Issues in Parallelization

Matteo Frigo

Cilk Arts

June 9, 2009
**Goal:** highlight common performance problems in parallel programs.

- Wrong grain size in `cilk_for` loops.
- Too little parallelism.
- Memory bandwidth limitations.
- Too little burdened parallelism.

**How:**

- Run microbenchmarks.
- Explain why they do or do not work as intended.
Example 1: a small loop with grain size = 1

Code:

```c
const int N = 100 * 1000 * 1000;

void cilk_for_grainsize_1()
{
    #pragma cilk_grainsize = 1
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Expectations:

- Parallelism should be large, perhaps $\Theta(N)$ or $\Theta(N/\log N)$.
- We should see great speedup.
Speedup is indeed great...
but performance is lousy
How `cilk_for` is implemented

**Source:**

```c
void recur(int lo, int hi) {
    if ((hi - lo) > GRAINSIZE) {
        int mid = lo + (hi - lo) / 2;
        cilk_spawn recur(lo, mid);
        cilk_spawn recur(mid, hi);
    } else
        for (int i = lo; i < hi; ++i)
            BODY(i);
}

recur(A, B);
```

**Implementation:**

```c
...
Cilk++ chooses a grain size if you don’t specify one.

```c
void cilk_for_default_grainsize()
{
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Cilk++’s heuristic for the grain size:

\[
\text{grain size} = \min \left\{ \frac{N}{8P}, 512 \right\}.
\]

- Generates about \(8P\) parallel leaves.
- Works well if the loop iterations are not too unbalanced.
Speedup with default grain size

Number of Processors

Speedup versus serial code

Number of Processors
A large grain size should be even faster, right?

```c
void cilk_for_large_grainsize()
{
    #pragma cilk_grainsize = N
    cilk_for (int i = 0; i < N; ++i)
        fib(2);
}
```

Actually, no (except for noise):

<table>
<thead>
<tr>
<th>Grain size</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.55 s</td>
</tr>
<tr>
<td>default (= 512)</td>
<td>2.44 s</td>
</tr>
<tr>
<td>$N$ (= $10^8$)</td>
<td>2.42 s</td>
</tr>
</tbody>
</table>
Speedup with grain size $= N$
Tradeoff between grain size and parallelism

Use the PPA to understand the tradeoff:

<table>
<thead>
<tr>
<th>Grain size</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6,951,154</td>
</tr>
<tr>
<td>default (= 512)</td>
<td>248,784</td>
</tr>
<tr>
<td>$N$ (= $10^8$)</td>
<td>1</td>
</tr>
</tbody>
</table>

In the PPA, $P = 1$:

\[
\text{default grain size} = \min \left\{ \frac{N}{8P}, 512 \right\} = \min \left\{ \frac{N}{8}, 512 \right\}.
\]
Lessons learned

- Measure overhead before measuring speedup.
  - Compare 1-processor Cilk++ versus serial code.
- Small grain size $\Rightarrow$ higher work overhead.
- Large grain size $\Rightarrow$ less parallelism.
- The default grain size is designed for small loops that are reasonably balanced.
  - You may want to use a smaller grain size for unbalanced loops or loops with large bodies.
- Use the PPA to measure the parallelism of your program.
Example 2: A for loop that spawns

Code:

```c
const int N = 10 * 1000 * 1000;

/* empty test function */
void f() {
}

void for_spawn()
{
    for (int i = 0; i < N; ++i)
        cilk_spawn f();
}
```

Expectations:
- I am spawning $N$ parallel things.
- Parallelism should be $\Theta(N)$, right?
“Speedup” of `for_spawn()`
Insufficient parallelism

PPA analysis:
- PPA says that both work and span are $\Theta(N)$.
- Parallelism is $\approx 1.62$, independent of $N$.
- Too little parallelism: no speedup.

Why is the span $\Theta(N)$?

```
for (int i = 0; i < N; ++i)
    cilk_spawn f();
```
Alternative: a cilk_for loop.

Code:

```c
/* empty test function */
void f() { }

void test_cilk_for()
{
    cilk_for (int i = 0; i < N; ++i)
        f();
}
```

PPA analysis:
The parallelism is about 2000 (with default grain size).
- The parallelism is high.
- As we saw earlier, this kind of loop yields good performance and speedup.
Lessons learned

- `cilk_for()` is different from `for(...) cilk_spawn`.
- The span of `for(...) cilk_spawn` is $\Omega(N)$.
- For simple flat loops, `cilk_for()` is generally preferable because it has higher parallelism.
- (However, `for(...) cilk_spawn` might be better for recursively nested loops.)
- Use the PPA to measure the parallelism of your program.
Example 3: Vector addition

Code:

```c
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i)
        A[i] = B[i] + C[i];
}
```

Expectations:

- The PPA says that the parallelism is 68,377.
- This will work great!
Speedup of `vector_add()`
Bandwidth of the memory system

A typical machine: AMD Phenom 920 (Feb. 2009).

<table>
<thead>
<tr>
<th>Cache level</th>
<th>daxpy bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>19.6 GB/s per core</td>
</tr>
<tr>
<td>L2</td>
<td>18.3 GB/s per core</td>
</tr>
<tr>
<td>L3</td>
<td>13.8 GB/s shared</td>
</tr>
<tr>
<td>DRAM</td>
<td>7.1 GB/s shared</td>
</tr>
</tbody>
</table>

**daxpy:** \( x[i] = a \times x[i] + y[i] \), double precision.

The memory bottleneck:

- A single core can generally saturate most of the memory hierarchy.
- Multiple cores that access memory will conflict and slow each other down.
How do you determine if memory is a bottleneck?

**Hard problem:**
- No general solution.
- Requires guesswork.

**Two useful techniques:**
- Use a profiler such as the Intel VTune.
  - Interpreting the output is nontrivial.
  - No sensitivity analysis.
- Perturb the environment to understand the effect of the CPU and memory speeds upon the program speed.
How to perturb the environment

- Overclock/underclock the processor, e.g. using the power controls.
  - If the program runs at the same speed on a slower processor, then the memory is (probably) a bottleneck.

- Overclock/underclock the DRAM from the BIOS.
  - If the program runs at the same speed on a slower DRAM, then the memory is not a bottleneck.

- Add spurious work to your program while keeping the memory accesses constant.

- Run $P$ independent copies of the serial program concurrently.
  - If they slow each other down then memory is probably a bottleneck.
const int N = 50 * 1000 * 1000;

double A[N], B[N], C[N];

void vector_add()
{
    cilk_for (int i = 0; i < N; ++i) {
        A[i] = B[i] + C[i];
        fib(5); // waste time
    }
}
Speedup of perturbed `vector_add()`

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

Graph showing speedup for original, `fib(5)`, and `fib(10)` functions as the number of processors increases. The graph is linear, with the speedup increasing proportionally with the number of processors.
Interpreting the perturbed results

The memory is a bottleneck:
- A little extra work \((\text{fib}(5))\) keeps 8 cores busy. A little more extra work \((\text{fib}(10))\) keeps 16 cores busy.
- Thus, we have enough parallelism.
- The memory is *probably* a bottleneck. (If the machine had a shared FPU, the FPU could also be a bottleneck.)

OK, but how do you fix it?
- \texttt{vector_add} cannot be fixed in isolation.
- You must generally restructure your program to increase the reuse of cached data. Compare the iterative and recursive matrix multiplication from yesterday.
- (Or you can buy a newer CPU and faster memory.)
Lessons learned

- Memory is a common bottleneck.
- One way to diagnose bottlenecks is to perturb the program or the environment.
- Fixing memory bottlenecks usually requires algorithmic changes.
Example 4: Nested loops

Code:

```c
const int N = 1000 * 1000;

void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

Expectations:

- The inner loop does 4 things in parallel. The parallelism should be about 4.
- The PPA says that the parallelism is 3.6.
- We should see some speedup.
"Speedup" of `inner_parallel()`

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>
Interchanging loops

**Code:**

```c
const int N = 1000 * 1000;

void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
```

**Expectations:**

- The outer loop does 4 things in parallel. The parallelism should be about 4.
- The PPA says that the parallelism is 4.
- Same as the previous program, which didn’t work.
Speedup of `outer_parallel()`

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>

- **PPA estimate**
- **Test run**
Parallelism vs. burdened parallelism

**Parallelism:**
The best speedup you can hope for.

**Burdened parallelism:**
Parallelism after accounting for the unavoidable migration overheads.

Depends upon:
- How well we implement the Cilk++ scheduler.
- How you express the parallelism in your program.

**The PPA prints the burdened parallelism:**
- 0.29 for `inner_parallel()`, 4.0 for `outer_parallel()`.
- In a good program, parallelism and burdened parallelism are about equal.
What is the burdened parallelism?

**Code:**

```
A();
cilk_spawn B();
C();
D();
cilk_sync;
E();
```

**Burdened critical path:**

The **burden** is $\Theta(10000)$ cycles (locks, malloc, cache warmup, reducers, etc.)
The burden in our examples

\( \Theta(N) \) spawns/syncs on the critical path (large burden):

```c
void inner_parallel()
{
    for (int i = 0; i < N; ++i)
        cilk_for (int j = 0; j < 4; ++j)
            fib(10); /* do some work */
}
```

\( \Theta(1) \) spawns/syncs on the critical path (small burden):

```c
void outer_parallel()
{
    cilk_for (int j = 0; j < 4; ++j)
        for (int i = 0; i < N; ++i)
            fib(10); /* do some work */
}
```
Lessons learned

- Insufficient parallelism yields \textit{no speedup}; high burden yields \textit{slowdown}.
- Many spawns but small parallelism: suspect large burden.
- The PPA helps by printing the burdened span and parallelism.
- The burden can be interpreted as the number of spawns/syncs on the critical path.
- If the burdened parallelism and the parallelism are approximately equal, your program is ok.
We have learned to identify and address these problems:

- High overhead due to small grain size in \texttt{cilk_for} loops.
- Insufficient parallelism.
- Insufficient memory bandwidth.
- Insufficient burdened parallelism.