CSE 160
Lecture 15

Cilk parallel programming language
Announcements

• No lab session today
Today’s Lecture

- A tip on A2 – autotuning
- Cilk parallel programming language
Implementation

• 3 parameters affect performance
  ‣ Number of threads nt
  ‣ Mesh size x
  ‣ Chunk size c (dynamic case)

• A full search of the parameter space is unnecessary

• For the static case, optimize –x on 1 thread, then use this value on multiple threads; you may tweak x

Jim Demmel, U. C. Berkeley
Optimization parameter space

- Sparse matrix vector multiply example

---

<table>
<thead>
<tr>
<th>row block size (r)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1.75</td>
<td>1.52</td>
<td>0.99</td>
<td>1.33</td>
<td>1.51</td>
<td>1.64</td>
<td>1.79</td>
<td>1.83</td>
<td>1.89</td>
<td>1.75</td>
<td>1.85</td>
<td>1.72</td>
</tr>
<tr>
<td>11</td>
<td>1.72</td>
<td>1.64</td>
<td>1.12</td>
<td>1.23</td>
<td>1.45</td>
<td>1.60</td>
<td>1.71</td>
<td>1.80</td>
<td>1.88</td>
<td>1.91</td>
<td>1.88</td>
<td>1.97</td>
</tr>
<tr>
<td>10</td>
<td>1.73</td>
<td>1.47</td>
<td>1.14</td>
<td>1.23</td>
<td>1.38</td>
<td>1.54</td>
<td>1.69</td>
<td>1.67</td>
<td>1.86</td>
<td>1.89</td>
<td>1.98</td>
<td>1.93</td>
</tr>
<tr>
<td>9</td>
<td>1.54</td>
<td>1.74</td>
<td>1.24</td>
<td>1.00</td>
<td>1.27</td>
<td>1.42</td>
<td>1.55</td>
<td>1.61</td>
<td>1.71</td>
<td>1.73</td>
<td>1.75</td>
<td>1.90</td>
</tr>
<tr>
<td>8</td>
<td>3.39</td>
<td>2.40</td>
<td>1.44</td>
<td>1.16</td>
<td>1.16</td>
<td>1.32</td>
<td>1.44</td>
<td>1.47</td>
<td>1.68</td>
<td>1.75</td>
<td>1.77</td>
<td>1.84</td>
</tr>
<tr>
<td>7</td>
<td>3.38</td>
<td>2.04</td>
<td>1.65</td>
<td>1.22</td>
<td>1.04</td>
<td>1.20</td>
<td>1.30</td>
<td>1.44</td>
<td>1.52</td>
<td>1.63</td>
<td>1.65</td>
<td>1.74</td>
</tr>
<tr>
<td>6</td>
<td>3.79</td>
<td>1.77</td>
<td>1.72</td>
<td>1.44</td>
<td>1.13</td>
<td>1.14</td>
<td>1.23</td>
<td>1.31</td>
<td>1.41</td>
<td>1.52</td>
<td>1.58</td>
<td>1.65</td>
</tr>
<tr>
<td>5</td>
<td>3.20</td>
<td>1.74</td>
<td>1.99</td>
<td>1.52</td>
<td>1.34</td>
<td>1.19</td>
<td>0.97</td>
<td>1.17</td>
<td>1.27</td>
<td>1.36</td>
<td>1.42</td>
<td>1.50</td>
</tr>
<tr>
<td>4</td>
<td>3.32</td>
<td>4.07</td>
<td>1.74</td>
<td>2.37</td>
<td>1.52</td>
<td>1.38</td>
<td>1.18</td>
<td>1.14</td>
<td>1.32</td>
<td>1.19</td>
<td>1.22</td>
<td>1.29</td>
</tr>
<tr>
<td>3</td>
<td>2.55</td>
<td>3.35</td>
<td>3.11</td>
<td>1.74</td>
<td>1.97</td>
<td>1.71</td>
<td>1.52</td>
<td>1.34</td>
<td>1.19</td>
<td>1.06</td>
<td>1.03</td>
<td>0.60</td>
</tr>
<tr>
<td>2</td>
<td>1.93</td>
<td>2.54</td>
<td>2.76</td>
<td>2.73</td>
<td>1.62</td>
<td>1.70</td>
<td>1.85</td>
<td>2.40</td>
<td>1.70</td>
<td>1.54</td>
<td>1.27</td>
<td>1.17</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.35</td>
<td>1.39</td>
<td>1.44</td>
<td>1.43</td>
<td>1.47</td>
<td>1.48</td>
<td>1.43</td>
<td>1.34</td>
<td>1.42</td>
<td>1.41</td>
<td>1.43</td>
</tr>
</tbody>
</table>

SpMV BCSR Profile [ref -294.5 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]

Jim Demmel, U. C. Berkeley

©2013 Scott B. Baden / CSE 160 / Winter 2013
Today’s lecture

• Parallel Programming Languages
  ‣ Cilk
Dynamic parallelism

- How to support dynamic creation of parallelism, while hiding the details?
- Dynamic parallelism is much harder to manage than static parallelism
  - How to keep the processors equally busy?
  - How to avoid excessive overhead costs?
Managing application complexity

- Threads communicate anonymously
  ‣ Correctness and synchronization
  ‣ Workload distribution
- Scalability
- Task granularity
An alternative

- Let’s think of a computation in terms of a graph, more precisely, a DAG
- Nodes denote computation, edges data dependence
Cilk

- Cilk is a programming language that supports a constrained model of thread-based parallelism with *performance guarantees*
- Useful in implementing divide and conquer algorithms
- See [http://supertech.lcs.mit.edu/cilk](http://supertech.lcs.mit.edu/cilk)
- Cilk Plus: an extension to C and C++
  - Supported by Intel compilers and GCC 4.7
A first CILK program

- `fib()` is called from a dynamically spawned thread
- Non-blocking call
- Spawned calls to `fib()` execute concurrently
- Parent continues until it reaches a `sync` barrier, and waits for children to return

```cilk
int fib (int n)
{
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
```
cilk int fib (int n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
Performance

- **Work** is the time to execute the entire computation on one processor ($T_1$)
- **Critical path length**: $T_\infty$; longest time to compute along any dependence path
- Assume $P$ processors
- $T_P =$ time on $P$ processors

Charles Leiserson
Performance bounds

• \( T_P \geq T_1 / P \)
    ‣ In one step, \( P \) processors can do at most \( P \) units of work

• \( T_P \geq T_\infty \)
  ‣ In one step, \( P \) processors can do no more work than an infinite number of processors can

• The maximum possible speedup is defined as the parallelism: \( T_1 / T_\infty \)

• Average amount of work per step along the span
Dot product

cilk void dotprod(
    double *s, double* x, double *y, int n){
    if (n<=MIN_N) {
        for (int i=0, double t=0; i<n; i++)
            t += x[i] * y[i];
        *s = t;
    } else {
        double t0, t1;
        spawn dotprod(&t0, x, y, n/2);
        spawn dotprod(&t1,x+n/2,y+n/2,n-n/2);
        sync;
        *s = t0 + t1;
    }
}

Scheduling

• Spawn expresses potential parallelism
• The scheduler assigns threads to cores dynamically
• When a processor runs out of work it steals work from another processor
  ‣ Picks a processor at random
  ‣ Removes a thread from the tail of the list of the shallowest nonempty level of the ready queue
• Why the shallowest level?
  ‣ Ensures progress along the critical path
  ‣ Granularity considerations
Performance

- \( T_P \approx \frac{T_1}{P} + c_\infty T_\infty \)
- \( c_\infty \approx 1.5 \)
- The critical path is a stronger lower bound on \( T_P \) if it exceeds the average parallelism \( \frac{T_1}{T_\infty} \)
- Otherwise, \( \frac{T_1}{P} \) is the stronger bound
- Depends on the ability to have good scheduler
Matrix multiply

\[
\begin{pmatrix}
  c_{11} & c_{12} & \cdots & c_{1n} \\
  c_{21} & c_{22} & \cdots & c_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{n1} & c_{n2} & \cdots & c_{nn}
\end{pmatrix}
= \begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\times
\begin{pmatrix}
  b_{11} & b_{12} & \cdots & b_{1n} \\
  b_{21} & b_{22} & \cdots & b_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{n1} & b_{n2} & \cdots & b_{nn}
\end{pmatrix}
\]

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}
\]

Charles Leiserson
Divide and Conquer algorithm

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= 
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\times 
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

\[
= 
\begin{pmatrix}
A_{11}B_{11} & A_{11}B_{12} \\
A_{21}B_{11} & A_{21}B_{12}
\end{pmatrix}
+ 
\begin{pmatrix}
A_{12}B_{21} & A_{12}B_{22} \\
A_{22}B_{21} & A_{22}B_{22}
\end{pmatrix}
\]
Matrix multiply in Cilk

    Allocate temporary n x n matrix
    base case & partition matrices
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    spawn Mult(C22,A21,B12,n/2);
    spawn Mult(C21,A21,B11,n/2);
    spawn Mult(T11,A12,B21,n/2);
    spawn Mult(T12,A12,B22,n/2);
    spawn Mult(T22,A22,B22,n/2);
    spawn Mult(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}

cilk void MtxAdd(*C, *T, n) {
    spawn MtxAdd(C11,T11,n/2);
    spawn MtxAdd(C12,T12,n/2);
    spawn MtxAdd(C21,T21,n/2);
    spawn MtxAdd(C22,T22,n/2);
    sync;
    return;
}
Performance

• Work = \Theta(n^3)
  ‣ Work for matrix add: A(n) = 4A(n/2) + \Theta(1) = \Theta(n^2)
    [case 1 of the master theorem]
  ‣ Work for Matrix mpy:
    M(n) = 8M(n/2) + A(n) + \Theta(1) = \Theta(n^3)  \ [case 1]

• Critical path (span) = \Theta(lg^2 n)
  ‣ Critical path for matrix add:
    A(n) = A(n/2) + \Theta(1) = \Theta(lg n)  \ [case 2]
  ‣ Critical path of Matrix mpy
    M(n) = M(n/2) + A(n) + \Theta(1) = \Theta(lg^2 n)  \ [case 2]

• Parallelism = \Theta(n^3/lg^2 n)

• We can modify the algorithm to avoid temporaries
  ‣ Span increases to \Theta(n), but faster in practice
Parallel Merge sort in Cilk

- Work = $\Theta(n \lg n)$
- Span = $\Theta(\lg^3 n)$
- Parallelism= $\Theta(n/\lg^2 n)$

```cilk
cilk void P_MergeSort(int *b, int *a, int n) {
    if (n==1) {
        b[0] = a[0];
    } else {
        int *c;
        c = allocate n element buffer;
        spawn P_MergeSort(c, a, n/2);
        spawn P_MergeSort(c+n/2, a+n/2, n-n/2);
        sync;
        spawn P_Merge(b, c, c+n/2, n/2, n-n/2);
    }
}
```
Compilation

- The translate produces two clones of a Cilk function
  - Fast clone: traditional serial code
  - Slow clone: multithreaded version
- Ever spawn invokes a fast clone
- When a thread is stolen, it becomes a slow clone