Threads in C

David Chisnall

February 25, 2011
The Basic C Model

- One computer
- One process
- One thread
- One stack
Multithreaded Memory Layout (Again)

- Program Code: 0x00000000 - 0x0000f000
- Library Code: 0x00100000 - 0x00140000
- Static Data: 0x000fe000
- Heap Memory: 0xf0000000
- Stack 1: 0xffffffff
- Stack 2: 0xffffffff
Thread APIs

- Threads are not part of C99
- They are part of C1X, but there are currently no implementations of C1X
- On UNIX-like platforms, the POSIX Thread APIs are portable
- Other platforms there are different APIs
Creating Threads

```c
pthread_t thread;
pthread_create(&thread, NULL, start_function, arg);
```

- Creates a new stack
- Registers it for scheduling
- Sets instruction pointer in new thread to `start_function`
Combining Threads

```c
void * ret;  
pthread_join(thread, &ret);
```

- Waits for thread to finish
- Sets `ret` to the return value from the thread start function
Aside: Function Pointers

```c
void example(void) { printf("Stuff goes here\n") ; }
void user(void)
{
    // Call example
example();
    // Store a pointer to example:
    void (*funcPtr)(void) = example;
    // Call example via the function pointer
    funcPtr();
}
```
Thread Overhead

- Creating the stack
- Copying all of the thread-local variables
- Context switching if number of threads exceeds number of cores
- Synchronisation costs (including implicit cache coherency cost)

Sometimes adding threads makes things slower!
Example: Parallel Quicksort

- Recursive sorting algorithm
- Perform sub sorts in a new thread
Revision: Quicksort

1. Pick pivot point
2. Split array into ‘values lower than pivot’ and ‘values greater than pivot’
3. Recursively sort each sub-array.
void quicksort(int *array, int left, int right) {
    if (right > left) {
        int pivotIndex = left + (right - left)/2;
        pivotIndex = partition(array, left, right, pivotIndex);
        quicksort(array, left, pivotIndex-1);
        quicksort(array, pivotIndex+1, right);
    }
}
Making it Parallel

- Partition can’t (easily) be done in parallel
- Recursive calls can
Recursive Quicksort

```c
void quicksort(int *array, int left, int right)
{
    if (right > left)
    {
        int pivotIndex = left + (right - left)/2;
        pivotIndex = partition(array, left, right, pivotIndex);
        struct qsort_starter arg = {array, left, pivotIndex-2};
        pthread_t thread;
        // Create a new thread for one subsort
        pthread_create(&thread, 0, qsthread, &arg);
        quicksort(array, pivotIndex+1, right);
        // Wait for both to finish
        pthread_join(thread, NULL);
    }
}
```
Why The Structure?

```c
struct qsort_starter
{
    int  *array;
    int  left;
    int  right;
};

void* qsthreadthread(void *init)
{
    struct qsort_starter *start = init;
    quicksort(start->array, start->left, start->right);
    return NULL;
}
```

- Thread function only takes one (void*) argument
- If we want to pass more than one, we must pass a struct pointer
Is this Safe?

```c
struct qsort_starter arg = {array, left, pivotIndex-2};
pthread_t thread;
// Create a new thread for one subsort
pthread_create(&thread, 0, qsthread, &arg);
```

- Usually not - passing stack allocation to another thread
- Safe here because of the `pthread_join()` call.
Testing Speedup

$ time ./a.out
real   0m30.792s
user   0m30.552s
sys    0m0.222s

real (a.k.a wall clock time) - elapsed time from start to finish
user   CPU time scheduled to the process
sys    CPU time scheduled to the kernel to handle system calls for the process
Multithreading Performance

- Wall Clock Time
- CPU Time
Speedup

\[ S_p = \frac{T_1}{T_p} \]

- \( T_1 \) time for sequential algorithm
- \( T_p \) time for parallel algorithm with \( p \) processors
- \( S_p \) speedup with \( p \) processors

Wall clock time, not CPU time!
Efficiency

\[ E_p = \frac{S_p}{p} \]

- More useful metric
- Can compare \( E_p \) values independently of \( p \)
### Parallel Quicksort Speedup

<table>
<thead>
<tr>
<th>Threads</th>
<th>( p )</th>
<th>( T_p )</th>
<th>( S_p )</th>
<th>( E_p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>30.792</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>24.044</td>
<td>1.28</td>
<td>0.64</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>23.780</td>
<td>1.29</td>
<td>0.65</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>24.548</td>
<td>1.25</td>
<td>0.63</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>18.784</td>
<td>1.63</td>
<td>0.82</td>
</tr>
</tbody>
</table>

- Linear speedup: \( S_p = p \), \( E_p = 1 \)
- Superlinear speedup (very rare!) \( E_p > 1 \)
Why Sublinear?

- The partition step is serial
- Uneven workload distribution between threads (e.g. one thread sorting 998 elements, one sorting 2)
Amdahl’s Law

- Maximum speedup of a program is limited by the sequential portion
- For quicksort, this is the partition step
- Absolute best case, with infinite processors, is $O(n)$ for quicksort
Parallel Quicksort is Easy

- No data sharing between threads
- No communication between threads, except on exit
- Concurrent parts could be in separate processes
- Threads just eliminate some copying overhead
Questions?