Announcements

- Homework 4 out today
- Dec 7th is the last day you can turn in Lab 4 and HW4, so plan ahead.
Thread level parallelism: Multi-Core Processors

- Two (or more) complete processors, fabricated on the same silicon chip
- Execute instructions from two (or more) programs/threads at the same time

IBM Power5
Multi-Cores are Everywhere

Intel Core Duo in new Macs: 2 x86 processors on same chip

XBox360: 3 PowerPC cores

Sony Playstation 3: Cell processor, an asymmetric multi-core with 9 cores (1 general-purpose, 8 special purpose SIMD processors)
Why Multi-cores Now?

- Number of transistors we can put on a chip growing exponentially...
... and performance growing too...

- But power is growing even faster!!
  - Power has become limiting factor in current chips
What is a Thread?

- What does Shared Memory imply?
- Machine model
As programmers, do we care?

- What happens if we run a program on a multi-core?

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 0 ; i < length ; ++i) {
        C[i] = A[i] + B[i];
    }
}
```
What if we want a program to run on both processors?

- We have to explicitly tell the machine exactly how to do this
  - This is called parallel programming or concurrent programming

- There are many parallel/concurrent programming models
  - We will look at a relatively simple one: fork-join parallelism
  - Posix threads and explicit synchronization (CSE451?)

See some usage!
Fork/Join Logical Example

1. Fork N-1 threads
2. Break work into N pieces (and do it)
3. Join (N-1) threads

void array_add(int A[], int B[], int C[], int length) {
    cpu_num = fork(N-1);
    int i;
    for (i = cpu_num; i < length; i += N) {
        C[i] = A[i] + B[i];
    }
    join();
}

How good is this with caches?
How does this help performance?

- Parallel **speedup** measures improvement from parallelization:

\[
\text{speedup}(p) = \frac{\text{time for best serial version}}{\text{time for version with } p \text{ processors}}
\]

- What can we realistically expect?
Reason #1: Amdahl’s Law

- In general, the whole computation is not (easily) parallelizable
Suppose a program takes 1 unit of time to execute serially
- A fraction of the program, \( s \), is inherently serial (unparallelizable)

For example, consider a program that, when executing on one processor, spends 10% of its time in a non-parallelizable region. How much faster will this program run on a 3-processor system?

\[
\text{New Execution Time} = \frac{1-s}{p} + s
\]

\[
\text{New Execution Time} = \frac{.9T}{3} + .1T = 0.3T + 0.17
\]

Speedup = \( \frac{1T}{0.47} = 2 \)

What is the maximum speedup from parallelization?

\[
S_p = \frac{1T}{0.17} = 10
\]
void array_add(int A[], int B[], int C[], int length) {
  cpu_num = fork(N-1);
  int i;
  for (i = cpu_num; i < length; i += N) {
    C[i] = A[i] + B[i];
  }
  join();
}

— Forking and joining is not instantaneous
  • Involves communicating between processors
  • May involve calls into the operating system
    — Depends on the implementation

\[
\text{New Execution Time} = \frac{1-s}{p} + s + \text{overhead}(P)
\]
Programming Explicit Thread-level Parallelism

- As noted previously, the programmer must specify how to parallelize.
- But, want path of least effort.

- Division of labor between the Human and the Compiler.
  - Humans: good at expressing parallelism, bad at bookkeeping.
  - Compilers: bad at finding parallelism, good at bookkeeping.

- Want a way to take serial code and say “Do this in parallel!” without:
  - Having to manage the synchronization between processors
  - Having to know a priori how many processors the system has
  - Deciding exactly which processor does what
  - Replicate the private state of each thread

- OpenMP: an industry standard set of compiler extensions.
  - Works very well for programs with structured parallelism.
Performance Optimization

- Until you are an expert, first write a working version of the program
- Then, and only then, begin tuning, first collecting data, and iterate
  - Otherwise, you will likely optimize what doesn’t matter

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” -- Sir Tony Hoare
Using tools to do instrumentation

- Two GNU tools integrated into the GCC C compiler

- **Gprof: The GNU profiler**
  - Compile with the `−pg` flag
    - This flag causes gcc to keep track of which pieces of source code correspond to which chunks of object code and links in a profiling signal handler.
  - Run as normal; program requests the operating system to periodically send it signals; the signal handler records what instruction was executing when the signal was received in a file called `gmon.out`
  - Display results using `gprof` command
    - Shows how much time is being spent in each function.
    - Shows the calling context (the path of function calls) to the hot spot.
### Example gprof output

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>calls</th>
<th>s/call</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81.89</td>
<td>4.16</td>
<td>4.16</td>
<td>37913758</td>
<td>0.00</td>
<td>0.00</td>
<td>cache_access</td>
</tr>
<tr>
<td>16.14</td>
<td>4.98</td>
<td>0.82</td>
<td>1</td>
<td>0.82</td>
<td>5.08</td>
<td>sim_main</td>
</tr>
<tr>
<td>1.38</td>
<td>5.05</td>
<td>0.07</td>
<td>6254582</td>
<td>0.00</td>
<td>0.00</td>
<td>update_way_list</td>
</tr>
<tr>
<td>0.59</td>
<td>5.08</td>
<td>0.03</td>
<td>1428644</td>
<td>0.00</td>
<td>0.00</td>
<td>dl1_access_fn</td>
</tr>
<tr>
<td>0.00</td>
<td>5.08</td>
<td>0.00</td>
<td>711226</td>
<td>0.00</td>
<td>0.00</td>
<td>dl2_access_fn</td>
</tr>
<tr>
<td>0.00</td>
<td>5.08</td>
<td>0.00</td>
<td>256830</td>
<td>0.00</td>
<td>0.00</td>
<td>yylex</td>
</tr>
</tbody>
</table>

Over 80% of time spent in one function

Provides calling context (main calls sim_main calls cache_access) of hot s

<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>0.82</td>
<td>4.26</td>
<td>1/1</td>
<td>main [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.82</td>
<td>4.26</td>
<td>1</td>
<td>sim_main [1]</td>
</tr>
</tbody>
</table>
|       | 4.18   | 0.07 | 36418454/36484188 |          | cache_access <cycle 1> [4
|       | 0.00   | 0.01 | 10/10    |        | sys_syscall [9]
|       | 0.00   | 0.00 | 2935/2967 |       | mem_translate [16]
|       | 0.00   | 0.00 | 2794/2824 |       | mem_newpage [18] |
Using tools for instrumentation (cont.)

- Gprof didn’t give us information on where in the function we were spending time. *(cache_access is a big function; still needle in haystack)*

- Gcov: the GNU coverage tool
  - Compile/link with the `-fprofile-arcs -ftest-coverage` options
    - Adds code during compilation to add counters to every control flow edge (much like our by hand instrumentation) to compute how frequently each block of code gets executed.
    - Run as normal
    - For each `xyz.c` file an `xyz.gdna` and `xyz.gcno` file are generated
  - Post-process with gcov `xyz.c`
    - Computes execution frequency of each line of code
    - Marks with `#####` any lines not executed
      - Useful for making sure that you tested your whole program
Example gcov output

Code never executed

```
14282656:  540:  if (cp->hsize) {
 541:      int hindex = CACHE_HASH(cp, tag);
 542:      for (blk=cp->sets[set].hash[hindex];
 543:            blk;
 544:            blk=blk->hash_next)
 546:          {
 547:              if (blk->tag == tag && (blk->status & CACHE_BLK_VALID)
 548:                  goto cache_hit;
 549:          }
 550:      } else {
 551:          /* linear search the way list */
 753030193:    552:          for (blk=cp->sets[set].way_head;
 553:              blk;
 554:                blk=blk->way_next)      {
 751950759:    555:              if (blk->tag == tag && (blk->status & CACHE_BLK_VALID)
 738747537:      goto cache_hit;
 556:          }
 557:      }
 558:  }
```

Loop executed over 50 iterations on average (751950759/14282656)
Summary

- Multi-core is having more than one processor on the same chip.
  - Soon most PCs/servers and game consoles will be multi-core
  - Results from Moore’s law and power constraint

- Exploiting multi-core requires parallel programming
  - Automatically extracting parallelism too hard for compiler, in general
  - But, can have compiler do much of the bookkeeping for us
  - OpenMP

- Fork-Join model of parallelism
  - At parallel region, fork a bunch of threads, do the work in parallel, and then join, continuing with just one thread
  - Expect a speedup of less than P on P processors
    - Amdahl’s Law: speedup limited by serial portion of program
    - Overhead: forking and joining are not free