Threading: Correctness & Performance

Week 4, Lecture 2
So Far …

• Parallelized “Game of Life” via Pthreads on Multicore
  – Showed how to move from sequential to parallel version
    • Addressed issues that arose with respect to correctness.
  – More on POSIX Threads:
    https://computing.llnl.gov/tutorials/pthreads/

• Today
  – Performance
    • What is the penalty of barriers for enforcing correctness?
    • How do we measure performance?
  – New Algorithm
    • Implement as part of a multicore programming assignment.
  – Transition to GPGPU Portion of the Course
Performance: Parallelizing “Game of Life”

• How well did we do? Execution time.
Another Perspective: Speedup

![Graph showing speedup with different scenarios: Ideal, No barriers, Barriers. The x-axis represents the number of cores, and the y-axis represents the speedup.](image)
Performance Analysis

• Expectation
  – Achieve speedup proportional to the core count.

• Reality
  – Linear scaling as the number of cores increases to 6 or 7 cores. Then, performance drops with 8 cores!

• What Happened?
Answer: The Operating System (OS)

- My machine (and most others) run an OS that is composed of a suite of kernel-level threads and user-space processes in the form of daemons.
  - Modern OS are pretty good at scheduling processes to cores such that they stay out of each others way.

- Game of Life? CPU intensive
  - OS tries to allocate exclusive cores to each thread.
  - OS pushes other tasks, e.g., daemons, to other cores.
  - Unfortunately, when we hit 8 cores, the daemons no longer have a home where they stay out of the way of our “Game of Life” processes.
My Compute Node

- How much ELSE runs next to your program?
- Here is the `ps` output from my compute node.

<table>
<thead>
<tr>
<th>USER</th>
<th>PID</th>
<th>%CPU</th>
<th>%MEM</th>
<th>VSZ</th>
<th>RSS</th>
<th>TT</th>
<th>STAT</th>
<th>STARTED</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>78580</td>
<td>536</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:53.28</td>
<td>/sbin/launchd</td>
</tr>
<tr>
<td>root</td>
<td>10</td>
<td>0.0</td>
<td>0.1</td>
<td>80400</td>
<td>5944</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>1:23.74</td>
<td>/usr/libexec/kextd</td>
</tr>
<tr>
<td>root</td>
<td>11</td>
<td>0.0</td>
<td>0.0</td>
<td>75948</td>
<td>432</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>3:34.67</td>
<td>/usr/sbin/notifyd</td>
</tr>
<tr>
<td>root</td>
<td>14</td>
<td>0.0</td>
<td>0.0</td>
<td>75332</td>
<td>192</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>87:10.72</td>
<td>/usr/sbin/update</td>
</tr>
<tr>
<td>root</td>
<td>15</td>
<td>0.0</td>
<td>0.0</td>
<td>75372</td>
<td>328</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:00.01</td>
<td>/usr/launchd/SystemStarter</td>
</tr>
<tr>
<td>root</td>
<td>17</td>
<td>0.0</td>
<td>0.0</td>
<td>77044</td>
<td>636</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>32:00.30</td>
<td>/usr/sbin/syslogd</td>
</tr>
<tr>
<td>root</td>
<td>19</td>
<td>0.0</td>
<td>0.0</td>
<td>77368</td>
<td>1948</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:18.33</td>
<td>/usr/sbin/securityd -i</td>
</tr>
<tr>
<td></td>
<td>_mdnsresponder</td>
<td>22</td>
<td>0.0</td>
<td>0.0</td>
<td>77680</td>
<td>1428</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>2:24.11</td>
</tr>
<tr>
<td>feng</td>
<td>23</td>
<td>0.0</td>
<td>0.2</td>
<td>480992</td>
<td>6896</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>52:50.60</td>
<td>/System/Library/CoreServices/loginwindow.app/Contents/MacOS/loginwindow console</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>root</td>
<td>24</td>
<td>0.0</td>
<td>0.0</td>
<td>75448</td>
<td>336</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:00.19</td>
<td>/usr/sbin/KernelEventAgent</td>
</tr>
<tr>
<td>root</td>
<td>26</td>
<td>0.0</td>
<td>0.0</td>
<td>77056</td>
<td>824</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>135:33.90</td>
<td>/usr/libexec/hidd</td>
</tr>
<tr>
<td>root</td>
<td>29</td>
<td>0.0</td>
<td>0.0</td>
<td>75376</td>
<td>424</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:20.41</td>
<td>/bin/dynamic_pager -F /private/var/vm/swapfile</td>
</tr>
<tr>
<td>root</td>
<td>32</td>
<td>0.0</td>
<td>0.0</td>
<td>75460</td>
<td>1052</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:28.53</td>
<td>/usr/sbin/diskarbitration</td>
</tr>
<tr>
<td>root</td>
<td>33</td>
<td>0.0</td>
<td>0.1</td>
<td>78768</td>
<td>3312</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>6:32.83</td>
<td>/usr/sbin/DirectoryService</td>
</tr>
<tr>
<td>root</td>
<td>35</td>
<td>0.0</td>
<td>0.0</td>
<td>77076</td>
<td>1780</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>12:57.96</td>
<td>/usr/sbin/configd</td>
</tr>
<tr>
<td>root</td>
<td>38</td>
<td>0.0</td>
<td>0.0</td>
<td>75388</td>
<td>432</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:00.91</td>
<td>autofsd</td>
</tr>
<tr>
<td>root</td>
<td>39</td>
<td>0.0</td>
<td>0.0</td>
<td>75704</td>
<td>740</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:53.44</td>
<td>/usr/libexec/ApplicationFirewall/socketfilterfw</td>
</tr>
<tr>
<td>daemon</td>
<td>41</td>
<td>0.0</td>
<td>0.0</td>
<td>75356</td>
<td>928</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>1:57.50</td>
<td>/usr/sbin/distnoted</td>
</tr>
<tr>
<td>root</td>
<td>44</td>
<td>0.0</td>
<td>0.1</td>
<td>159404</td>
<td>6224</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>5:38.76</td>
<td>/System/Library/CoreServices/coreservicesd</td>
</tr>
<tr>
<td>root</td>
<td>75</td>
<td>0.0</td>
<td>0.0</td>
<td>86844</td>
<td>1388</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>0:09.38</td>
<td>/usr/sbin/blued</td>
</tr>
<tr>
<td></td>
<td>_windowserver</td>
<td>88</td>
<td>0.7</td>
<td>10.1</td>
<td>1475892</td>
<td>425456</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>1839:57.41</td>
</tr>
<tr>
<td>feng</td>
<td>223</td>
<td>0.0</td>
<td>0.0</td>
<td>75508</td>
<td>556</td>
<td>??</td>
<td>Ss</td>
<td>21Dec08</td>
<td>1:30.06</td>
<td>/sbin/launchd</td>
</tr>
</tbody>
</table>

Copyright © 2009 by W. Feng. Based on material from M. Sottile.
Measure “Wait-at-Barrier” Times
Source of Performance Problems?

- Load Imbalance
  - Recall the barrier problem. Skew in the arrival time at the barrier. What could cause this skew?
- Internal Load Imbalance
  - Different processors are doing different amounts of work.
- External Load Imbalance
  - Processes on different processors get perturbed by external forces leading to slower execution of a fixed workload.
  - A perfectly internally balanced application could see skew in per-process performance from this source.

What kind of load imbalance in our “Game of Life”? 
Effect on Synchronization

- Why did the barrier version perform worse?
  - Non-barrier version (w/ incorrect results) was indifferent to skew in the arrival time at the point where the barrier should occur.
  - External effects increased the variation in the skew observed in when each processor arrived at the barrier point.
    - Threads that were on cores shared with OS tasks took longer to complete their work, skewing their arrival time to the barrier point.
Correctness Issues: Synchronization

• Recall: \textit{Enforcing globally consistent state or invariants often requires cross-thread synchronization.}
  – The \textbf{barrier}, the simplest (and most disruptive) form of synchronization. Why?
  – Every thread must stop and wait for every other thread to catch-up.

• When is the above “OK”?
  – Have a balanced workload, or
  – Require a global control-flow state to be reached before making state changes that impact the global algorithm.

The “Game of Life” has both of these.
Why Synchronization?

• Need to restrict the advancing of one or more program counters to avoid getting into situations where nondeterminism or erroneous computations can occur.
  – Require synchronization to protect shared data that requires atomicity in accesses (most often writes).

• How do we implement a barrier?
Anatomy of a Barrier

- A shared counter with simple operations defined on it.
  - A thread arrives at the barrier and atomically increments it.
  - If the value after the increment is less than the barrier threshold, the thread blocks.
  - Else it sets the value of the barrier back to zero and signals the blocked processes to unblock and exit the barrier.

- What assumptions are made?
  - Atomic update of barrier counter.
  - Ability to block and wait for a signal.
Atomicity

• A complex operation is *atomic* if, from the outside, the result of intermediate operations during the execution of the operation are not visible.
  – An atomic operation has exclusive control over its state.
  – To outside operations, the state modified by or used by the operation only takes on the pre-atomic and post-atomic values. Outsiders can *never* observe the intermediate values.
  – All this becomes important when you have multiple streams of instructions executing in parallel.
Reflection: Concurrency vs. Parallelism

- What is the difference?
- They are sometimes used interchangeably.
- Is this OK to do?
Concurrency

• A sequence of instructions executes **concurrently** if they execute independent of each other as if they were executed at the same time.
  
  – They do *not* need to be executed truly at the same time though.
  
  – On a single processor computer, multi-tasking systems execute programs concurrently by interleaving their operations such that they appear to execute at the same time.
Parallelism

- Instruction streams that execute in parallel actually execute at the same time.
  - Parallelism allows multiple instructions to be executed at the exact same time.
  - Parallelism requires multiple processing units, ranging from small pipeline stages up through multithreaded architectures and multicore and multiprocessing systems.
What’s the Difference?

• Time.

• In concurrency, at any given time, a single operation is occurring.
  – High clock rates and clever interleaving can give the illusion of parallelism.
  – All modern desktop/server OS give you this. Embedded, maybe not.

• In parallelism, at a given point in time, multiple operations are occurring.
  – This is important to distinguish. Parallelism means it is extremely difficult (often impossible) to predict the interleaving of instructions.
Pragmatic Parallelism

• Presence of parallelism makes it hard to enforce atomicity.
• In concurrent systems that interleave, one can inform the interleaving algorithm of atomic operations to restrict how operations are interleaved.
  – Example: Cooperative multithreading can just not ever give up control during an atomic operation.
• In a parallel system, this is not true. Atomicity must be enforced without any control over the interleaving of operations.
  – There is no way to cheat in the parallel world to make atomicity work.
• So … how to make atomicity work?
Critical Regions and Mutual Exclusion

- Critical Regions (or Critical Sections)
  - Blocks of code that implement atomic operations.
  - Atomicity is realized by ensuring **mutual exclusion** in critical regions. What does this mean?
    - One and only one thread can be executing the region at any given time.
Classic Example: Bank Accounts

- A class called `Account` with methods `withdraw()` and `deposit()`. What happens if they are *not* atomic in a multithreaded environment?

```java
void withdraw(Account a, int i) {
    int balance = a.getBalance();
    if (balance < i) {
        error("Not enough money.");
        return;
    }
    balance = balance - i;
    a.setBalance(balance);
}

void deposit(Account a, int i) {
    int balance = a.getBalance();
    balance = balance + i;
    a.setBalance(balance);
}
```
Problem Due to Non-Atomic Operations

```
withdraw(200)
balance=a.getBalance();  // a.balance == 1000
if (balance > i)
    balance=balance-200;
    a.setBalance(balance);  // a.balance == 800
```

```
deposit(2000)
balance=a.getBalance();  // a.balance == 1000
balance=balance+2000;
a.setBalance(balance);  // a.balance == 3000
```

```
withdraw(200)
balance=a.getBalance();  // a.balance == 1000
if (balance > i)
    balance=balance-200;
    a.setBalance(balance);  // a.balance == 800
```

```
deposit(2000)
balance=a.getBalance();  // a.balance == 1000
balance=balance+2000;
a.setBalance(balance);  // a.balance == 3000
```

Neither yields the correct balance of 2800!
Preventing Atomicity Bugs

• Different interleavings of operations can cause differences in the results.
  – When some of these interleavings can result in incorrect output, we need to prevent them from occurring. How?

• Put “guards” around the critical section that restricts execution of the critical section to a single processor at a time.
  – Lock must be acquired to enter the critical section.
  – Lock is released when the critical section is complete.
  – Only one thread may hold the lock at any given time.
“Mutexes”

- **pthread_mutex** provides this locking capability with four functions of interest:
  - **pthread_mutex_init()**
    - Initialize a mutex.
  - **pthread_mutex_destroy()**
    - Free the resources associated with a mutex.
  - **pthread_mutex_lock()**
    - Acquire the mutex lock, block until it becomes available.
  - **pthread_mutex_unlock()**
    - Release the mutex lock.
Fixing the Bank Account …

- How do we fix this mess?
- Associate a mutex with an account so we lock the account when inside a critical region.
  - Assume the mutex is initialized in the account constructor.

```c
void withdraw(Account a, int i) {
    int balance = a.getBalance();
    if (balance < i) {
        error("Not enough money.");
        return;
    }
    balance = balance - i;
    a.setBalance(balance);
}

void deposit(Account a, int i) {
    int balance = a.getBalance();
    balance = balance + i;
    a.setBalance(balance);
}
```
Programmer Beware!

• Management of locks must be done with care.
  – What if I had forgotten to unlock on the insufficient funds condition?
    • For complex code, this can actually be very difficult to manage to ensure that locks are acquired and released at the right points without missing any.
    • Locks only prevent errors, *if properly used*. Due to their existence as library calls, the compiler has no idea that they are meaningful to the correctness of the code.
Semaphores: Origin and Specification

• Introduced by E.W. Dijkstra as a basic abstraction for enforcing mutual exclusion with simple semantics.
  – A semaphore is an integer.
  – It has simple operations:
    • Initialize \( s \) to 1, where \( s \) is the semaphore
    • \( P(s) \) : If \( s > 0 \), then decrement \( s \). Otherwise, block.
    • \( V(s) \) : If another thread is blocked on \( s \), unblock that thread. Otherwise increment \( s \).

**Interesting fact**: \( P() \) and \( V() \) are inherited from the original papers by Dijkstra in Dutch. \( P \) stands for *Prolaken*, meaning “to try and lower”, derived from *Probeer te verlagen*, meaning “to try and decrease”. \( V \) stands for *Verhogen*, meaning “to raise” or “to increase”.
Semaphores: Function

• Binary semaphore: Value is only ever zero or one.
• Given that a semaphore is an integer, other positive integers possible. Why?
  – Maybe a critical region is not protected for correctness but to control access to a limited resource.
  – Sometimes a limited resource can support a small number (>1) of threads using it. Semaphores can generalize to these situations.
• A detail that is often hidden from the programmer.
  – How to choose from a set of blocked threads to unblock?
  – FIFO? Prevents starvation and implements a crude level of fairness.
Monitors

• The concept of a monitor was inspired by two things:
  – Dijkstra’s semaphore concept
  – The notion of encapsulation in precursor languages to OO such as Simula.
• Monitors help prevent the case mentioned earlier where code circumvents locking to access data that must be protected in critical regions.
  – Data is encapsulated and hidden, and routines associated with the data enforce locking.
Yawn … Why Should I Care?

• These topics come up quite often in computer science. You may have seen them before (or are likely to see them again).

• **Databases**: Atomicity of sequences of operations in the form of transactions.

• **Operating systems**: OS brokers many resources for multiple processes. Many of these resources require a disciplined control of interactions.
Add “Transfer” Capability to Bank

- Add a function `transfer()` that withdraws from one account and deposits into the other.
- Design the algorithm such that the source of funds performs the transfer.

```c
void transfer(Account src, Account dest, int i) {
    pthread_mutex_lock(src.lock);
    pthread_mutex_lock(dest.lock);
    int balance = src.getBalance();

    if (balance < i) {
        error("Not enough money.");
        pthread_mutex_unlock(src.lock);
        pthread_mutex_unlock(dest.lock);
        return;
    }
    balance = balance - i;
    src.setBalance(balance);

    balance = dest.getBalance();
    balance = balance + i;
    dest.setBalance(balance);
    pthread_mutex_unlock(src.lock);
    pthread_mutex_unlock(dest.lock);
}
```
Correctness?

• What if two threads attempt to transfer between two accounts, one from A to B and the other from B to A?

• What could possibly go wrong?
  - \( t=1 \): A acquires a lock on itself.
  - \( t=2 \): B acquires a lock on itself.
  - \( t=3 \): A blocks because it cannot acquire a lock on B since B already locked itself.
  - \( t=4 \): B blocks for a similar reason.

• Result: Deadlock
Preventing Deadlock

• Lock Reordering
• “trylock” lock acquisition routines that attempt to acquire a lock, but if they fail immediately return instead of blocking. On failure, one can unlock potential sources of deadlock before retrying from the beginning.
• Thread analysis tools.
• Avoiding explicit locks.
And Now Back to Our Barrier …

• Function
  – Atomically increment a counter
    • Block if the counter is not at a critical value, or
    • Reset the counter and notify all blocked threads if it is at a critical value

• How?
  – Use pthreads mutex and condition variables.
Mutex

- **pthreads_mutex_lock(pthreads_mutex_t)**
  - Tries to acquire the lock, and blocks if the lock is already taken (waits until it becomes free).
  - Fairness details are not the responsibility of the programmer. The thread system handles that.

- **pthread_mutex_trylock()**
  - Nonblocking lock probe.

- **pthread_mutex_unlock(pthreads_mutex_t)**
  - Releases the lock held by a thread.
Condition Variables

- Mutexes protect critical regions ...
- Condition variables allow threads to block until other threads inform them that it is OK to proceed.
- Six relevant functions
  - `pthread_cond_init(pthread_cond_t *, pthread_condattr_t *)`
  - `pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *)`
  - `pthread_cond_timedwait(pthread_cond_t *, pthread_mutex_t *, const struct timespec *)`
  - `pthread_cond_signal(pthread_cond_t *)`
  - `pthread_cond_broadcast(pthread_cond_t *)`
  - `pthread_cond_destroy(pthread_cond_t *)`
Functions for Condition Variables

- **pthread_cond_init(pthread_cond_t *, pthread_condattr_t **)  
  - Initialization. **NULL** for attributes yields default behavior (most common).

- **pthread_cond_wait(pthread_cond_t *, pthread_mutex_t **)  
  - Equivalent to unlocking the mutex, blocking until signaled by another thread, and then reacquiring the lock.

    ```c
    pthread_unlock_mutex(mutex);
    block_until_signal();
    pthread_lock_mutex(mutex);
    ```

- **pthread_cond_timedwait(pthread_cond_t *, pthread_mutex_t *, const struct timespec **)  
  - Same as **pthread_cond_wait** except with a timeout to abort waiting. If the timeout occurs, the **pthread_cond_timedwait()** function returns **ETIMEDOUT** so the thread knows that the wait failed.
Functions for Condition Variables

- **pthread_cond_signal(pthread_cond_t *)**
  - At least one thread blocked on the condition variable shall be unblocked.
- **pthread_cond_broadcast(pthread_cond_t *)**
  - All threads blocked on the condition variable shall be unblocked.
- **pthread_cond_destroy(pthread_cond_t *)**
  - Destroy and cleanup.
barrier_init and barrier_destroy

```c
int barrier_init(barrier_t *barrier, int needed)
{
    barrier->needed = needed;
    barrier->called = 0;
    pthread_mutex_init(&barrier->mutex, NULL);
    pthread_cond_init(&barrier->cond, NULL);
    return 0;
}

int barrier_destroy(barrier_t *barrier)
{
    pthread_mutex_destroy(&barrier->mutex);
    pthread_cond_destroy(&barrier->cond);
    return 0;
}
```
```c
int barrier_wait(barrier_t *barrier) {
    pthread_mutex_lock(&barrier->mutex);
    barrier->called++;
    if (barrier->called == barrier->needed) {
        barrier->called = 0;
        pthread_cond_broadcast(&barrier->cond);
    } else {
        pthread_cond_wait(&barrier->cond, &barrier->mutex);
    }
    pthread_mutex_unlock(&barrier->mutex);
    return 0;
}
```
Other pthread Functions

- `pthread_self()` / `pthread_equal()`
  - Access to unique thread IDs
- `pthread_once()`
  - Initialization routines with enforced once-only calling.
- `pthread_yield()`
  - Force thread to yield processor to others and rejoin queue.
- Miscellaneous routines for canceling threads and attribute manipulation. Useful for handling the tedious little details of real programs, but not often critical for the overall code design. (See man pages for more information.)
Smith-Waterman Algorithm

- A dynamic programming algorithm that is guaranteed to find the optimal local alignment between a pair of sequences.
  - Compares sequence segments of all possible lengths and optimizes the similarity measure, i.e., alignment score.
  - Outputs the highest scoring local alignment.

- Two-stage algorithm
  - Matrix filling
    - Start with matrix elements initialized with 0s.
    - Suggest inserting a ‘0’-filled column and row on the left and at the top of the matrix.
  - Backtracing
Stage One: Matrix Filling

- Substitution matrix, $M$
  - Each entry $M(i,j)$ indicates the score of aligning the characters $i$ and $j$.
    
    **Reward:** If $M(i,j) > 0$, match. (Large positive # \(\rightarrow\) better match.)
    
    **Penalty:** If $M(i,j) < 0$, mismatch. (Large positive # \(\rightarrow\) better match.)

- Gap Scoring Scheme
  - Gaps (i.e., ‘-’) are introduced into sequences so that similar characters across sequences might better align and increase the alignment score, but are considered a mismatch that incurs some penalty. Two types:
    
    **Gap Open Penalty** ($o$): Penalty for starting (or opening) a gap in the alignment.
    
    **Gap Extension Penalty** ($e$): Penalty for extending a previously existing gap in the alignment by one unit.
  
  - If there are $k$ consecutive gaps in an alignment, then the total gap penalty incurred by that gap is $o + ke$. 

Stage One: Matrix Filling

• The recursive data dependence of the elements in the dynamic programming matrix can be explained by the following equations:

\[
DP_N[i, j] = e + \max \left\{ \begin{array}{l}
DP[i - 1, j] \\
DP_W[i - 1, j] + o \\
DP_{NW}[i - 1, j] + o
\end{array} \right.
\]

\[
DP_W[i, j] = e + \max \left\{ \begin{array}{l}
DP_N[i, j - 1] + o \\
DP_W[i, j - 1] \\
DP_{NW}[i, j - 1] + o
\end{array} \right.
\]

\[
DP_{NW}[i, j] = M(X_i, Y_j) + \max \left\{ \begin{array}{l}
DP_N[i - 1, j - 1] \\
DP_W[i - 1, j - 1] \\
DP_{NW}[i - 1, j - 1]
\end{array} \right.
\]

• Equations indicate the presence of three weighted matrices and imply a 3D dependency among the elements of the matrix.
Stage Two: Backtracing

- Yields the highest scoring local alignment.
- The backtrace begins at the matrix cell that holds the optimal alignment score (i.e., highest score) and proceeds in a direction opposite to that of the matrix filling until a cell with score zero is encountered.
- The path traced by this operation generates the alignment.
Example

Optimal Local Alignment

ATCAGAGTC
GTCAG--TCA

++++^+^+
A Single Sequence

Copyright © 2009 by W. Feng. Based on material from M. Sottile.
Programming Assignment #1
Due 2/26/09 in class. Bring your laptop (as a back-up).

1. Implement a sequential version of Smith-Waterman in C/C++.
2. Parallelize the sequential algorithm using pthreads on a single multicore machine of the rlogin.cs.vt.edu cluster.

Evaluation Criteria
- Documentation and Readability (20%)
- Correctness (40%)
- Performance (40%)
Languages, Tools, and Models

- You will use pthreads and multicore for your programming assignment, but in fact, there are MANY choices …
Languages

• The key features that make languages count here are the introduction of parallelism in the syntax.
• This is key, because it allows the compiler to help generate correct and efficient parallel code.
• The fact that traditional thread libraries are library-based has always limited this (unless you have a whole-program analyzing compiler, which is rare, and even then it has no idea that pthreads is any different than any other library).
Tools and Models

• Tools and models are add-ons to existing languages.
• Example
  – Tuple spaces and Hadoop are higher-level concepts implemented on top of an existing language.
  – Linda and Hadoop are tools that implement a model.
  – If you were bored and had a lot of time on your hands …
    • You might choose to implement a model yourself instead of using an existing tool.
Types of Parallel Programs

- A coarse classification of parallel programs is to break them down by how many different instruction streams run in parallel (different=actually different code, not just different PCs in the same code) and how many different owners exist for data.
- **SPSD**: Single program, single data = sequential programming
- **SPMD**: Single program, multiple data = one thread per core, local data for each. Parallelism is explicit.
- **MPSD**: Multiple program, single data
- **MPMD**: Multiple program, multiple data
Flynn’s Taxonomy

• Flynn’s taxonomy predated the previous classification, and classified at a finer granularity of operations and is defined in terms of instruction parallelism.
  – This is why vector processing is usually referred to as SIMD or Single Instruction Multiple Data. Single instructions operate on vectors containing multiple elements. This is why AltiVec, MMX, SSE, etc... are often referred to as SIMD units.
  – The use of Program versus Instruction in most parallel programming discussions is because the parallelism occurs at the subroutine or program level. In the 90s, the trend moved this way with message passing being popular and vectorization falling into niche worlds like graphics processing.
  – This may come back though, especially with processor vendors being willing to take new risks on interesting parallel architectures lately.
Languages

- UPC, CaF, Titanium
- Matlab, ZPL, High Performance Fortran, NESL, Sawzall, CUDA, Brook
- Chapel, Fortress, X10
- Single Assignment C
- Sisal, Haskell + STM
- Erlang
PGAS = Parallel Global Address Space.

- Explicitly parallel with the SPMD model.
- Memory is globally addressable by the disjoint threads, even if it is not shared memory under the covers.
- This requires runtime support, such as the Berkeley GASNet or PNNL ARMCI, targeted by the compiler to support the executable.
- There is a logical partitioning of local to remote memory. This is apparent in how memory is indexed with mixed local and remote indices.
- UPC and CaF have commercial compiler support, UPC and Titanium also have open source options. UPC is based on C, Titanium on Java. http://upc.lbl.gov/
  http://titanium.cs.berkeley.edu/
Data-Parallel Languages

• Many codes that work on arrays are loop based, and parallelism is achieved by parallelizing loops (such as what we did for the GoL example).

• Data parallel notation allows you to make the loops implicit, giving the compiler the freedom to figure out the best strategy for implementing them, such as how data is distributed or how iterations are scheduled.

• Example:
  – Traditional: for (i=0; i<n; i++) { x[i] = y[i]+z[i]; }
  – Data parallel: x = y + z

• Currently, a hot topic for GPU or Cell-like multicore platforms. Historically of interest in massively parallel machines like the Connection Machine or Maspar.
The HPCS Languages

• These are really all encompassing languages.
• Chapel and X10 seem to provide support for more than one model of parallel programming. This makes them a bit big relative to other languages.
  – Chapel comes from Cray, X10 from IBM, Fortress from Sun. X10 is likely the best bet for a project here, and an initial implementation is available that runs on top of Java.
  – Chapel also has a compiler available, although it is not openly downloadable.
  – http://x10.sourceforge.net/
  http://chapel.cs.washington.edu/
Single-Assignment C

• Single assignment is an interesting concept, where variables can be assigned a value once and only once. This is similar to the purely functional model, as it eliminates potential side effects that can cause compilation to be a nightmare.

• Especially with respect to generating concurrent code. Everything is an array. C-based language.

• Mature compilers exist. http://www.sac-home.org/
Functional Languages

- Functional languages have long been regarded as good candidates for parallel compilation due to the lack of (or discouragement of) reliance on side effects to write programs.
  - **Haskell + STM**: Haskell with software transactional memory. See `Control.Concurrent.STM` library in GHC6.
  - **Alice ML**: Standard ML with threading and concurrency features in the type system.
  - **Sisal**: Single assignment functional language developed in the late 1980s/early 1990s that lives on in an open source implementation.
Erlang

- Channel communication based language inspired by Communicating Sequential Processes from Tony Hoare and the Actor model for concurrency.
- Functional language.
- Very mature compiler. Used in telecommunications industry. Currently quite popular in the blog world, although it remains to be seen if the fad fascination with it sticks or not.
- A wild option for a project: http://www.erlang.org/