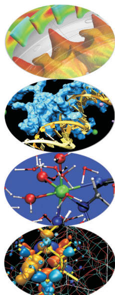
CINECA SCAI SuperComputing Applications and Innovation

SAPIENZA UNIVERSITÀ DI ROMA

Programmazione Avanzata

Vittorio Ruggiero

(v.ruggiero@cineca.it)



Roma, 26 Maggio 2014

CINECA

Outline

- Debugging
  - gdb
- PAPI
- Parallel profiling
- Parallel debugging

My program fails!

- Erroneous program results.
- Execution deadlock.
- Run-time crashes.

Assumption 1

Question: The same code still worked last week.

- Possible problem: An application runs in an environment with many components (OS, dynamic libraries, network, disks, ..). If any one has changed, this can impact the application.
- Fix: ask the helpdesk if anything changed since last week.
- Alternative problem: Often, the code is not exactly the same, only the user assumed that the change was not important.
- Fix: a version control system (cvs, svn, git, ..) will help you check that the code really was the same as last week or to see what changes were made.

Assumption 2

Question: The same program works for my colleague.

- Possible problem: do you really use the same executable?
- Check: use

```
which [command]
```

to see the location.

- Possible problem: dynamic executables load libraries at runtime. These are loaded from directories in your $*LD_LIBRARY_PATH*. Maybe you use different versions of libraries without knowing it?
- Check: use

```
ldd [program]
```

to check which libraries are used.

Assumption 3

Question: the program works fine on my own system, so something is wrong with yours

```
program helloworld
implicit none
print*,'Hello World!'
return
end program
```

- works fine with GNU and IBM compilers, but won't compile with the Intel compiler. Why?
- return statement is not allowed in Fortran mainprogram

```
> ifort return.f90
return.f90(5): error #6353: A RETURN statement is invalid in
the main program.
```

- Solution: check your program to see if it follows the language standard.

Assumption 4

Question: My program crashed. What did I do wrong?

- Answer: Your program depends on other libraries, the compiler, the OS, the network, etc. System libraries and compilers have bugs and hardware can fail, so it might not be your program's fault.

```
CALL MPI_Barrier(MPI_COMM_SELF, IERR)
```

- This call would segfault in some version of IBM MPI, although it is a correct MPI call.
- Solutions: read Changelogs or KnownBugs. Isolate the problem and send it to the helpdesk. Ask if it could be a known bug.
- However, 99% of the problems are related to the application.

Assumption 5

Question: It works only sometimes.

- Problem: Probably a race condition (bugs that cause undefined behaviour, depending on timing differences)

The ideal debugging process

- Find origins.
  - Identify test case(s) that reliably show existence of fault (when possible). Duplicate the bug.
- Isolate the origins of infection.
  - Correlate incorrect behaviour with program logic/code error.
- Correct.
  - Fixing the error, not just a symptom of it.
- Verify.
  - Where there is one bug, there is likely to be another.
  - The probability of the fix being correct is not 100 percent.
  - The probability of the fix being correct drops as the size of the program increases.
  - Beware of the possibility that an error correction creates a new error.

Bugs that can't be duplicated

- Dangling pointers.
- Initializations errors.
- Poorly syinchronized threads.
- Broken hardware.

Isolate the origins of infection

- Divide and conqueror.
- Change one thing at time.
- Determine what you changed since the last time it worked.
- Write down what you did, in what order, and what happened as a result.
- Correlate the events.

What is a Debuggers?

Debuggers are a software tools that help determine why program does not behave correctly. They aid a programmer in understanding a program and then finding the cause of the discrepancy. The programmer can then repair the defect and so allow the program to work according to its original intent. A debugger is a tool that controls the application being debugged so as to allow the programmer to follow the flow of program execution and , at any desired point, stop the program and inspect the state of the program to verify its correctness.

"How debuggers works Algorithms, data,structure, and Architecture"

Jonathan B. Rosemberg

Why use a Debugger?

- No need for precognition of what the error might be.
- Flexible.
  - Allows for ”live” error checking (no need to re−write and re−compile when you realize a certain type of error may be occuring).
- Dynamic.
  - Execution Control Stop execution on specified conditions: breakpoints
  - Interpretation Step-wise execution code
  - State Inspection Observe value of variables and stack
  - State Change Change the state of the stopped program.

Why people don't use debuggers?

- With simple errors, may not want to bother with starting up the debugger environment.
  - Obvious error.
  - Simple to check using prints/asserts.
- Hard-to-use debugger environment.
- Error occurs in optimized code.
- Changes execution of program (error doesn't occur while running debugger).

Why don't use print?

- Cluttered code.
- Cluttered output.
- Slowdown.
- Loss of data.
- Time consuming.
- And can be misleading.
  - Moves things around in memory, changes execution timing, etc.
  - Common for bugs to hide when print statements are added, and reappear when they're removed.

Outline

Debugging
- gdb

PAPI

Parallel profiling

Parallel debugging

What is gdb?

- The GNU Project debugger, is an open-source debugger.
- Released 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.
- 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-processor that GNU supports.
- `www.gnu.org/software/gdb`

prime-numbers 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


```
#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


```
#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


```
<~>gcc Main.c CheckPrime.c  -o trova_primi
```

```
<~> ./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.

Starting gdb


```
<~>gcc Main.c CheckPrime.c -g  -o trova_primi
```

```
<~>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.
- kill (k): kill the child process in which program is running under gdb.
- where, backtrace (bt): print a backtrace of entire stack.
- quit(q): exit gdb.

- break (b) : set breakpoint at specified line or function

- print (p) expr: print value of expression expr
- display expr: print value of expression expr each time the program stops.

gdb: Basic commands

- continue: continue program being debugged, after signal or breakpoint.
- next: step program, proceeding through subroutine calls.
- step: step program until it reaches a different source line

- help (h): print list of commands.
- she: execute the rest of the line as a shell command.
- list (l) linenum: print lines centered around line number lineum in the current source file.

prime-number


```
(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
```

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


```
1   #include <stdio.h>
2   #define MaxPrimes 50
3   int Prime[MaxPrimes],UpperBound;
4   int main()
5   {
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

```
1  #include <stdio.h>
2  #define MaxPrimes 50
3  int Prime[MaxPrimes],UpperBound;
4  int main()
5  {
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


```
1  #include <stdio.h>
2  #define MaxPrimes 50
3  int Prime[MaxPrimes],UpperBound;
4  int main()
5  {
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 }
```

In other shell COMPILATION

prime-number


```
(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                   }
```

CheckPrime.c:new version

```
1   #define MaxPrimes 50
2   extern int Prime[MaxPrimes];
3   void 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              }
12        J++;
13        }
14        Prime[K] = 1;
15     }
```

CheckPrime.c:new version


```
1   #define MaxPrimes 50
2   extern int Prime[MaxPrimes];
3   void CheckPrime(int K)
4      {
5        int J;
6            for (J = 2; J*J <= K; J++)
7            if (Prime[J] == 1)
8               if (K % J == 0)   {
9                  Prime[K] = 0;
10                 return;
11              }
12
13
14        Prime[K] = 1;
15     }
```

prime-number finding program


```
(gdb) kill
```

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

```
(gdb) she gcc -g Main.c CheckPrime.c -o triva_primi
```

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

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

prime-number finding program

```
(gdb) r

Starting program: trova_primi
Breakpoint 1, main () at Main.c:7
9               printf("enter upper bound\n");

(gdb) next

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

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

(gdb) next

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

(gdb) next

14                  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) 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          }
```

Main.c :new version


```
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)
11           CheckPrime(N);
12           if (Prime[N]) printf("%d is a prime\n",N);
13
14        return 0;
15     }
```

Main.c :new version


```
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){
11           CheckPrime(N);
12           if (Prime[N]) printf("%d is a prime\n",N);
13      }
14        return 0;
15     }
```

prime-number finding program


```
(gdb) kill
```

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

```
(gdb) she gcc -g Main.c CheckPrime.c -o trova_primi
```

```
(gdb) d
```

Delete all breakpoints? (y or n) y

prime-number finding program


```
(gdb) r
```

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

20

```
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) 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 /home/guest/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.
```

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.pid
```

Graphical Debuggers

- gdb -tui or gdbtui (text user interface)
- ddd (data dysplay debugger) is a graphical front-end for command-line debuggers.
- allinea ddt (Distributed Debugging Tool) is a comprehensive graphical debugger for scalar, multi-threaded and large-scale parallel applications that are written in C, C++ and Fortran.
- Rouge Wave Totalview
- Etc.

Esercitazione:debugging

```
https://hpc-forge.cineca.it/files/CoursesDev/public/2014/
Introduction_to_HPC_Scientific_Programming:_tools_and_techniques/
Rome/Debug_esercizi.tar

tar xvf Debug_esercizi.tar
```

- TEST1: semplice bug per familiarizzare con i comandi
- TEST2: altro semplice bug per familiarizzare con i comandi
- TEST3: calcolo di un elemento della successione di Fibonacci.
  - Input: un numero che rappresenta la posizione dell'elemento della successione di cui si vuole il valore
  - Output: stampa a schermo del valore dell'elemento richiesto
- TEST4: sorting
  - Input: un intero che rappresenta il numero di interi che si vogliono ordinare ed i relativi valori (in C i valori da ordinare)
  - Output: i valori in ordine crescente
- TEST5: Crivello di Eratostene (ricerca dei numeri primi)
  - Input: un numero intero che rappresenta il limite di ricerca
  - Output: L'elenco dei numeri primi inferiori e uguali a quello fornito come limite

Outline

Debugging

PAPI

Papi

Parallel profiling

Parallel debugging

Outline

Debugging

PAPI
- Papi

Parallel profiling

Parallel debugging

Papi

- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.

Papi

- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.

Papi

- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.
- PAPI (Performance Application Programming Interface) nasce esattamente con questo scopo. Caratteristiche principali:

Papi

- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.
- PAPI (Performance Application Programming Interface) nasce esattamente con questo scopo. Caratteristiche principali:
  - portabilità sulla maggior parte delle architetture Linux, Windows, etc inclusi acceleratori (NVIDIA, Intel MIC, etc)

Papi

- Strumenti come Gprof, etc hanno il loro punto di forza nella semplicità di utilizzo e nella scarsa se non nulla intrusività.
- Ovviamente se si vuole andare più a fondo nell'ottimizzazione del nostro codice, soprattutto in relazione a quelle che sono le caratteristiche peculiari dell'hardware a disposizione, occorrono altri strumenti.
- PAPI (Performance Application Programming Interface) nasce esattamente con questo scopo. Caratteristiche principali:
  - portabilità sulla maggior parte delle architetture Linux, Windows, etc inclusi acceleratori (NVIDIA, Intel MIC, etc)
  - si basa sull'utilizzo dei cosiddetti *Hardware Counters*: un insieme di registri "special-purpose" che misurano determinati eventi durante l'esecuzione del nostro programma.

Papi-cont.

- PAPI fornisce un’interfaccia agli *Hardware Counters* essenzialmente di due tipi:

Papi-cont.

- PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)

Papi-cont.

- PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)
  - *Low level interface*, che fornisce informazioni specifiche dell'hardware a disposizione per indagini maggiormente sofisticate. Molto più complessa da usare.

Papi-cont.

- PAPI fornisce un'interfaccia agli *Hardware Counters* essenzialmente di due tipi:
  - *High level interface*, un insieme di routines per accedere ad una lista predefinita di eventi (*PAPI Preset Events*)
  - *Low level interface*, che fornisce informazioni specifiche dell'hardware a disposizione per indagini maggiormente sofisticate. Molto più complessa da usare.
- Occorre verificare il numero di *Hardware Counters* disponibili sulla macchina. Questo numero ci fornisce la misura del numero di eventi che possono essere monitorati in contemporanea.

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point
  - PAPI_L1_DCA - accessi in cache L1

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point
  - PAPI_L1_DCA - accessi in cache L1
  - PAPI_L1_DCM - cache misses L1

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point
  - PAPI_L1_DCA - accessi in cache L1
  - PAPI_L1_DCM - cache misses L1
  - PAPI_SR_INS - istruzioni di store

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point
  - PAPI_L1_DCA - accessi in cache L1
  - PAPI_L1_DCM - cache misses L1
  - PAPI_SR_INS - istruzioni di store
  - PAPI_TLB_DM - TLB misses

PAPI Preset Events

- Un insieme di eventi di particolare interesse per un "tuning" delle prestazioni
- PAPI definisce circa un centinaio di *Preset Events* per una data CPU che, generalmente, ne implementerà un sottoinsieme. Vediamone qualcuno dei più importanti:
  - PAPI_TOT_CYC - numero di cicli totali
  - PAPI_TOT_INS - numero di istruzioni completate
  - PAPI_FP_INS - istruzioni floating-point
  - PAPI_L1_DCA - accessi in cache L1
  - PAPI_L1_DCM - cache misses L1
  - PAPI_SR_INS - istruzioni di store
  - PAPI_TLB_DM - TLB misses
  - PAPI_BR_MSP - conditional branch mispredicted

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time
  - PAPI_accum_counters

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters
  - PAPI_start_counters - start counting hw events

PAPI:le funzioni di alto livello

- Un piccolo insieme di routines che servono a "instrumentare" un codice. Le funzioni sono:
  - PAPI_num_counters - ritorna il numero di hw counters disponibili
  - PAPI_flips - floating point instruction rate
  - PAPI_flops - floating point operation rate
  - PAPI_ipc - instructions per cycle e time
  - PAPI_accum_counters
  - PAPI_read_counters - read and reset counters
  - PAPI_start_counters - start counting hw events
  - PAPI_stop_counters - stop counters return current counts

Papi: interfaccia C e Fortran

- Le chiamate alla libreria di alto livello sono piuttosto intuitive.
- Sebbene PAPI sia scritto in C è possibile chiamare le funzioni di libreria anche da codici Fortran.
- Un esempio in Fortran:

```
#include "fpapi_test.h"
...  ; integer events(2), retval   ;   integer*8 values(2)
...  ;
events(1) = PAPI_FP_INS     ;   events(2) = PAPI_L1_DCM
...
        call PAPIf_start_counters(events, 2, retval)
        call PAPIf_read_counters(values, 2, retval)  ! Clear values
             [sezione di codice da monitorare]
        call PAPIfstop_counters(values, 2, retval)
           print*,'Floating point instructions:  ',values(1)
           print*,' L1 Data Cache Misses:   ',values(2)
...
```

Papi: interfaccia C e Fortran

un esempio in C:

```
%\begin{lstlisting}
#include <stdio.h>
#include <stdlib.h>
#include "papi.h"

#define NUM_EVENTS 2
#define THRESHOLD 10000
#define ERROR_RETURN(retval) { fprintf(stderr, "Error %d %s:line %d: \n",
retval,__FILE__,__LINE__);  exit(retval); }
...
/* stupid codes to be monitored */
void computation_mult()

 ....
int main()
{
   int Events[2] = {PAPI_TOT_INS, PAPI_TOT_CYC};
   long long values[NUM_EVENTS];
...
if ( (retval = PAPI_start_counters(Events, NUM_EVENTS)) != PAPI_OK)
       ERROR_RETURN(retval);
   printf("\nCounter Started: \n");
   computation_add();
...
if ( (retval=PAPI_read_counters(values, NUM_EVENTS)) != PAPI_OK)
      ERROR_RETURN(retval);
   printf("Read successfully\n");
   printf("The total instructions executed for addition are %lld \n",values[0]);
   printf("The total cycles used are %lld \n", values[1] );
...
```

PAPI: Esercitazione

- Scaricare papi dal sito: http://icl.cs.utk.edu/papi/
- Installarla sul proprio PC
- Inserire le direttive PAPI all'interno del codice prodotto matrice-matrice
- Verificare i risultati ottenuti

Outline

- Debugging
- PAPI
- Parallel profiling
  - Scalasca
- Parallel debugging

Outline

Debugging

PAPI

Parallel profiling
- Scalasca

Parallel debugging

Scalasca: caratteristiche

- Tool sviluppato da Felix Wolf del Juelich Supercomputing Centre e collaboratori.
- In realtà nasce come il successore di un altro tool di successo (KOJAK)
- È il Toolset di riferimento for la "scalable" "performance analysis" di large-scale parallel applications (MPI & OpenMP).
- Utilizzabile sulla maggior parte dei sistemi High Performance Computing (HPC) con decine di migliaia di "cores"....
- ...ma anche su architetture parallele "medium-size"
- È un prodotto "open-source", continuamente aggiornato e mantenuto da Juelich.
- Il sito: `www.scalasca.org`

Scalasca: caratteristiche

- Supporta applicazioni scritte in Fortran, C e C++.
- In pratica, Scalasca consente due tipi di analisi:
  - una modalità "summary" che consente di ottenere informazioni aggregate per la nostra applicazione (ma sempre dettagliate a livello di singola istruzione)
  - una modalità "tracing" che è "process–local" e che consente di raccogliere molte più informazioni ma che può risultare particolarmente onerosa dal punto di vista dell'uso di risorse di "storage"
- Dopo l'esecuzione dell'eseguibile (instrumentato) Scalasca è in grado di caricare i files in memoria ed analizzarli in parallelo usando lo stesso numero di "cores" della nostra applicazione.
- Il codice può essere instrumentato sia automaticamente dal compilatore che manualmente dall'utente

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

Scalasca: istruzioni per l’uso

L’intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  ifort -openmp [altre_opzioni] <codice_sorgente>

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>

Scalasca: istruzioni per l’uso

L’intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  mpif90 [opzioni] <codice_sorgente>

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>

Scalasca: istruzioni per l’uso

L’intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  <codice_eseguibile>

Scalasca: istruzioni per l’uso

L’intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  scalasca -analyze [opzioni_scalasca] <codice_eseguibile>

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
  mpirun [opzioni] <codice_eseguibile>

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
  scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
  scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>

  Viene creata una directory epik_[caratteristiche]

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
  scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>
  Viene creata una directory epik_[caratteristiche]
- Analisi risultati:

Scalasca: istruzioni per l'uso

L'intero processo avviene in tre passi:

- Compilazione (il codice sorgente viene "instrumentato"):
  scalasca -instrument [opzioni_scalasca] ifort -openmp [altre_opzioni] <codice_sorgente>
  scalasca -instrument [opzioni_scalasca] mpif90 [opzioni] <codice_sorgente>
- Esecuzione:
  scalasca -analyze [opzioni_scalasca] <codice_eseguibile>
  scalasca -analyze [opzioni_scalasca] mpirun [opzioni] <codice_eseguibile>
  Viene creata una directory epik_[caratteristiche]
- Analisi risultati:
  scalasca -examine [opzioni_scalasca] epik_[caratteristiche]

Scalasca: un esempio

- Codice sismologia elementi finiti (fem.F).
- Parallelizzato utilizzando OpenMP.
- Eseguito su 8 processori (parallelismo "moderato") su singolo nodo
- Compilatore intel (comando ifort -O3 -free -openmp...)
- Alcuni numeri:
  - Numeri dei nodi della griglia 2060198.
  - Dimensione della matrice uguale al doppio del numero dei nodi di griglia (4120396).
  - Numeri degli elementi non nulli della matrice 57638104.

Scalasca: compilazione ed esecuzione


```
[ruggiero@neo258 SRC]$ scalasca -instrument -user ifort -O3 -free -openmp  *.F
```

Scalasca: compilazione ed esecuzione


```
[ruggiero@neo258 SRC]$ scalasca -instrument -user ifort -O3 -free -openmp  *.F
```

```
[ruggiero@neo258 EXE]$ scalasca -analyze ./fem.x
```

Scalasca: compilazione ed esecuzione


```
[ruggiero@neo258 SRC]$ scalasca -instrument -user ifort -O3 -free -openmp  *.F
```

```
[ruggiero@neo258 EXE]$ scalasca -analyze ./fem.x
```

```
S=C=A=N: Scalasca 1.2.2 runtime summarization
S=C=A=N: Abort: measurement blocked by existing archive ./epik_fem_Ox8_sum
[ruggiero@neo258 EXE]$ rm -r epik_fem_Ox8_sum/
[ruggiero@neo258 EXE]$ scalasca -analyze ./fem.x
S=C=A=N: Scalasca 1.2.2 runtime summarization
S=C=A=N: ./epik_fem_Ox8_sum experiment archive
S=C=A=N: Thu Jan  7 16:03:56 2010: Collect start
./fem.x
[.0]EPIK: Activated ./epik_fem_Ox8_sum [NO TRACE]
 tri   25.6178519725800       seconds
S=C=A=N: Thu Jan  7 16:04:23 2010: Collect done (status=130) 27s
Abort: incomplete experiment ./epik_fem_Ox8_sum
.....
```

Scalasca: analisi dei risultati


```
[ruggiero@neo258 EXE]$ scalasca -examine -s ./epik_fem_Ox8_sum/
```

Scalasca: analisi dei risultati


```
[ruggiero@neo258 EXE]$ scalasca -examine -s ./epik_fem_Ox8_sum/
```

```
cube3_score ./epik_fem_Ox8_sum/epitome.cube
Reading epik_fem_Ox8_sum/epitome.cube... done.
            .....
Estimated aggregate size of event trace (total_tbc): 1102182744 bytes
Estimated size of largest process trace (max_tbc):   1090211280 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate flushes
 or reduce requirements using file listing names of USR regions to be filtered.)

flt  type     max_tbc        time       % region
      ANY  1090211280     6142.73  100.00 (summary) ALL
      OMP     1684464     4324.02   70.39 (summary) OMP
      COM       17184      385.86    6.28 (summary) COM
      USR  1088536224     1432.86   23.33 (summary) USR
```

Scalasca output: descrizione

1. Per tipologia:

Scalasca output: descrizione

1. Per tipologia:
   - ANY: tutte le routines che compongono il programma

Scalasca output: descrizione

1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).

Scalasca output: descrizione

1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
   - COM:tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.

Scalasca output: descrizione

1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
   - COM:tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
   - USR: quelle che sono coinvolte in operazioni puramente locali al processo.

Scalasca output: descrizione

1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
   - COM:tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
   - USR: quelle che sono coinvolte in operazioni puramente locali al processo.
2. La massima capacità del trace-buffer richiesta (misurata in in bytes).

Scalasca output: descrizione

1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
   - COM:tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
   - USR: quelle che sono coinvolte in operazioni puramente locali al processo.
2. La massima capacità del trace-buffer richiesta (misurata in in bytes).
3. Il tempo impiegato (in secondi) per l'esecuzione di quella parte di codice.

Scalasca output: descrizione

1. Per tipologia:
   - ANY: tutte le routines che compongono il programma
   - OMP: quelle che contengono costrutti di parallellizazione (OpenMP in questo caso, altrimenti MPI o anche entrambi).
   - COM:tutte quelle routines che interagiscono con quelle che contengono istruzioni di parallelizzazione.
   - USR: quelle che sono coinvolte in operazioni puramente locali al processo.
2. La massima capacità del trace-buffer richiesta (misurata in in bytes).
3. Il tempo impiegato (in secondi) per l'esecuzione di quella parte di codice.
4. La percentuale del tempo impiegato, rispetto a quello totale, per la sua esecuzione.

Scalasca output: maggior dettaglio


```
[ruggiero@neo258 EXE]$ cube3_score -r epik_fem_Ox8_sum/summary.cube.gz
```

Scalasca output: maggior dettaglio


```
[ruggiero@neo258 EXE]$ cube3_score -r epik_fem_0x8_sum/summary.cube.gz
```

```
...
      USR   889293768       68.51    1.12 expand_
      USR    98889504        9.96    0.16 ordinamento_
      USR    98747208       12.60    0.21 elem2d_
      USR      730224        0.05    0.00 elem_ij_
      OMP      349200        0.48    0.01 !$omp do @solutore_parallelo.F:157
      OMP      349200        0.34    0.01 !$omp ibarrier @solutore_parallelo.F:163
      USR      171840        0.03    0.00 dwalltime00_
      USR      171840        0.05    0.00 dwalltime00
      USR      142224        0.01    0.00 abc03_bis_
      USR      142224        0.01    0.00 abc03_ter_
      USR       85896        1.39    0.02 printtime_
      USR       85896        0.02    0.00 inittime_
      OMP       51480       19.53    0.32 !$omp ibarrier @fem.F:2554
      OMP       51480      196.73    3.20 !$omp do @fem.F:2548
      OMP       51480       88.14    1.43 !$omp ibarrier @fem.F:2540
      OMP       51480     1555.91   25.33 !$omp do @fem.F:2532
      OMP       34320       18.10    0.29 !$omp ibarrier @fem.F:2742
      OMP       31460        0.27    0.00 !$omp parallel @fem.F:2511
      OMP       31460        0.17    0.00 !$omp parallel @fem.F:2725
      OMP       31460        0.41    0.01 !$omp parallel @fem.F:2589
      OMP       31460        0.38    0.01 !$omp parallel @solutore_parallelo.F:40
....
```

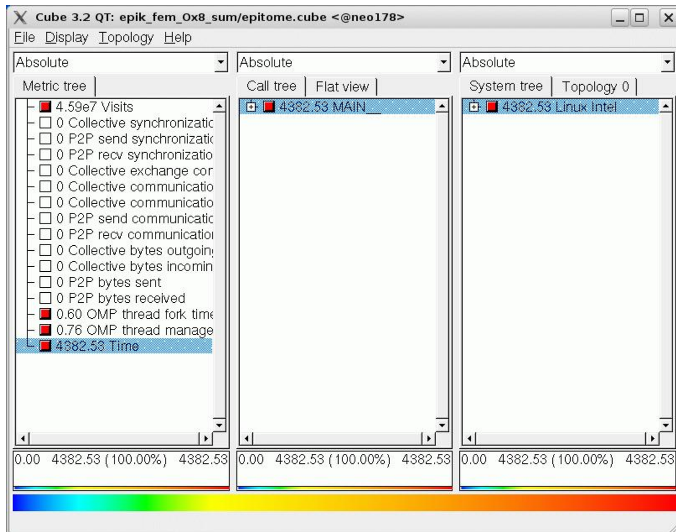
Instrumentazione manuale


```
        program fem
        implicit none
#include "epik_user.inc"
...
...
...
EPIK_USER_REG(r_write, "<<write>>")
        real*8,allocatable :: csi(:),eta(:)
...
...
EPIK_USER_START(r_write)
         do i=1,13
            write(i+5000,*) t,csi(2*p(i)-1),eta(2*p(i)-1)
            write(i+6000,*) t,csi(2*p(i)),eta(2*p(i))
         end do
EPIK_USER_END(r_write)
...
...
...
         end
```

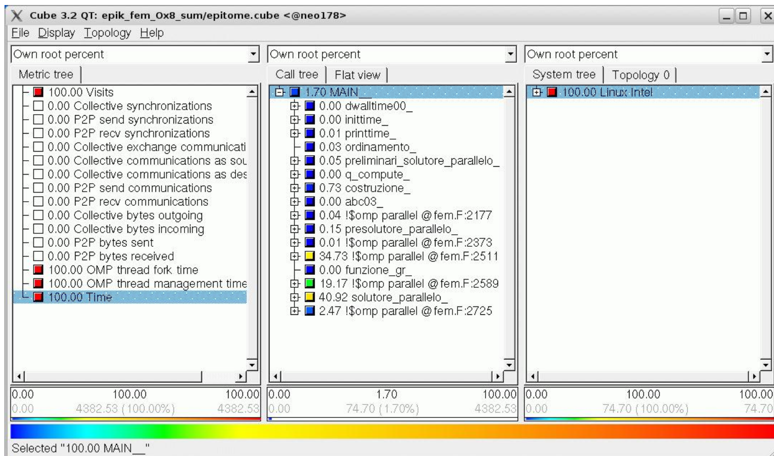
Scalasca output: maggior dettaglio - 2


```
OMP        23280        0.21     0.00 !$omp ibarrier @solutore_parallelo.F:86
OMP        23280        0.04     0.00 !$omp single @solutore_parallelo.F:84
OMP        23280       53.29     0.87 !$omp do @solutore_parallelo.F:89
OMP        23280        4.37     0.07 !$omp ibarrier @solutore_parallelo.F:99
OMP        23280      899.44    14.64 !$omp do @solutore_parallelo.F:104
USR        23280      931.42    15.16 sol_
OMP        23280      106.11     1.73 !$omp ibarrier @solutore_parallelo.F:133
OMP        23280       41.25     0.67 !$omp do @solutore_parallelo.F:142
OMP        23280        2.67     0.04 !$omp ibarrier @solutore_parallelo.F:150
OMP        23280        0.05     0.00 !$omp single @solutore_parallelo.F:172
OMP        23280        0.73     0.01 !$omp ibarrier @solutore_parallelo.F:174
USR        17160        3.89     0.06 <<sint>>
OMP        17160        1.43     0.02 !$omp do @solutore_parallelo.F:42
OMP        17160       64.71     1.05 !$omp do @solutore_parallelo.F:47
OMP        17160        7.68     0.12 !$omp ibarrier @solutore_parallelo.F:55
OMP        17160       36.40     0.59 !$omp do @solutore_parallelo.F:56
OMP        17160        6.45     0.11 !$omp ibarrier @solutore_parallelo.F:66
OMP        17160       15.30     0.25 !$omp do @solutore_parallelo.F:67
USR        17160        3.08     0.05 <<write>>
OMP        17160        3.18     0.05 !$omp ibarrier @solutore_parallelo.F:81
OMP        17160       95.11     1.55 !$omp do @fem.F:2726
OMP        17160        0.03     0.00 !$omp ibarrier @solutore_parallelo.F:182
OMP        17160        0.84     0.01 !$omp ibarrier @solutore_parallelo.F:180
OMP        17160        0.01     0.00 !$omp single @solutore_parallelo.F:178
```

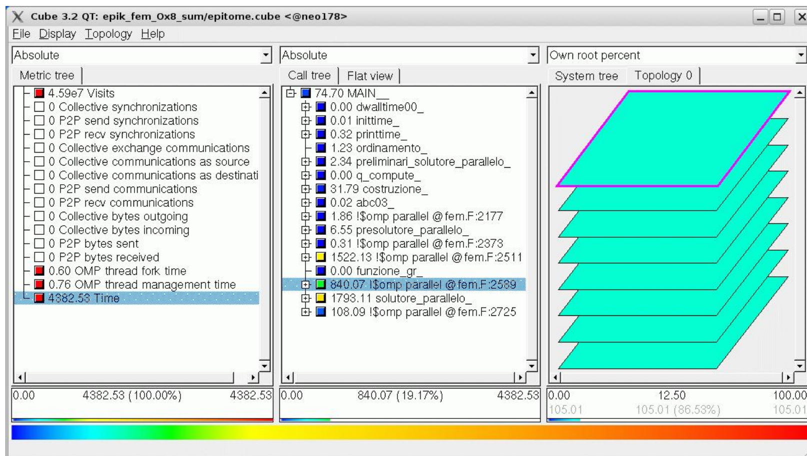
Scalasca: output grafico



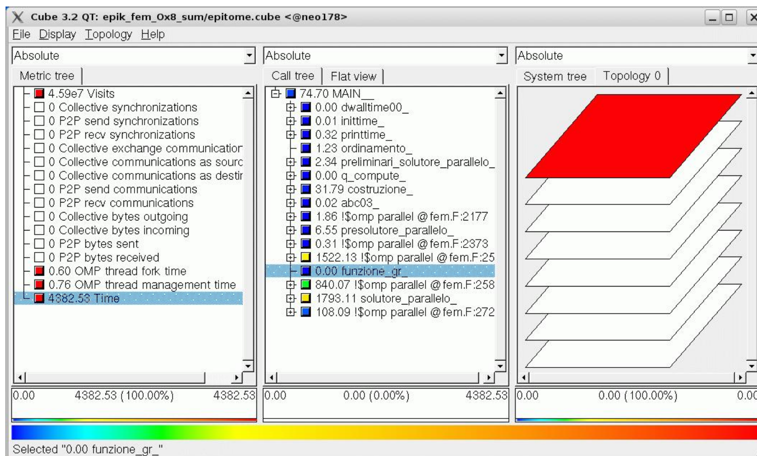
Scalasca: output grafico



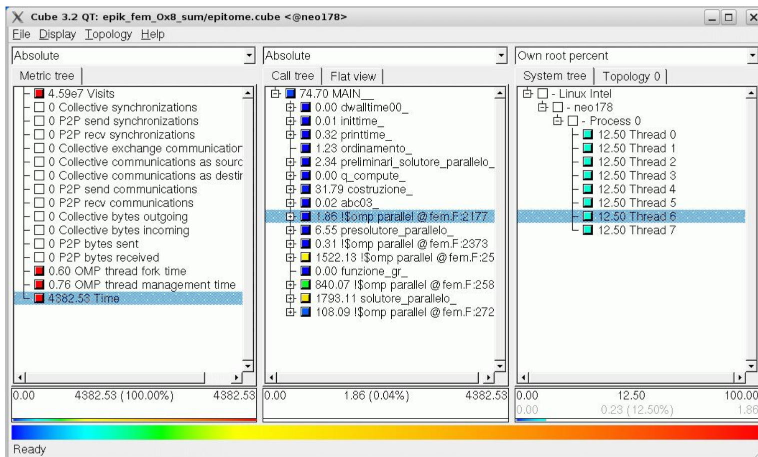
Scalasca: output grafico



Scalasca: output grafico



Scalasca: output grafico



Esercitazione:scalasca


```
https://hpc-forge.cineca.it/files/CoursesDev/public/2014/
Introduction_to_HPC_Scientific_Programming:_tools_and_techniques/
Rome/scalasca_esercizi.tar

tar xvf scalasca_esercizi.tar
```

- Codice prodotto matrice matrice scritto in C.
- Versione parallela MPI-OpenMP
- La parallelizzazione OpenMP implementata in 4 modi diversi, la scelta avviene con un'opportuna flag di compilazione condizionale all'interno del file *Makefile.mod2am.inc*
- la dimensione della matrice viene scelta nel file *mod2am.in*

Outline

Debugging

PAPI

Parallel profiling

Parallel debugging
- Marmot
- Totalview

Outline

Debugging

PAPI

Parallel profiling

Parallel debugging
- Marmot
- Totalview

Marmot

- It is a library written in C++, which has to be linked to your application (Fortran,C,C++) in addition to the existing MPI library
- It will check whether your application conforms to the MPI standard and will issue warnings if there are errors or non-portable constructs.
- You do not need to modify your source code, you only need one additional process working as debug server.
- It supports hybrid programs (MPI-OpenMP).
- The output is a human-readable text file, an HTML file, or uses a format that allows display in other tools, e.g. Cube.
- The tool can be configured via environment variables.
- www.hlrs.de/organization/av/amt/projects/marmot

Compilation and running


```
marmot(f77,f90,cc,cxx) [option]  file_source -o file_exe
```

MARMOT DEBUG MODE	0: errors 1: errors and warnings 2: errors, warnings and remarks are reported (default)
MARMOT LOGFILE TYPE	0: ASCII Logging (default) 1: HTML Logging 2: CUBE Logging (when enabled via configure)
MARMOT LOG FILTER COUNT	Limits how often a sepecific problem is recoreded (default: 2)
MARMOT LOG FLUSH TYPE	0: Flush on Error (default) 1: Immediate Flush
MARMOT SERIALIZE	0: code is not serialized 1: code is serialized (default)
....	....

```
mpirun -np n+1 file_exe
```

Example


```
1   program mpi_bug1
2         include 'mpif.h'
3
4         integer numtasks, rank, dest, source, count, tag, ierr
5         integer stat(MPI_STATUS_SIZE)
6         character inmsg, outmsg
7         outmsg = 'x'
8
9         call MPI_INIT(ierr)
10        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
11        call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
12        print *, 'Task',rank,'starting...'
13
14        if (rank .eq. 0) then
15          if (numtasks .gt. 2) then
16            print *, 'Numtasks=',numtasks,'. Only 2 needed.'
17            print *, 'Ignoring extra...'
18          endif
19          dest = rank + 1
20          source = dest
21          tag = rank
22              call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
23       &                 MPI_COMM_WORLD, ierr)
24        print *, 'Sent to task',dest
25          call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
26       &                 MPI_COMM_WORLD, stat, ierr)
27        print *, 'Received from task',source
```

Example


```
28        else if (rank .eq. 1) then
29          dest = rank - 1
30          source = dest
31          tag = rank
32          call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
33       &            MPI_COMM_WORLD, stat, err)
34        print *, 'Received from task',source
35          call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
36       &            MPI_COMM_WORLD, err)
37        print *, 'Sent to task',dest
38        endif
39         if (rank .le. 1) then
40          call MPI_GET_COUNT(stat, MPI_CHARACTER, count, ierr)
41          print *, 'Task ',rank,': Received', count, 'char(s) from task',
42       &          stat(MPI_SOURCE), 'with tag',stat(MPI_TAG)
43        endif
44
45        call MPI_FINALIZE(ierr)
46
47        end
```

Output

Timestamp	Rank	Thread	Type	Message	Location	MPI-Standard Reference
0	Global	0	Information	Text: The following entries describe all the Environmental variables.		
				Text: MARMOT_DEBUG_MODE = 2		
						Standard
9	Global	0	**Error**	Text: WARNING: all clients are pending! Last calls (max. 10) on node 0: timestamp 2: MPI_INIT(ierror) mpi_bug1.f line: 17 timestamp 4: MPI_COMM_RANK(comm = MPI_COMM_WORLD, *rank, ierror) mpi_bug1.f line: 18 timestamp 5: MPI_COMM_SIZE(comm = MPI_COMM_WORLD, *size, ierror) mpi_bug1.f line: 19 timestamp 7: MPI_SEND(*buf, count = 1, datatype = MPI_CHARACTER, dest = 1, tag = 0, comm MPI_COMM_WORLD, ierror) m line: 30 timestamp 9: MPI_RECV(*buf, count = 1, datatype = MPI_CHARACTER, source = 1, tag = 0, comm = MPI_COMM_WORLD, *status, ierror) m line: 33 Last calls (max. 10) on node 1: timestamp 1: MPI_INIT(ierror) mpi_bug1.f line: 17 timestamp 3: MPI_COMM_RANK(comm = MPI_COMM_WORLD, *rank, ierror) mpi_bug1.f line: 18 timestamp 6: MPI_COMM_SIZE(comm = MPI_COMM_WORLD, *size, ierror) mpi_bug1.f line: 19 timestamp 8: MPI_RECV(*buf, count = 1, datatype = MPI_CHARACTER, source = 0, tag = 1, comm = MPI_COMM_WORLD, *status, ierror) m line: 41		Infos see MPI-Standard

Outline

- Debugging
- PAPI
- Parallel profiling
- Parallel debugging
  - Marmot
  - Totalview

Totalview

- 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 Tv

- 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.

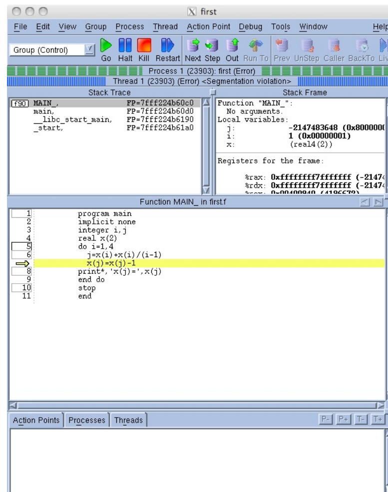
```
gcc [option]  -g file_source.c -o filename
```

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.



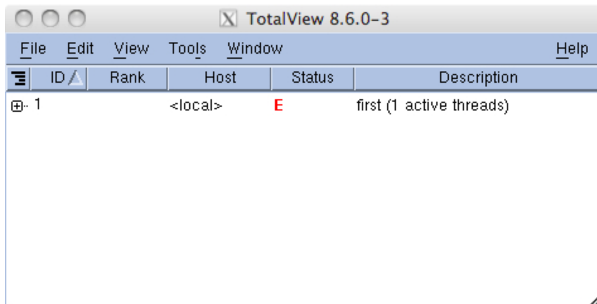
Totalview: Action points

- 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

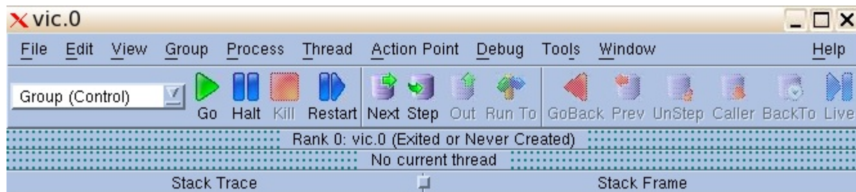
- Breakpoint
  - Right click on a source line → Set breakpoint
  - Click on the line number
- Watchpoint
  - Right click on a variable → Create watchpoint
- Barrier point
  - Right click on a source line → Set barrier
- Edit action point property
  - Rigth click on a action point in the Action Points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

Totalview: Status



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.