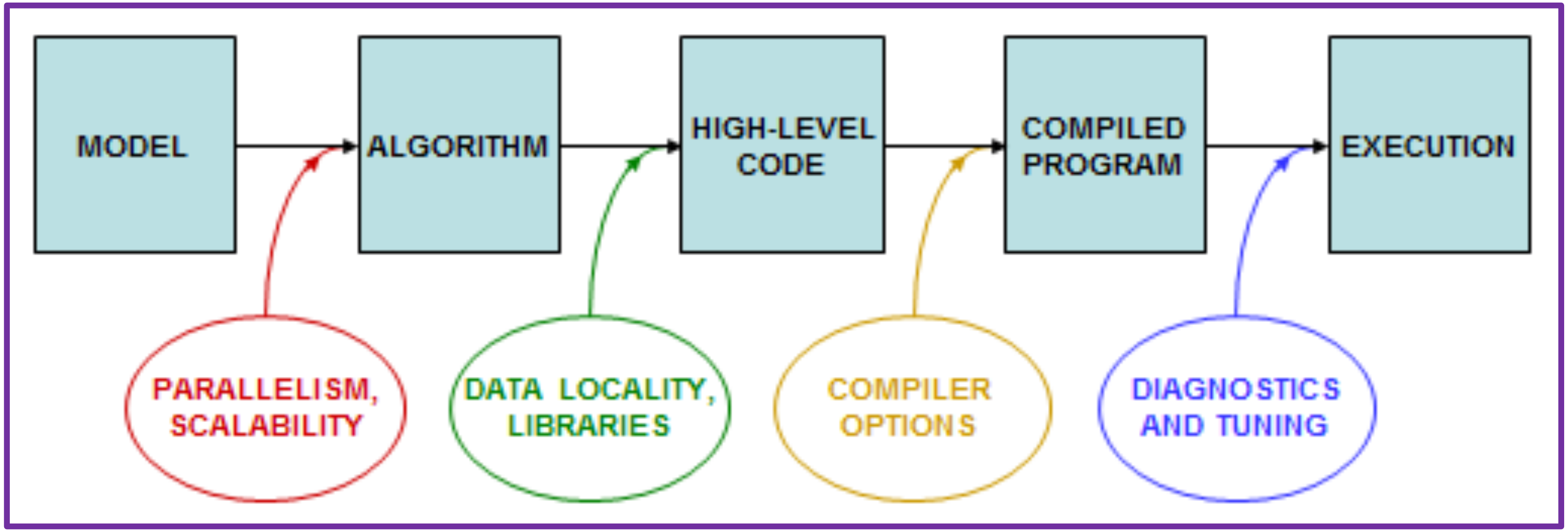
MAE 5032 High Performance Computing: Methods and Practices

Lecture 9: Code profiling

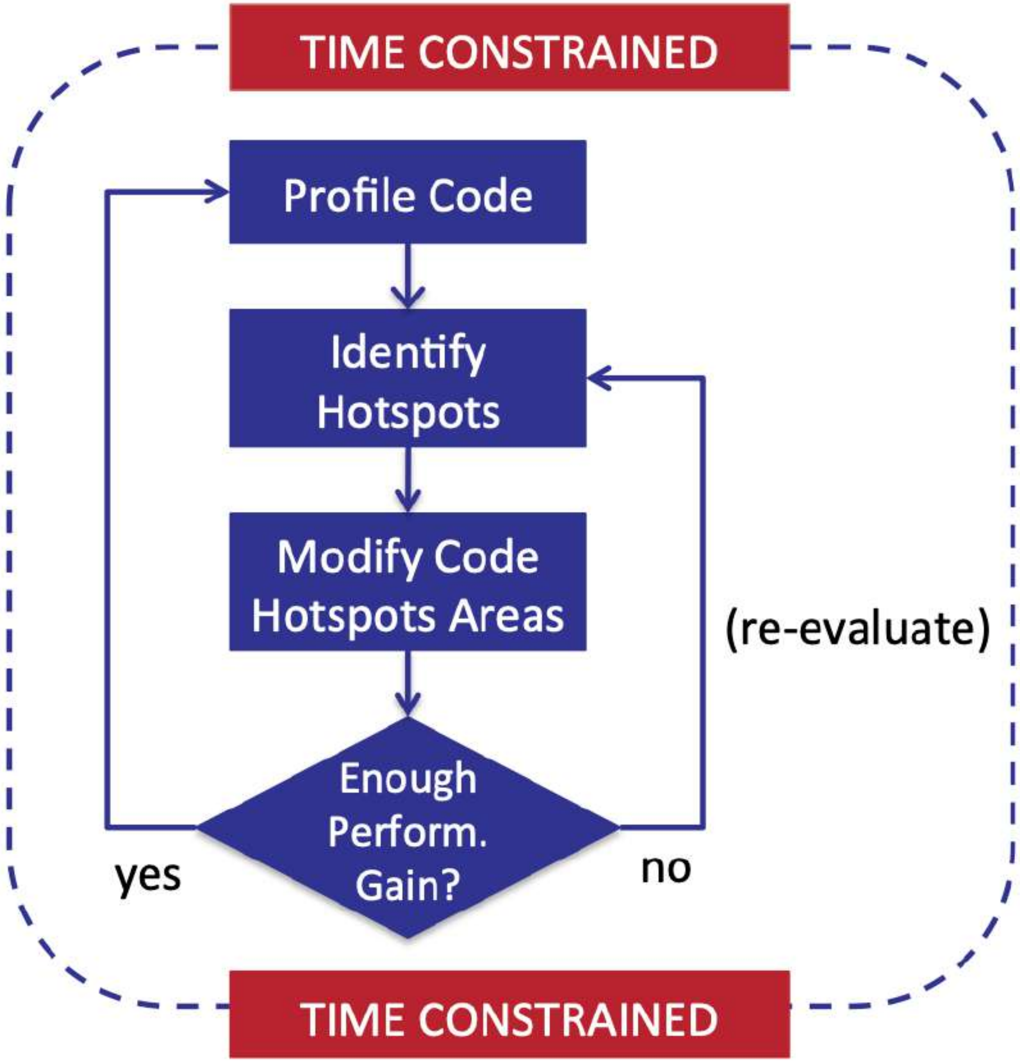
Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn

Motivation

- Profiling focuses on characterizing the performance of an application.
- It measure performance characteristics, helps identify areas for performance improvement.



Iterative process



Introduction

Two main approaches to profiling an application

- **Instrumentation**
  - adds instructions to an application to collect information (function call duration, number of invocations, etc.)
  - It alters the execution of a program
  - It may degrade the performance as a hole while being profiled

- **Statistical sampling**
  - Query the state of unmodified executable at regular intervals.
  - Less comprehensive set of measurements.
  - Usually does not degrade the performance of an application

Timers

- Basic timing can be obtained by measuring the entire run time of a code.
- The command time is available in Unix systems.
- The time command gives the total run time of its argument process in seconds.

Total time elapsed from execution to termination. It is also known as the wall clock time.

Total time spent by the CPU processing the instruction contained in the program. It noes not include the time when your program waits for service from the OS.

Total time spent by the CPU processing service requests (known as system calls) for your program from the OS. The system call often is I/O requesting disk, netwrok, or terminal access.

Timers

- C standard libraries provide a number of standard calls for getting timing data.
- It can be targeted to specific locations in the code rather than the whole application.

Routine	Type	Resolution (usec)	OS/Compiler
times	user, sys	1000	Linux, AIX, IRIX, UNICOS
getrusage	wall, user, sys	1000	Linux, AIX, IRIX
gettimeofday	wall	1	Linux, AIX, IRIX, UNICOS
rdtsc	wall	0.1	Linux
read_real_time	wall	0.001	AIX
system_clock	wall	(system dependent)	Fortran 90 intrinsic
MPI_Wtime	wall	(system dependent)	MPI Library (C and Fortran)

Timers

- C standard libraries provide a number of standard calls for getting timing data.
- It can be targeted to specific locations in the code rather than the whole application.

```
starttime = times();

your code to be monitored

time_elapsed = times() - startime;
```

Timers

- C standard libraries provide a number of standard calls for getting timing data.
- It can be targeted to specific locations in the code rather than the whole application.

```
starttime = MPI_Wtime();

your code to be monitored

time_elapsed = MPI_Wtime() - startime;
```

GPROF

- GPROF is the GNU Project Profiler, which belongs to the GNU Binutils
- Requires recompilation of the code
- It is a form of **instrumented** profiling, as the compiler is adding profiling instructions to the resulting executable
- Provides three types of profiles
  - flat profile
  - call graph
  - annotated source

Types of profiles

- Flat profile
  - CPU time spent in each funciton
  - Number of times a function is called
  - Useful to identify most expensive routines

- Call graph
  - Number of times a funciton was called by other functions
  - Number of times a function called other funcitons
  - Useful to identify function relations
  - Suggestive of places where function calls could be eliminated

- Annotated source
  - Indicates number of times a line was executed

Types of profiles

- Use the –pg flag during compilation:

```
gcc -g -pg srcFile.c
icc -g -p srcFile.c
```

- Run the executable. By default, an output gmon.out file will be generated with the profiling information. It is readable by gprof.

- The profile data may be read and interpreted by running gprof.

```
gprof ./exeFile gmon.out > prifile.txt
gprof -A ./exeFile gmon.out > prifile_annotated.txt
```

Example 1: srcFile.c

```c
#include<stdio.h>

void new_func1(void)
{
  printf("\n Inside new_func1()\n");
    int i = 0;

    for(;i<0xffffffee;i++);
}

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;

    for(;i<0xffffffff;i++);
    new_func1();

    return;
}
```

```c
static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;

    for(;i<0xffffffaa;i++);
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;

    for(;i<0xffffff;i++);
    func1();
    func2();

    return 0;
}
```

Output from gprof: flat profile

The number of times the function was called

Average time spent per call

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 34.96      5.97     5.97        1     5.97    11.45  func1
 33.26     11.65     5.68        1     5.68     5.68  func2
 32.08     17.13     5.48        1     5.48     5.48  new_func1
  0.24     17.17     0.04                             main
```

Percentage of the total execution time your program spent in this function.

Cumulative total number of seconds spent executing this function, plus time spent in all functions above this one in this table

The time accounted for by this function alone

Output from gprof: call graph

each function has an index number

Percentage of the total time spent in this function

Total time spent in this fun. This should be identical to self seconds in flat profile.

Time spent in the subroutine calls made by this function. This should equal the sum of self and children in flat profile

Times the function was called

```
          Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.06% of 17.17 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.04   17.13                 main [1]
                5.97    5.48       1/1           func1 [2]
                5.68    0.00       1/1           func2 [3]
-----------------------------------------------
                5.97    5.48       1/1           main [1]
[2]     66.7    5.97    5.48       1         func1 [2]
                5.48    0.00       1/1           new_func1 [4]
-----------------------------------------------
                5.68    0.00       1/1           main [1]
[3]     33.1    5.68    0.00       1         func2 [3]
-----------------------------------------------
                5.48    0.00       1/1           func1 [2]
[4]     31.9    5.48    0.00       1         new_func1 [4]
-----------------------------------------------
```

Reference

GNU gprof:
https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_toc.html

Valgrind callgrind tool

- Valgrind has an instrumentation framework for code profiling named **callgrind**.
- The results can be visualized by Kcachegrind.
  https://kcachegrind.github.io/
  sudo apt-get install kcachegrind

Usage:

```
valgrind –tool=callgrind a.out [arguments]
```

- callgrind will run your program with instrumentation added.
- The run is considerably slower
- Thus, run a representative task that is small, if possible

Valgrind callgrind tool

```
juliu@Ladyzhenskaya:~/mae5032/week-10/gprof-02$ valgrind --tool=callgrind ./a.out
==5498== Callgrind, a call-graph generating cache profiler
==5498== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==5498== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5498== Command: ./a.out
==5498==
==5498== For interactive control, run 'callgrind_control -h'.

 Inside main()

 Inside func1

 Inside new_func1()

 Inside func2
==5498==
==5498== Events    : Ir
==5498== Collected : 47295173985
==5498==
==5498== I   refs:      47,295,173,985
```

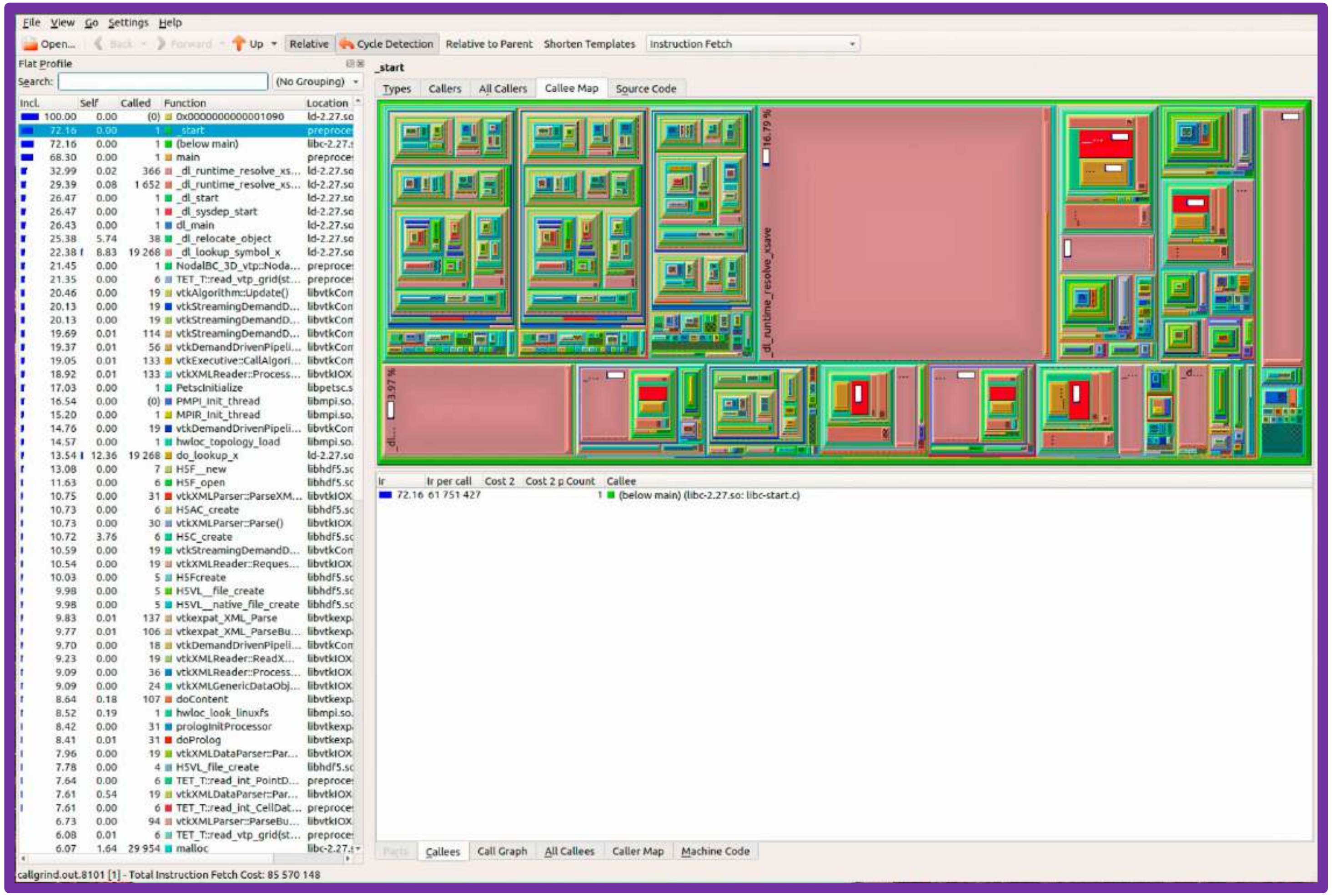
Valgrind callgrind tool

```
kcachegrind callgrind.out.12345
```

Time spent in each function.

Two cost metrics:

- incl. shows the total cost of a function;
- self shows the time spent in each function itself.



Other tools

- MPIP: lightweight scalable MPI profiling tool mpip.sourceforge.net

- IPM: Integrated Performance Monitoring for MPI scalability analysis ipm-hpc.sourceforge.net

- Tau: Suite of tuning and analysis utilities www.cs.uoregon.edu/research/tau

- Scalasca: Complete suit of tuning and analysis tools www.fz-juelich.de/jsc/scalasca

- PAPI: Performance Application Programming Interface icl.cs.utk.edu/papi
  - developed by J. Dongarra