentation fault
```

Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

```
enter upper bound
20
Segmentation fault
```

Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

```
enter upper bound
20
Segmentation fault
```

Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

```
enter upper bound
20
Segmentation fault
```

Compilation and run

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -O3 -o trova_primi
```

```
<ruggiero@matrix2 ~> ./trova_primi
```

```
enter upper bound
20
Segmentation fault
```

Compilation options for gdb

- You will need to compile your program with the appropriate flag to enable generation of symbolic debug information, the -g option is used for this.
- Don’t compile your program with optimization flags while you are debugging it.
  Compiler optimizations can "rewrite" your program and produce machine code that doesn’t necessarily match your source code.
  Compiler optimizations may lead to:
  - Misleading debugger behaviour.
    - Some variables you declared may not exist at all
    - some statements may execute in different places because they were moved out of loops
  - Obscure the bug.

Lower optimization level

- When your program has crashed, disable or lower optimization to see if the bug disappears.
  (optimization levels are not comparable between compilers, not even -O0).
- If the bug persists $\Longrightarrow$ you can be quite sure there's something wrong in your application.
- If the bug disappears,without a serious performance penalty $\Longrightarrow$ send the bug to your computing center and continue your simulations.
- But your program may still contain a bug that simply doesn't show up at lower optimization $\Longrightarrow$ have some checks to verify the correctness of your code.

Lower optimization level

- When your program has crashed, disable or lower optimization to see if the bug disappears.
  (optimization levels are not comparable between compilers, not even -O0).
- If the bug persists $\Longrightarrow$ you can be quite sure there's something wrong in your application.
- If the bug disappears,without a serious performance penalty $\Longrightarrow$ send the bug to your computing center and continue your simulations.
- But your program may still contain a bug that simply doesn't show up at lower optimization $\Longrightarrow$ have some checks to verify the correctness of your code.

Starting gdb

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -g -O0 -o trova_primi
```

```
<ruggiero@matrix2 ~>gdb trova_primi
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb)
```

Starting gdb

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -g -O0 -o trova_primi
```

```
<ruggiero@matrix2 ~>gdb trova_primi
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb)
```

Starting gdb

```
<ruggiero@matrix2 ~>gcc Main.c CheckPrime.c -g -O0 -o trova_primi
```

```
<ruggiero@matrix2 ~>gdb trova_primi
```

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.1)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

```
(gdb)
```

gdb: Basic commands

- run (r): start debugged program.
- help (h): print list of commands.
- she: execute the rest of the line as a shell command.
- where, backtrace (bt): print a backtrace of entire stack.
- kill (k): kill the child process in which program is running under gdb.
- list (l) linenum: print lines centered around line number lineum in the current source file.
- quit(q): exit gdb.

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6
#2 0x000000000040054f in main () at Main.c:8
```

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6
#2 0x000000000040054f in main () at Main.c:8
```

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6
#2 0x000000000040054f in main () at Main.c:8
```

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6
#2 0x000000000040054f in main () at Main.c:8
```

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound

20

Program received signal SIGSEGV, Segmentation fault.
0xd03733d4 in number () from /usr/lib/libc.a(shr.o)
```

```
(gdb) where
```

```
#0 0x0000003faa258e8d in _IO_vfscanf_internal () from /lib64/libc.so.6
#1 0x0000003faa25ee7c in scanf () from /lib64/libc.so.6
#2 0x000000000040054f in main () at Main.c:8
```

prime-number finding program

```
(gdb) list Main.c:8
```

```
3    int Prime[MaxPrimes],UpperBound;
5      main()
6      {   int N;
7          printf("enter upper bound\n");
8          scanf("%d",UpperBound);
9          Prime[2] = 1;
10         for (N = 3; N <= UpperBound; N += 2)
11            CheckPrime(N);
12            if (Prime[N]) printf("%d is a prime\n",N);
```

prime-number finding program

```
(gdb) list Main.c:8
```

```
3    int Prime[MaxPrimes],UpperBound;
5      main()
6      {  int N;
7         printf("enter upper bound\n");
8         scanf("%d",UpperBound);
9         Prime[2] = 1;
10        for (N = 3; N <= UpperBound; N += 2)
11           CheckPrime(N);
12           if (Prime[N]) printf("%d is a prime\n",N);
```

Main.c :new version

```
#include <stdio.h>
#define MaxPrimes 50
int Prime[MaxPrimes],UpperBound;
int main()
{
      int N;
      printf("enter upper bound\n");
      scanf("%d", &UpperBound);
      Prime[2] = 1;
      for (N = 3; N <= UpperBound; N += 2)
         CheckPrime(N);
         if (Prime[N]) printf("%d is a prime\n",N);
      return 0;
}
```

In other shell COMPILATION

Main.c :new version

```
#include <stdio.h>
#define MaxPrimes 50
int Prime[MaxPrimes],UpperBound;
int main()
{
      int N;
      printf("enter upper bound\n");
      scanf("%d", &UpperBound);
      Prime[2] = 1;
      for (N = 3; N <= UpperBound; N += 2)
         CheckPrime(N);
         if (Prime[N]) printf("%d is a prime\n",N);
      return 0;
}
```

In other shell COMPILATION

Main.c :new version

```c
#include <stdio.h>
#define MaxPrimes 50
int Prime[MaxPrimes],UpperBound;
int main()
{
      int N;
      printf("enter upper bound\n");
      scanf("%d", &UpperBound);
      Prime[2] = 1;
      for (N = 3; N <= UpperBound; N += 2)
         CheckPrime(N);
         if (Prime[N]) printf("%d is a prime\n",N);
      return 0;
}
```

In other shell COMPILATION

prime-number finding program

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005bb in CheckPrime (K=0x3) at CheckPrime.c:7
7                if (Prime[J] == 1)
```

prime-number finding program

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005bb in CheckPrime (K=0x3) at CheckPrime.c:7
7                if (Prime[J] == 1)
```

prime-number finding program

```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005bb in CheckPrime (K=0x3) at CheckPrime.c:7
7               if (Prime[J] == 1)
```

prime-number finding program

```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005bb in CheckPrime (K=0x3) at CheckPrime.c:7
7                if (Prime[J] == 1)
```

prime-number finding program

```
(gdb) p J
```

```
$1 = 1008
```

```
gdb l CheckPrime.c:7
```

```
2       extern int Prime[MaxPrimes];
3         CheckPrime(int K)
4         {
5             int J; J = 2;
6             while (1)  {
7                if (Prime[J] == 1)
8                   if (K % J == 0)  {
9                      Prime[K] = 0;
10                     return;
11                  }
```

prime-number finding program

```
(gdb) p J
```

```
$1 = 1008
```

```
gdb l CheckPrime.c:7
```

```
2       extern int Prime[MaxPrimes];
3         CheckPrime(int K)
4         {
5             int J; J = 2;
6             while (1)  {
7                if (Prime[J] == 1)
8                   if (K % J == 0)  {
9                      Prime[K] = 0;
10                     return;
11                  }
```

prime-number finding program

```
(gdb) p J
```

```
$1 = 1008
```

```
gdb l CheckPrime.c:7
```

```
2       extern int Prime[MaxPrimes];
3         CheckPrime(int K)
4         {
5             int J; J = 2;
6             while (1)  {
7                if (Prime[J] == 1)
8                   if (K % J == 0)  {
9                      Prime[K] = 0;
10                     return;
11                  }
```

prime-number finding program

```
(gdb) p J
```

```
$1 = 1008
```

```
gdb l CheckPrime.c:7
```

```
2       extern int Prime[MaxPrimes];
3         CheckPrime(int K)
4         {
5             int J; J = 2;
6             while (1)  {
7                if (Prime[J] == 1)
8                   if (K % J == 0)  {
9                      Prime[K] = 0;
10                     return;
11                  }
```

CheckPrime.c:new version

```
#define MaxPrimes 50
extern int Prime[MaxPrimes];
void CheckPrime(int K)
   {
      int J; J = 2;
       while(1){
          if (Prime[J] == 1)
             if (K % J == 0)  {
                Prime[K] = 0;
                return;
             }
       J++;
       }
       Prime[K] = 1;
   }
```

CheckPrime.c:new version

```c
#define MaxPrimes 50
extern int Prime[MaxPrimes];
void CheckPrime(int K)
   {
      int J;
       for (J = 2; J*J <= K; J++)
          if (Prime[J] == 1)
             if (K % J == 0)  {
                Prime[K] = 0;
                return;
             }
       Prime[K] = 1;
   }
```

prime-number finding program

In other shell COMPILATION

```
(gdb)
```

Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program exited normally.
```

prime-number finding program

In other shell COMPILATION

```
(gdb)
```

Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program exited normally.
```

prime-number finding program

In other shell COMPILATION

```
(gdb)
```

Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program exited normally.
```

prime-number finding program

In other shell COMPILATION

```
(gdb)
```

Kill the program being debugged? (y or n)   y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program exited normally.
```

prime-number finding program

In other shell COMPILATION

```
(gdb)
```

Kill the program being debugged? (y or n) y

```
(gdb) run
```

```
Starting program: trova_primi
enter upper bound
```

20

```
Program exited normally.
```

gdb commands

```
(gdb) help break
```

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

```
(gdb) help display
```

```
Print value of expression EXP each time the program stops.
.........
Use "undisplay" to cancel display requests previously made.
```

gdb commands

```
(gdb) help break
```

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

```
(gdb) help display
```

```
Print value of expression EXP each time the program stops.
.........
Use "undisplay" to cancel display requests previously made.
```

gdb commands

```
(gdb) help break
```

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

```
(gdb) help display
```

```
Print value of expression EXP each time the program stops.
.........
Use "undisplay" to cancel display requests previously made.
```

gdb commands

```
(gdb) help break
```

```
Set breakpoint at specified line or function.
break [LOCATION] [thread THREADNUM] [if CONDITION]
LOCATION may be a line number, function name, or "*" and an address.
If a line number is specified, break at start of code for that line.
If a function is specified, break at start of code for that function.
If an address is specified, break at that exact address.
.........
Multiple breakpoints at one place are permitted,
and useful if conditional.
.........
```

```
(gdb) help display
```

```
Print value of expression EXP each time the program stops.
.........
Use "undisplay" to cancel display requests previously made.
```

gdb commands

```
(gdb) help next
```

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) help step
```

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) break Main.c:1
```

```
Breakpoint 1 at 0x8048414: file Main.c, line 1.
```

gdb commands

```
(gdb) help next
```

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) help step
```

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) break Main.c:1
```

```
Breakpoint 1 at 0x8048414: file Main.c, line 1.
```

gdb commands

```
(gdb) help next
```

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) help step
```

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) break Main.c:1
```

```
Breakpoint 1 at 0x8048414: file Main.c, line 1.
```

gdb commands

```
(gdb) help next
```

```
Step program, proceeding through subroutine calls.
Like the "step" command as long as subroutine calls do not happen;
when they do, the call is treated as one instruction.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) help step
```

```
Step program until it reaches a different source line.
Argument N means do this N times
(or till program stops for another reason).
```

```
(gdb) break Main.c:1
```

```
Breakpoint 1 at 0x8048414: file Main.c, line 1.
```

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
Failed to read a valid object file image from memory.

Breakpoint 1, main () at Main.c:6
6           {   int N;
```

```
(gdb) next
```

```
main () at Main.c:7
7               printf("enter upper bound\n");
```

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
Failed to read a valid object file image from memory.

Breakpoint 1, main () at Main.c:6
6          {  int N;
```

```
(gdb) next
```

```
main () at Main.c:7
7              printf("enter upper bound\n");
```

prime-number finding program

```
(gdb) r
```

```
Starting program: trova_primi
Failed to read a valid object file image from memory.

Breakpoint 1, main () at Main.c:6
6           {  int N;
```

```
(gdb) next
```

```
main () at Main.c:7
7              printf("enter upper bound\n");
```

prime-number finding program

```
(gdb) next

8               scanf("%d",&UpperBound);

(gdb) next

20

9               Prime[2] = 1;

(gdb) next

10              for (N = 3; N <= UpperBound; N += 2)

(gdb) next

11                 CheckPrime(N);
```

prime-number finding program

```
(gdb) next

8               scanf("%d",&UpperBound);

(gdb) next
20
9               Prime[2] = 1;

(gdb) next

10              for (N = 3; N <= UpperBound; N += 2)

(gdb) next

11                 CheckPrime(N);
```

prime-number finding program

```
(gdb) next

8               scanf("%d",&UpperBound);

(gdb) next
20
9               Prime[2] = 1;

(gdb) next

10              for (N = 3; N <= UpperBound; N += 2)

(gdb) next

11                  CheckPrime(N);
```

prime-number finding program

```
(gdb) next

8               scanf("%d",&UpperBound);

(gdb) next
20
9               Prime[2] = 1;

(gdb) next

10              for (N = 3; N <= UpperBound; N += 2)

(gdb) next

11                 CheckPrime(N);
```

prime-number finding program

```
(gdb) next

8               scanf("%d",&UpperBound);

(gdb) next

20
9               Prime[2] = 1;

(gdb) next

10              for (N = 3; N <= UpperBound; N += 2)

(gdb) next

11                  CheckPrime(N);
```

prime-number finding program

```
(gdb) next

8               scanf("%d",&UpperBound);

(gdb) next

20

9               Prime[2] = 1;

(gdb) next

10              for (N = 3; N <= UpperBound; N += 2)

(gdb) next

11                 CheckPrime(N);
```

prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6
6             for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

```
12             Prime[K] = 1;
```

```
(gdb) next
```

```
13        }
```

prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6
6             for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

```
12            Prime[K] = 1;
```

```
(gdb) next
```

```
13        }
```

prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6
6             for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

```
12            Prime[K] = 1;
```

```
(gdb) next
```

```
13          }
```

prime-number finding program

```
(gdb) display N
```

```
1: N = 3
```

```
(gdb) step
```

```
CheckPrime (K=3) at CheckPrime.c:6
6             for (J = 2; J*J <= K; J++)
```

```
(gdb) next
```

```
12            Prime[K] = 1;
```

```
(gdb) next
```

```
13        }
```

prime-number finding program

```
(gdb)  n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

```
11                  CheckPrime(N);
1: N = 5
```

```
(gdb) n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 5
```

```
(gdb) n
```

```
11                  CheckPrime(N);
1: N = 7
```

prime-number finding program

```
(gdb)  n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 5
```

```
(gdb) n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 5
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 7
```

prime-number finding program

```
(gdb)  n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 5
```

```
(gdb) n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 5
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 7
```

prime-number finding program

```
(gdb)  n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 5
```

```
(gdb) n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 5
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 7
```

prime-number finding program

```
(gdb)  n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 3
}
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 5
```

```
(gdb) n
```

```
10              for (N = 3; N <= UpperBound; N += 2)
1: N = 5
```

```
(gdb) n
```

```
11                 CheckPrime(N);
1: N = 7
```

prime-number finding program

```
(gdb) l  Main.c:10
```

```
5          main()
6          {  int N;
7             printf("enter upper bound\n");
8             scanf("%d",&UpperBound);
9             Prime[2] = 1;
10            for (N = 3; N <= UpperBound; N += 2)
11               CheckPrime(N);
12               if (Prime[N]) printf("%d is a prime\n",N);
13            return 0;
14         }
```

prime-number finding program

```
(gdb) l  Main.c:10
```

```
5          main()
6          {  int N;
7             printf("enter upper bound\n");
8             scanf("%d",&UpperBound);
9             Prime[2] = 1;
10            for (N = 3; N <= UpperBound; N += 2)
11               CheckPrime(N);
12               if (Prime[N]) printf("%d is a prime\n",N);
13            return 0;
14         }
```

Main.c :new version

```
#include <stdio.h>
#define MaxPrimes 50
 int Prime[MaxPrimes],
 UpperBound;
   main()
   {  int N;
      printf("enter upper bound\n");
      scanf("%d",&UpperBound);
      Prime[2] = 1;
      for (N = 3; N <= UpperBound; N += 2){
         CheckPrime(N);
         if (Prime[N]) printf("%d is a prime\n",N);
      }
      return 0;
   }
```

Main.c :new version

```
#include <stdio.h>
#define MaxPrimes 50
 int Prime[MaxPrimes],
 UpperBound;
   main()
   {  int N;
      printf("enter upper bound\n");
      scanf("%d",&UpperBound);
      Prime[2] = 1;
      for (N = 3; N <= UpperBound; N += 2){
         CheckPrime(N);
         if (Prime[N]) printf("%d is a prime\n",N);
      }
      return 0;
   }
```

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

```
Kill the program being debugged? (y or n)    y
```

```
(gdb) d
```

```
Delete all breakpoints? (y or n)    y
```

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

```
20
```

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb) r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

```
(gdb) d
```

Delete all breakpoints? (y or n)   y

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n) y

```
(gdb) d
```

Delete all breakpoints? (y or n) y

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

In other shell COMPILATION

```
(gdb) kill
```

Kill the program being debugged? (y or n)   y

```
(gdb) d
```

Delete all breakpoints? (y or n)   y

```
(gdb)r
```

```
Starting program: trova_primi
enter upper bound
```

20

prime-number finding program

```
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime

Program exited normally.
```

prime-number finding program

```
(gdb) list Main.c:6
```

```
1       #include <stdio.h>
2       #define MaxPrimes 50
3        int Prime[MaxPrimes],
4        UpperBound;
5          main()
6          {  int N;
7             printf("enter upper bound\n");
8             scanf("%d",&UpperBound);
9             Prime[2] = 1;
10            for (N = 3; N <= UpperBound; N += 2){
```

prime-number finding program

```
(gdb) list Main.c:6
```

```
1       #include <stdio.h>
2       #define MaxPrimes 50
3        int Prime[MaxPrimes],
4        UpperBound;
5          main()
6          {  int N;
7             printf("enter upper bound\n");
8             scanf("%d",&UpperBound);
9             Prime[2] = 1;
10            for (N = 3; N <= UpperBound; N += 2){
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8             scanf("%d",&UpperBound);
```

```
(gdb) next
```

20

```
9             Prime[2] = 1;
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8             scanf("%d",&UpperBound);
```

```
(gdb) next
```

```
20

9             Prime[2] = 1;
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8               scanf("%d",&UpperBound);
```

```
(gdb) next
```

```
20

9               Prime[2] = 1;
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8               scanf("%d",&UpperBound);
```

```
(gdb) next
```

20

```
9               Prime[2] = 1;
```

prime-number finding program

```
(gdb) break Main.c:8
```

```
Breakpoint 1 at 0x10000388: file Main.c, line 8.
```

```
(gdb)run
```

```
Starting program: trova_primi
enter upper bound
Breakpoint 1, main () at /afs/caspur.it/user/r/ruggiero/Main.c:8
8             scanf("%d",&UpperBound);
```

```
(gdb) next
```

```
20

9             Prime[2] = 1;
```

prime-number finding program

```
(gdb) set UpperBound=40
(gdb) continue
```

```
Continuing.
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime
23 is a prime
29 is a prime
31 is a prime
37 is a prime

Program exited normally.
```

prime-number finding program

```
(gdb) set UpperBound=40
(gdb) continue
```

```
Continuing.
3 is a prime
5 is a prime
7 is a prime
11 is a prime
13 is a prime
17 is a prime
19 is a prime
23 is a prime
29 is a prime
31 is a prime
37 is a prime

Program exited normally.
```

Debugging post mortem

- When a program exits abnormally the operating system can write out core file, which contains the memory state of the program at the time it crashed.
- Combined with information from the symbol table produced by -g the core file can be used fo find the line where program stopped, and the values of its variables at that point.
- Some systems are configured to not write core file by default, since the files can be large and rapidly fill up the available hard disk space on a system.
- In the GNU Bash shell the command ulimit -c control the maximum size of the core files. If the size limit is set to zero, no core files are produced.

```
ulimit -c unlimited
gdb exe_file core
```

Debugging post mortem

- When a program exits abnormally the operating system can write out core file, which contains the memory state of the program at the time it crashed.
- Combined with information from the symbol table produced by -g the core file can be used fo find the line where program stopped, and the values of its variables at that point.
- Some systems are configured to not write core file by default, since the files can be large and rapidly fill up the available hard disk space on a system.
- In the GNU Bash shell the command ulimit -c control the maximum size of the core files. If the size limit is set to zero, no core files are produced.

```
ulimit -c unlimited
gdb exe_file core
```

Outline

Totalview (www.totalviewtech.com)

- Used for debugging and analyzing both serial and parallel programs.
- Supported languages include the usual HPC application languages:
  - C,C++,Fortran
  - Mixed C/C++ and Fortran
  - Assembler
- Supported many commercial and Open Source Compilers.
- Designed to handle most types of HPC parallel coding(multi-process and/or multi-threaded applications).
- Supported on most HPC platforms.
- Provides both a GUI and command line interface.
- Can be used to debug programs, running processes, and core files.
- Provides graphical visualization of array data.
- Includes a comprehensive built-in help system.
- And more...

Compilation options for Totalview

- You will need to compile your program with the appropriate flag to enable generation of symbolic debug information. For most compilers, the -g option is used for this.
- It is recommended to compile your program without optimization flags while you are debugging it.
- TotalView will allow you to debug executables which were not compiled with the -g option. However, only the assembler code can be viewed.
- Some compilers may require additional compilation flags. See the *TotalView User's Guide* for details.

```
ifort [option] -O0 -g file_source.f -o filename
```

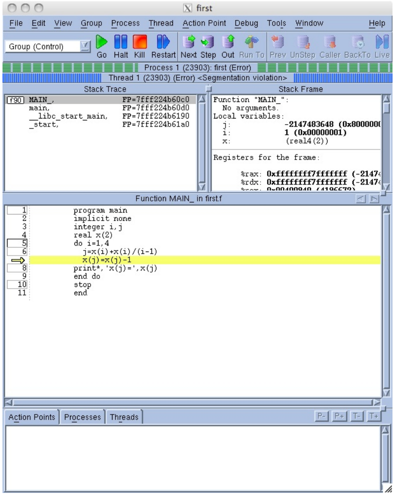
How to use TOTALVIEW on FERMI:
TOTALVIEW on FERMI.

Starting Totalview

Command	Action
totalview	Starts the debugger.You can then load a program or corefile, or else attach to a running process.
totalview *filename*	Starts the debugger and loads the program specified by *filename*.
totalview *filename corefile*	Starts the debugger and loads the program specified by *filename* and its core file specified by *corefile*.
totalview *filename* -a *args*	Starts the debugger and passes all subsequent arguments (specified by *args*) to the program specified by *filename*. The -a option must appear after all other TotalView options on the command line.

Totalview:panel

1. Stack Trace
   - Call sequence
2. Stack Frame
   - Local variables and their values
3. Source Window
   - Indicates presently executed statement
   - Last statement executed if program crashed
4. Info tabs
   - Informations about processes and action points.


first
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Prev UnStep Caller BackTo Live
Stack Trace
Stack Frame
Function MAIN_ in first.f
Action Points Processes Threads
P- P+ T- T+

```
Process 1 (23903): first (Error)
Thread 1 (23903) (Error) <Segmentation violation>
```

```
f90  MAIN_,               FP=7fff224b60c0
     main,                FP=7fff224b60d0
     __libc_start_main,   FP=7fff224b6190
     _start,              FP=7fff224b61a0
```

```
Function "MAIN_":
  No arguments.
Local variables:
  j:                -2147483648 (0x8000000
  i:                1 (0x00000001)
  x:                (real4(2))

Registers for the frame:

      %rax: 0xffffffff7fffffff (-2147
      %rdx: 0xffffffff7fffffff (-2147
```

```
      program main
      implicit none
      integer i,j
      real x(2)
      do i=1,4
        j=x(i)+x(i)/(i-1)
        x(j)=x(j)-1
      print*,'x(j)=',x(j)
      end do
      stop
      end
```

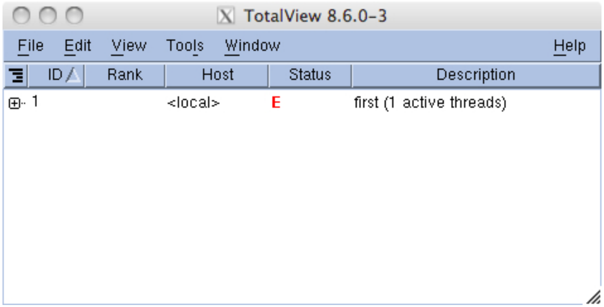
Totalview:Action points

- Breakpoint stops the excution of the process and threads that reach it.
  - Unconditional
  - Conditional: stop only if the condition is satisfied.
  - Evaluation: stop and excute a code fragment when reached.
- Process barrier point synchronizes a set of processes or threads.
- Watchpoint monitors a location in memory and stop execution when its value changes.

Totalview:Setting Action points

- Breakpoint
  - Right click on a source line → Set breakpoint
  - Click on the line number
- Watchpoint
  - Right click on a variable → Create watchpoint
- Barrier point
  - Right click on a source line → Set barrier
- Edit action point property
  - Rigth click on a action point in the Action Points tab → Properties.

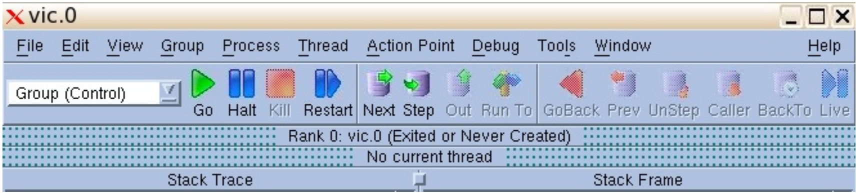
Totalview:Status



ID	Rank	Host	Status	Description
1		<local>	E	first (1 active threads)

Status Code	Description
T	Thread is stopped
B	Stopped at a breakpoint
E	Stopped because of a error
W	At a watchpoint
H	In a Hold state
M	Mixed - some threads in a process are running and some not
R	Running

Totalview:Execution control commands



Command	Description
Go	Start/resume excution
Halt	Stop excution
Kill	Terminate the job
Restart	Restarts a running program, or one that has stopped without exiting
Next	Run to next source line or instruction. If the next line/instruction calls a function the entire function will be excuted and control will return to the next source line or instruction.
Step	Run to next source line or instruction. If the next line/instruction calls a function, excution will stop within function.
Out	Excute to the completion of a function. Returns to the instruction after one which called the function.
Run to	Allows you to arbitrarily click on any source line and then run to that point.

Totalview:Mouse buttons

Mouse Button	Purpose	Description	Examples
Left	Select	Clicking on object causes it to be selected and/ or to perform its action	Clicking a line number sets a breakpoint. Clicking on a process/thread name in the root window will cause its source code to appear in the Process Window's source frame.
Middle	Dive	Shows additional information about the object - usually by popping open a new window.	Clicking on an array object in the source frame will cause a new window to pop open, showing the array's values.
Rigth	Menu	Pressing and holding this button a window/frame will cause its associated menu to pop open.	Holding this but ton while the mouse pointer is in the Root Window will cause the Root Window menu to appear. A menu selection can then be made by dragging the mouse pointer while continuing to press the middle button down.

Outline

Other debbugers...

- DDD
- Kdevelop
- Eclipse
- ...

Outline

Core Files and addr2line utility

- When a Blue Gene/Q program runs unsuccesfully, a core file is generated.
- You can use the **addr2line** utility to analyze the core file.
- This utility uses the debugging information in an executable program to provide information about the file name and line number for the source that was used to create the program.
- addr2line retrieves source code location from hexadecimal address
- Standard Linux command
  http://www.linuxcommand.org/man_pages/addr2line1.html

addr2line – how-to

- Blue Gene core files are lightweight text files
- Hexadecimal addresses in section STACK describe function call chain until program exception: section delimited by tags: +++STACK / —STACK
- Example of core file output format:

```
+++STACK
Frame Address    Saved Link Reg
0000001fffff5ac0  000000000000001c
0000001fffff5bc0  00000000018b2678
0000001fffff5c60  00000000015046d0
0000001fffff5d00  00000000015738a8
0000001fffff5e00  00000000015734ec
0000001fffff5f00  000000000151a4d4
0000001fffff6000  00000000015001c8
---STACK
```

- Control core generation (c.f BG/Q Application Development redbook)
  - export BG_COREDUMPBINARY= 0 | 1
  - export BG_COREDUMPDISABLED= 0 | 1
  - export BG_COREDUMPONEXIT= 0| 1

addr2line – how-to

- From the core file output, save only the addresses in the Saved Link Reg column:

```
000000000000001c
00000000018b2678
00000000015046d0
00000000015738a8
00000000015734ec
000000000151a4d4
00000000015001c8
```

- Replace the first eight 0s with 0x:
  00000000018b2678 => 0x018b2678
  (page 89 of Application redbook - bgqtranslate.pl)
- Run the addr2line utility:
  addr2line -e <**binary**> <**hexadecimal address**>
  addr2line -e <**executable**> < <**file with hexadecimal address**>

QUESTIONS ???

Outline

SCALASCA on FERMI

In what follows,

- examples of job scripts to use SCALASCA on FERMI.
- how to have a GUI on FERMI

Analysis – Pure MPI

```
#!/bin/bash
#
# @ job_name = myjob.$(jobid)
# @ output = $(job_name).out
# @ error = $(job_name).err
# @ environment = COPY_ALL
# @ job_type = bluegene
# @ wall_clock_limit = 1:00:00
# @ bg_size = 128
# @ account_no = <Account number>
# @ notification = always
# @ notify_user = <valid email address>
# @ queue

module load bgq-xl/1.0
module load scalasca/1.4.2
scalasca -analyze runjob --np 256 --ranks-per-node 2
--exe <my_exe>
```

Analysis – MPI+OpenMP

```
#!/bin/bash
#
# @ job_name = myjob.$(jobid)
# @ output = $(job_name).out
# @ error = $(job_name).err
# @ environment = COPY_ALL
# @ job_type = bluegene
# @ wall_clock_limit = 1:00:00
# @ bg_size = 128
# @ account_no = <Account number>
# @ notification = always
# @ notify_user = <valid email address>
# @ queue

module load bgq-xl/1.0
module load scalasca/1.4.2
scalasca -analyze runjob --np 256 --ranks-per-node 4
-envs OMP_NUM_THREADS=4 --exe <my_exe>
```

Archive with log files 1/2

- Pure MPI:
  scalasca -analyze runjob –np 256 –ranks-per-node 2 –exe <my_exe>
  ==> **epik_<myexe>_2p256_sum**

- MPI + OpenMP:
  scalasca -analyze runjob –np 256 –ranks-per-node 4 -envs OMP_NUM_THREADS=4 –exe <my_exe>
  ==> **epik_<myexe>_4p256x4_sum**

Archive with log files 2/2

In each epik archive there are the following files:

epik.conf	Measurement configuration when the experiment was collected
epik.log	Output of the instrumented program and measurement system
epik.path	Callpath-tree recorded by the measurement system
epitome.cube	Intermediate analysis report of the runtime summarization system
summary.cube[.gz]	Post-processed analysis report of runtime summarization

VNC on FERMI

- In order to use a tool with GUI, first you need to have downloaded and installed **VNCviewer** on your local machine. (http://www.realvnc.com/download/viewer/)
- Windows users will also find useful **Cygwin**, a Linux-like environment for Windows. During installation, be sure to select ’openSSH’ from the list of available packages. (http://cygwin.com/setup.exe)

VNC connection on FERMI 1

- On FERMI, load tightvnc module: **module load tightvnc**
- Execute the script vncserver_wrapper: **vncserver_wrapper**
- Instructions will appear.

  Copy/paste to your local machine (Cygwin shell if Windows) this line from those instructions:
  ssh -L 59xx:localhost:59xx -L 58xx:localhost:58xx -N <username>@login<no>.fermi.cineca.it

  where **xx** is your VNC display number, and **<no>** is the number of the front-end node you are logged into (01, 02, 07 or 08)

VNC connection on FERMI 2

- Open VNCViewer:
  - On Linux, use another local shell and type:
    vncviewer localhost:xx
  - On Windows, double click on VNCviewer icon and write **localhost:xx** when asked for the server.
- Type your VNC password (or choose it, if it is your first visit)
- The GUI will appear and enjoy to use SCALASCA on FERMI !!!

Back to main-scalasca.

Totalview on FERMI

In what follows,

- preliminaries to use Totalview on FERMI
- setting of the job script
- start to debug

Using Totalview: preliminaries

- In order to use a tool with GUI, first you need to have downloaded and installed VNCviewer on your local machine. (http://www.realvnc.com/download/viewer/)
- Windows users will also find useful Cygwin, a Linux-like environment for Windows. During installation, be sure to select 'openSSH' from the list of available packages. (http://cygwin.com/setup.exe)

Using Totalview: preparation

- On FERMI, load tightvnc module: **module load tightvnc**
- Execute the script vncserver_wrapper: **vncserver_wrapper**
- Instructions will appear.

  Copy/paste to your local machine (Cygwin shell if Windows) this line from those instructions:
  ssh -L 59xx:localhost:59xx -L 58xx:localhost:58xx -N <username>@login<no>.fermi.cineca.it

  where **xx** is your VNC display number, and **<no>** is the number of the front-end node you are logged into (01, 02, 07 or 08)
- Open VNCViewer:
  - On Linux, use another local shell and type:
    vncviewer localhost:xx
  - On Windows, double click on VNCviewer icon and write **localhost:xx** when asked for the server.
- Type your VNC password (or choose it, if it is your first visit)

Using Totalview: job script setting

1. Inside your job script, you have to load the proper module and export the DISPLAY environment variable:

   module load totalview

   export DISPLAY=fen<no>:xx

   where **xx** and **<no>** are as the above slide (you will find the correct DISPLAY name to export in vncserver_wrapper instructions)
2. Totalview execution line (inside your LoadLeveler script) will be as follows:

   totalview runjob -a <runjob arguments...>
3. Launch the job. When it will start running, you will find a Totalview window opened on your VNCviewer display!
4. Closing Totalview will also kill the job.

Using Totalview: start debugging

- Select “BlueGene” as a parallel system, and a number of tasks and nodes according to the arguments you gave to runjob during submission phase.
- Click “Go” (the green arrow) on the next screen and your application will start running.

Using Totalview: start debugging

User Guide for Totalview:

- module load totalview
- module show totalview
- $MANPATH : /cineca/prod/tools/totalview/8.11.0-0/binary/toolworks/totalview.8.11.0-0/doc/pdf

Using Totalview: licenses

WARNING:
due to license issues, you are NOT allowed to run Totalview sessions with more than 1024 tasks simultaneously!!!

You can visualize the usage status of the licenses by typing the command:

- module load totalview
- lmstat -c $LM_LICENSE_FILE -a

Back to main-totalview.