Serial Equivalence's Impact on Space Bounds.

Arch D. Robison

SSG/DPD

Serial equivalence's impact on space bounds

Quicksort Example

Key to efficient implementation of Quicksort is semi-recursion.

- Recurse on smaller subproblem
- Iterate (tail-call) on bigger subproblem.

Worst-case bounds for serial execution:

- Recursion depth ≤ lg n.
- Iteration count ≤ n-1.

Cilk translation is trivial

- Spawn the recursive call
- Additional space = O(P lg n)

Familiar practices for avoiding space blowup still work.

```
void parallel_quicksort( T* first, T* last ) {
    while( last-first>QUICKSORT_CUTOFF ) {
        // Divide
        T* middle = divide(first,last);
        if( !middle ) return;
        // Now have two subproblems: [first..middle) and (middle..last)
        if( middle-first < last-(middle+1) )  {
            // Left problem [first..middle) is smaller, so spawn it.
            cilk_spawn parallel_quicksort( first, middle );
            // Solve right subproblem in next iteration.
            first = middle+1;
        } else {
            // Right problem (middle..last) is smaller, so spawn it.
            cilk_spawn parallel_quicksort( middle+1, last );
            // Solve left subproblem in next iteration.
            last = middle;
        }
    }
    // Base case
    std::sort(first,last);
}
```

Impact of Breaking Serial Semantics

In worst case, loop creates n-1 tasks before executing any of them.

- Serial version never did this.
- Additional space now O(n) instead of O(P lg n)
- Throttle mechanisms can offer some relief, but effect on space and parallelism can be hard to predict.

Familiar practices for avoiding space blowup may break.

```
void parallel_quicksort( T* first, T* last ) {
    task_group g;
    while( last-first>QUICKSORT_CUTOFF ) {
        // Divide
        T* middle = divide(first,last);
        if( !middle ) {
            g.wait();
            return;
        }
        // Now have two subproblems: [first..middle) and (middle..last)
        if( middle-first < last-(middle+1) )  {
            // Left problem [first..middle) is smaller, so spawn it.
            g.run([=]{quicksort( first, middle );});
            // Solve right subproblem in next iteration.
            first = middle+1;
        } else {
            // Right problem (middle..last) is smaller, so spawn it.
            g.run([=]{quicksort( middle+1, last );});
            // Solve left subproblem in next iteration.
            last = middle;
        }
    }
    // Base case
    std::sort(first,last);
    g.wait();
}
```

Thanks

Legal Disclaimer & Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804