Profiling & Debugging
CISC 879

Tristan Vanderbruggen & John Cavazos
Dept of Computer & Information Sciences
University of Delaware
Lecture Overview

• Profiling and Debugging
  o Why?
  o Tools
  o Data sets

• Race Condition and Deadlock
  o Task graph and allocation
  o Examples:
    • Race condition with OpenMP sections
    • Deadlock with OpenMP sections and lock
    • Correct OpenMP programs

• Communication Interlock with MPI
1.a - Why Profiling?

- "A program cannot be executed faster than its slowest part"

- Large applications can have multiple slow parts!
- **Bottleneck** are difficult to find.
- **Profiling tool** help to find where **most time spent**
1.b - Tools

• Gprof
• ValGrind
  o MemCheck
  o CallGrind
• VTune
1.b - Gprof: Compile Step

- gprof: GNU profiling
- Require specifically compile binary
  - Have the program working, NOT a debugger
  - Add the option `-pg` to `gcc` & `linker`

```bash
$> gcc -pg -c target.c -o target.o
$> link -pg target.o -o target_binary
```
1.b - Gprof: Run

- To get a profile of your application
  - Run the application as usual
    - `$> ./target_binary [application-options]`
  - It generates a file gmon.out, it is not readable
  - gprof creates a "human readable" profile
    - `$> gprof target_binary`
1.b - Gprof: output

Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>44.46</td>
<td>0.32</td>
<td>0.32</td>
<td>124114</td>
<td>0.00</td>
</tr>
<tr>
<td>30.56</td>
<td>0.54</td>
<td>0.22</td>
<td>6441</td>
<td>0.03</td>
</tr>
<tr>
<td>16.67</td>
<td>0.66</td>
<td>0.12</td>
<td>124114</td>
<td>0.00</td>
</tr>
</tbody>
</table>

... 

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.72</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td>0.00</td>
<td>0.57</td>
<td>1/1</td>
<td></td>
<td></td>
<td>getAlphaFromTrainSet [3]</td>
</tr>
<tr>
<td>0.02</td>
<td>0.06</td>
<td>24700/124114</td>
<td></td>
<td>polynomial [5]</td>
<td></td>
</tr>
</tbody>
</table>

... 

---

... 

--- 

0.10 0.16 3004/6441 takeStep [7] 
0.12 0.19 3437/6441 examineExample [2] 
[4] 78.7 0.22 0.35 6441 cal_learned_func [4] 
0.09 0.25 97839/124114 polynomial [5] 
0.00 0.00 195678/497706 fMallocHandle [10] 

---
1.b - ValGrind

- ValGrind is a classic dynamic analysis tool
  - Memory checking with **memcheck**
  - Cache analysis with **cachegrind**
  - Call graph with the tool **callgrind**
• `$> valgrind --tool=memcheck ./app$

• **Memory error detector**
  - Accessing memory you shouldn't
  - Using undefined values
  - Incorrect freeing of heap memory
  - Overlapping src and dst in memcpy
  - Memory leaks
• $>\text{valgrind --tool=memcheck ./app}$

• Memory error detector
  
  o Accessing memory you shouldn't
    ▪ overrunning and underrunning heap blocks
    ▪ overrunning the top of the stack
    ▪ accessing memory after it has been freed
  
  o Using undefined values
  
  o Incorrect freeing of heap memory
  
  o Overlapping src and dst in memcpy
  
  o Memory leaks
1.b - ValGrind - MemCheck

- `$> valgrind --tool=memcheck ./app`
- Memory error detector
  - Accessing memory you shouldn't
  - Using undefined values
    - values that have not been initialised
    - derived from other undefined values
  - Incorrect freeing of heap memory
  - Overlapping src and dst in memcpy
  - Memory leaks
• $> \text{valgrind --tool=memcheck ./app}

• Memory error detector
  o Accessing memory you shouldn't
  o Using undefined values
  o Incorrect freeing of heap memory
    ▪ double-freeing heap blocks
    ▪ mismatched use of malloc/new/new[] versus free/delete/delete[]
  o Overlapping src and dst in memcpy
  o Memory leaks
• $>\text{valgrind} \ --\text{tool} = \text{memcheck} \ \text{./app}$

• Memory error detector
  - Accessing memory you shouldn't
  - Using undefined values
  - Incorrect freeing of heap memory
  - Overlapping src and dst in memcpy
  - Memory leaks
• Using callgrind:
  ○ Compile with `-pg`
• Run the application with valgrind and callgrind
  ○ `$> valgrind --tool=callgrind binary`
• No ”human readable” output
• `KCacheGrind` provide visualization of the call graph
Demo time!
1.c - Using ValGrind on Parallel Apps

- Threads safe
  - no problem with OpenMP
- for MPI
  - mpirun launches valgrind, not the opposite:
    - `mpirun [mpi-opts] valgrind [valgrind-opt] ./app`
  - memcheck report large amount of false errors
    - an MPI wrapper exist for MemCheck
    - Reducing the number of false positive errors
Quick Overview of VTune™

- Performance Profiles
  - Hotspot
  - Hardware-Event Based Sampling (EBS)

- Locks and waits analysis

- Thread Profiling
  - Visualize thread interactions on timeline
  - Balance workloads

- Hardware event sampling

- Source code view
Example: Hotspot Analysis

Example: OpenMP Analysis

- Identifies where your application is waiting on synchronization objects or I/O operations.
- Discovers how these waits affect your application performance.

source: http://www.polyhedron.com/plain-VTune
Example: OpenMP Analysis

More detailed information

Time information
- Total, wait, spin, ...

Top waiting objects
- In this example, OMP Join has the highest wait time

source: http://www.polyhedron.com/plain-VTune
Example: OpenMP Analysis

You can see how effectively using processor cores and get insights how to change your programs to run faster.

- Each thread’s run and wait
- Transitions between threads

source: http://www.polyhedron.com/plain-VTune
Example: MPI Analysis

- **Use** `amplex-cl` command to collect data and post-process results
  - Individual result directory will be created for each spawned process

- **Example**
  - `> mpiexec -n 4 amplex-cl -r my_result -collect hotspot -- my_app [my_app options]`
  - This will create result directory for 4 processes named `my_result.1` to `my_result.4`

- **Using GUI viewer to analyze each result**

  source: [http://www.polyhedron.com/plain-VTune](http://www.polyhedron.com/plain-VTune)
Example: MPI Analysis

OpenMP regions with routines called inside

OpenMP threads shown together with MPI (dapl) service threads

Future Work for Vtune™

• Upcoming updates
  o Will be part of Intel® Cluster Studio XE
  o Will be supported on clusters

1.d - Data Sets

• Without **good data set**: results are **meaningless**

• 4 criteriums:
  - **Not too small**: initialization
  - **Not too big**: profiling take longer
  - **Good coverage**: irregular code may need multiple dataset to show all the bottleneck
  - **Generic**: special cases are **always** dangerous
Common Parallelism Bugs
The following task graph:
• initializes two data-structures A and B
• then sets A using B, and B using A
2.a - OpenMP sections

/* Fork a team of threads */
#pragma omp parallel shared(a, b)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        {
            // init a
            // write b, read a
        }
    }
    #pragma omp section
    {
        // init b
        // write a, read b
    }
}
• This code could:
  o Write in a or b before they are fully initialize
    ▪ What if some memory have yet not been allocated.
  o It is a Race Condition
2.a - Race Condition

**Correct behavior**
- Task 1
- Shared data
- Task 2

- Task 2 gets the updated value from task 1

**Incorrect behavior**
- Task 1
- Shared data
- Task 2

- Task 1 and task 2 work on the same data
- Update from task 2 gets overwritten by task 1
• We have to add some synchronisations
  o We will use *locks*...
2.b - OpenMP locks

- a lock is a synchronization object
  - It can be set by one thread only
    - becoming owner of the lock
  - Only the owner can release a lock
  - Don't forget to initialize and destroy the locks
2.b - OpenMP locks

• API:
  - decl: `omp_lock_t lock;`
  - init: `void omp_init_lock(&lock);`
  - set: `void omp_set_lock(&lock);`
  - unset: `void omp_unset_lock(&lock);`
  - destroy: `void omp_destroy_lock(&lock);`
  - test: `int omp_test_lock(&lock);`
2.b - OpenMP locks

```c
#pragma omp parallel shared(a, b, locka, lockb)
{
    #pragma omp sections nowait

    #pragma omp section
    {
        omp_set_lock(&locka);
        // init a
        omp_set_lock(&lockb);
        // write b, read a
        omp_unset_lock(&lockb);
        omp_unset_lock(&locka);
    }

    #pragma omp section
    {
        omp_set_lock(&lockb);
        // init b
        omp_set_lock(&locka);
        // write a, read b
        omp_unset_lock(&locka);
        omp_unset_lock(&lockb);
    }
```
2.b - Deadlock
2.c - OpenMP Race Condition

```c
#pragma omp parallel shared(a, b, locka, lockb)
{
    #pragma omp sections nowait

#pragma omp section
{
    omp_set_lock(&locka);
    // init a
    omp_unset_lock(&locka);
    omp_set_lock(&lockb);
    // write b, read a
    omp_unset_lock(&lockb);
}
```

```c
#pragma omp section
{
    omp_set_lock(&lockb);
    // init b
    omp_unset_lock(&lockb);
    omp_set_lock(&locka);
    // write a, read b
    omp_unset_lock(&locka);
}
```
We want to build a communication ring with MPI
3.a - Linear Chain

```
if (rank > 0)
    MPI_Send(&out_msg, 1, MPI_CHAR, rank - 1, 0, MPI_COMM_WORLD);
if (rank < numtasks - 1)
    MPI_Recv(&in_msg, 1, MPI_CHAR, rank + 1, 0, MPI_COMM_WORLD, &status);
```
3.b - Ring with Interlock

prev = rank == 0 ? numtasks - 1 : rank - 1;
MPI_Send(&out_msg, 1, MPI_CHAR, prev, 0, MPI_COMM_WORLD);
next = rank == numtasks - 1 ? 0 : rank + 1;
MPI_Recv(&in_msg, 1, MPI_CHAR, next, 0, MPI_COMM_WORLD, &status);
prev = rank == 0 ? numtasks - 1 : rank - 1;
MPI_Isend(&out_msg, 1, MPI_CHAR, prev, 0, MPI_COMM_WORLD, &reqs[0]);
next = rank == numtasks - 1 ? 0 : rank + 1;
MPI_Irecv(&in_msg, 1, MPI_CHAR, next, 0, MPI_COMM_WORLD, &reqs[1]);

MPI_Waitall(2, reqs, status);
Links

- https://computing.llnl.gov/tutorials/mpi/
- https://computing.llnl.gov/tutorials/openMP/
  - Section on MPI