Parallelization Primer

by
Christian Bienia
March 05, 2007
What is Parallelization?

Answer: The creation of a new algorithm!

- **Trivial case:** Run sequential algorithm on multiple CPUs, throw locks around shared data
- **Common case:** Rewrite & extend parts of sequential algorithm
- **Hard case:** Rewrite program from scratch
So how does it work?

1. Take Sequential Program
2. Find Hot Spot
3. Parallelize code
4. Find & Analyze Bottlenecks
5. Refinement
Metrics

Speedup $S_p = \frac{T_1}{T_p}$

Ideal Speedup $S_p = p$

Parallel Efficiency $E_p = \frac{S_p}{p}$
Preparation
Choosing a Sequential Program

• Not all programs can be parallelized
  - Example: Some cryptographic programs

• Must be CPU bound

• Well-written programs make your life easier

• You already made this step
Finding the Hot Spot: Profiling

- Use a profiler like `gprof` to find the hot spot
- Remember Amdahl's Law:

You need at least 99.9% of the program runtime!
Preparing your program

Compile & link all files of your program with profiling support:

```
gcc -O3 -g -pg prog.c -o prog
```

**WARNING:** gprof doesn't work correctly with multi-threaded programs. Details & Workaround:

http://sam.zoy.org/writings/programming/gprof.html

**WARNING 2:** Even then, improper parallelization can give you distorted timing results!
Using gprof

• Run your program with typical input:

  ./prog 40
  Result: N! = 18376134811363311616

• Run gprof on program & profile data file (gmon.out):

  gprof prog
  ...

Profiling Example: N!

```c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

/* Compute n! */
uint64_t factorial(uint64_t n) {
    if(n <= 1) return 1;
    return n * factorial(n - 1);
}

int main(int argc, char **argv) {
    int i, n;
    uint64_t fac;

    n = atoi(argv[1]);
    for(i=0; i<1000000; i++) fac = factorial(n);
    printf("Result: N! = %"PRIu64"
", fac);
    return 0;
}
```
Output of `gprof` (Excerpt)

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 5.20% of 0.19 seconds

<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>73.7</td>
<td>0.01</td>
<td>0.13</td>
<td></td>
<td>main [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.13</td>
<td>0.00</td>
<td>1000000/1000000</td>
<td>factorial [2]</td>
</tr>
<tr>
<td>[2]</td>
<td>68.4</td>
<td>0.13</td>
<td>0.00</td>
<td>1000000+38000000</td>
<td>factorial [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.13</td>
<td>0.00</td>
<td>380000000</td>
<td>factorial [2]</td>
</tr>
<tr>
<td>[3]</td>
<td>26.3</td>
<td>0.05</td>
<td>0.00</td>
<td></td>
<td>frame_dummy [3]</td>
</tr>
</tbody>
</table>
Limitations of gprof

- Profiling multi-threaded programs can give you misleading results
- Results depend on chosen input
- Need to compile all code with profiling support to get accurate results – What about shared libraries?
- Profiling information limited by sampling granularity
Parallelization
Why is Parallelization hard?

• Magnitude: Timing-related issues in addition to sequential logic errors
• Determinism: Parallel programs are non-deterministic
• Expression: Data synchronization separate from data
• Biology: Humans can't think concurrently
What to do about it?

- You **must** work systematically and with methodology
- You **must** get it right the first time (or at least as much as possible)
- Hacking won't work
- Search for previous work on parallelization of your algorithms
Synchronization

Accesses to shared data which is updated during parallel phase must be synchronized.

• Best approach is to eliminate need for synchronization!

• Synchronization is expensive: Try to defer and aggregate updates
Shared Data

- Possible locations of shared data:
  - Global variables
  - Static variables in functions
  - Heap allocated data (shared pointers)

- Use good engineering to add locks:

```c
struct {
    int count;
    void *list;
    pthread_mutex_t list_mutex;
} shared_list;
```
Pthreads - You already know...

• Mutexes
• Condition variables
• MESA-style monitors:

```c
pthread_mutex_lock(&mutex);
while (!cond) {
    pthread_cond_wait(&condvar, &mutex);
}
do_work();
pthread_cond_signal(&other_cond);
pthread_mutex_unlock(&mutex);
```
Pthreads – But there's more!

- Barriers: Wait for specified number of threads
- Rwlocks: Concurrent reads & sequential writes
- Spinlocks: Don't block, spin (for short waits)
- Trylocks: Don't block, return result of lock operation immediately
- Timed locks: Try to acquire lock, but only wait up to specified amount of time
Deadlocks

Four necessary conditions for deadlocks:

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait
Deadlock Avoidance

• You need a locking protocol

• Define a partial order on locks:

\[ \text{lock}_1 <_p \text{lock}_2 <_p \text{lock}_3 <_p \ldots <_p \text{lock}_N \]

• Acquire locks only in this order (no circular wait)

• Deadlocks are a symptom of poorly designed software
Race Conditions

• You forgot to synchronize accesses to shared data

• Non-deterministic, can be very hard to find

• Tool for automatic detection: `helgrind` (part of Valgrind tool suite, see [http://www.valgrind.org/](http://www.valgrind.org/))
helgrind Overview

- Uses Eraser algorithm: Stefan Savage et al. 
  “Eraser: A Dynamic Data Race Detector for Multithreaded Programs”

- Usage:

  ```
  valgrind --tool=helgrind ./race
  ```

- Unavailable in Valgrind release 2.4 and later, use an older version
Data Race Example

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

int counter = 0;

void *threadx(void *arg) {
    int i;
    for(i=0; i<10; i++)  {counter++; printf("x"); sleep(1); }
}

void *thready(void *arg) {
    int i;
    for(i=0; i<10; i++)  {counter++; printf("o"); sleep(1); }
}

void main() {
    pthread_t tx, ty;
    pthread_create(&tx, NULL, &threadx, NULL);
    pthread_create(&ty, NULL, &thready, NULL);
    pthread_join(tx, NULL);
    pthread_join(ty, NULL);
    printf("Counter: %i
", counter);
}
```
Output of helgrind (Excerpt)

==25878== Helgrind, a data race detector for x86-linux.
==25878== Copyright (C) 2002-2004, and GNU GPL'd, by Nicholas Nethercote et al.
==25878== Using valgrind-2.2.0, a program supervision framework for x86-linux.
==25878== Copyright (C) 2000-2004, and GNU GPL'd, by Julian Seward et al.
==25878== For more details, rerun with: -v
==25878==
==25878== Thread 3:
==25878== Possible data race writing variable at 0x80497C8
==25878== at 0x8048500: thready (race.c:14)
==25878== by 0x1D4AFCDA: thread_wrapper (vg_libpthread.c:867)
==25878== by 0xB000F714: do__quit (vg_scheduler.c:1872)
==25878== Address 0x80497C8 is in BSS section of
/n/fs/grad/cbienia/course/race/race
==25878== Previous state: shared RO, no locks
...
xoxoxoxoxoxoxoxoxoxoxo
Counter: 20
...
==25878== ERROR SUMMARY: 11 errors from 11 contexts (suppressed: 5 from 2)
==25878== 16 possible data races found; 0 lock order problems
Limitations of helgrind

- False negatives (not all data races will be detected)
- False positives (lots of output)
- Only supports x86 processors
Refinement
Refining Synchronization: pthreadw

• pthreadw is a thread library wrapper
• Collects synchronization statistics during runtime
• No recompilation required, but recommended
• Usage:

  pthreadw ./prog

• Author of pthreadw is talking to you right now
Hot Lock Example

![Graph showing speedup vs number of CPUs]

- **Linear**
- **Program**
**Output of pthreadw (Excerpt)**

[threadw] Mutex functions:
[threadw]   g_mutex_lock:
[threadw]     - number of calls 23317786
[threadw]     - blocking rate 15.54%
[threadw]     - total elapsed time 328957707436 ns
[threadw]     - share of total CPU time 66.35%
[threadw]     - time per call (mean) 141075.9 ns
[threadw]     - share of thread lifetime (mean) 66.36%
...

[threadw] Mutex variables:
[threadw]       Rank  Name / Addr          Time [ns]       Uses     Contention
[threadw]           1. 0x6000000000021350 3104207952264 (62.6%) 800764 29.3%
[threadw]            Uses: - 0.0% as N/A in im_init() (im_init.c:133)
[threadw]            - 0.0% as im->sslock in im__call_stop() (region.c:139)
[threadw]            - 0.0% as im->sslock in im__call_stop() (region.c:141)
[threadw]            - 20.0% as im->sslock in im_buffer_unref() (buffer.c:65)
[threadw]            - 20.0% as im->sslock in im_buffer_unref() (buffer.c:111)
...
[threadw]            - 20.0% as im->sslock in im_buffer_ref() (buffer.c:214)
[threadw]            - 20.0% as im->sslock in im_buffer_ref() (buffer.c:225)
[threadw]            - 0.0% as im->sslock in im__call_start() (region.c:112)
[threadw]            - 0.0% as im->sslock in im__call_start() (region.c:114)
[threadw]            - 9.9% as im->sslock in im_buffer_done() (buffer.c:122)
[threadw]            - 9.9% as im->sslock in im_buffer_done() (buffer.c:128)
Result of Hot Lock Elimination

![Graph showing the result of Hot Lock Elimination. The x-axis represents the number of CPUs, ranging from 0 to 64. The y-axis represents speedup, ranging from 0 to 65. Two lines are shown: one labeled "Linear" and another labeled "Program." The "Linear" line shows a linear relationship between the number of CPUs and speedup, while the "Program" line shows a slightly curved relationship. The graph indicates that speedup increases linearly with the number of CPUs.]
Limitations of `pthreadw`

- Only detects issues related to synchronization
- Slows down program
- Might affect thread schedule
- Does not detect non-standard forms of synchronization
Other Reasons for Bad Scalability

● Incomplete list, in no particular order:
  - Program becomes I/O bound
  - Program becomes memory bound
  - Thrashing
  - More spinning
  - Increasing number of cache misses
  - ...

Instrumentation

• Instrument your source code to find bottlenecks
• You need a timer with very high precision.
• Recommendation: `clock_gettime()`
• To use `clock_gettime()`, you have to `#include <time.h>` and link with librt (`-lrt`)
• Store counter values in `uint64_t` from `inttypes.h`
Thank you!