Parallelization on Multi-Core CPUs
Amdahl’s Law

• suppose we parallelize a task using $n$ cores and $p$ is the proportion of the task that can be parallelized ($p - 1$ cannot be parallelized)
• then the speedup of the whole task is
  \[ \frac{1}{(1-p) + \frac{p}{n}} \]
• assuming infinite parallelism, the speedup is
  \[ \frac{1}{1-p} \]
• for example, if 90% of the work is parallelized, the maximum speedup is only 10
• one should make sure that every phase of one’s algorithm that depends on the input data size is parallelized
Parallelization Constructs and Libraries

- low-level: C++ threads, pthreads (threads, mutexes, barriers, condition variables)
- parallel patterns: parallel reduce, parallel for, fork/join parallelism
- parallel frameworks: TBB, OpenMP
Intel Thread Building Blocks

- Open Source library for parallelism and concurrency
- fairly nice for prototyping
- manages a pool of worker threads
- implements work stealing
- provides high-level abstractions
- enables nested parallelism
- large systems (e.g., database systems) will have their own framework
Thread-Local Storage

- in C++ variables can be annotated as `thread_local` (each thread has its own copy)
- however, sometimes it would be convenient to access the thread-local state of other threads
- `tbb::enumerable_thread_specific` allows this
Parallel Reduce

tbb::parallel_reduce(
    tbb::blocked_range<uint64_t>(0, n), // range
    0ull, // identity
    [](const tbb::blocked_range<uint64_t>& r, uint64_t init) {
        // accumulate
        for (uint64_t i=r.begin(); i!=r.end(); i++)
            init += array[i];
        return init;
    },
    [](uint64_t x, uint64_t y) { return x+y; }); // combine

tbb::blocked_range<Value>(Value begin, Value end, size_type grainsize=1);
Parallel For

```cpp
tbb::parallel_for(tbb::blocked_range<uint64_t>(0, n),
    [&] (const tbb::blocked_range<uint64_t>& r) {
        for (uint64_t i=r.begin(); i!=r.end(); i++)
            array[i] *= 2;
    });
```
Parallelization on Multi-Core CPUs

Partitioners

- `parallel_for` and `parallel_reduce` split the given range to enable parallel execution.
- There are multiple built-in partitioners:
  - `static_partitioner` splits work equally among threads up-front (no dynamic work stealing).
  - `simple_partitioner` splits the range as much as possible (e.g., until `grainsize` is reached).
  - `auto_partitioner` heuristic similar to `simple_partitioner`, but tries to avoid creating too many ranges (default).
Fork/Join Parallelism

- sometimes the amount of work to parallelize is not known upfront
- fork/join allows one to perform work on other threads ("fork"), and then to wait until these tasks are finished ("join")
- often recursive parallelism structure
Naive Merge Sort with Fork/Join (TBB)

```cpp
const ptrdiff_t limit = 1024;

template<class Iter>
void merge_sort(Iter first, Iter last) {
    if (last - first > limit) {
        Iter middle = first + (last - first) / 2;
        tbb::task_group g; // alternative: tbb::parallel_invoke
        g.run([&]{ merge_sort(first, middle); });
        merge_sort(middle, last);
        g.wait();
        std::inplace_merge(first, middle, last);
    } else {
        merge_sort_serial(first, last);
    }
}
```
Parallelization on Multi-Core CPUs

Analysis

- What is the maximum speedup (with infinite cores) for sorting $n$ elements?
- serial execution: $\log_2(n) \cdot n$
- rough upper bound:
  - the final merge is serial: $n$
  - lower bound for fraction of serial part $\frac{n}{\log_2(n) \cdot n} = \frac{1}{\log_2(n)}$
  - using Amdahl’s law the maximum speedup is $\frac{1}{\log_2(n)} = \log_2(n)$
  - for example, if $n = 2^{20}$ the upper bound is $\log_2(n) = 20$
- better upper bound:
  - parallel execution: $\sum_{i=0}^{\log_2(n)-1} \frac{n}{2^i} = n + \frac{n}{2} + \frac{n}{4} + \cdots < 2n$
  - for example, if $n = 2^{20}$ the upper bound is $\frac{20n}{2n} = 10$
- (both analyses assume that each level recursion level takes the same amount of time, which is not quite true in reality)
Parallelization on Multi-Core CPUs

Speedup, $n = 2^{20}$

![Graph showing speedup vs. number of threads]
Parallelization on Multi-Core CPUs

Speedup with 10 Threads

![Graph showing speedup with 10 threads vs data size in a log scale.]
Parallelization Overhead, $n = 2^{20}$
Parallel Merge (1)

```cpp
template<typename It>
void parallelMerge(It begin1, It end1, It begin2, It end2, It out) {
    tbb::parallel_for(ParallelMergeRange<It>(begin1, end1, begin2, end2, out),
        [&](ParallelMergeRange<It>& r) {
            std::merge(r.begin1, r.end1, r.begin2, r.end2, r.out);
        },
        tbb::simple_partitioner());
}

template<typename It>
struct ParallelMergeRange {
    It begin1, end1, begin2, end2, out;

    bool empty() const { return (end1 - begin1) + (end2 - begin2) == 0; }

    bool is_divisible() const {
        return std::min(end1 - begin1, end2 - begin2) > limit;
    }

    ParallelMergeRange(It begin1_, It end1_, It begin2_, It end2_, It out_) :
        begin1(begin1_), end1(end1_), begin2(begin2_), end2(end2_), out(out_) {};
```
Parallel Merge (2)

ParallelMergeRange(ParallelMergeRange& r, tbb::split) {
    if (r.end1-r.begin1 < r.end2-r.begin2) {
        // first range should be the larger one
        std::swap(r.begin1, r.begin2);
        std::swap(r.end1, r.end2);
    }
    It m1 = r.begin1 + (r.end1-r.begin1)/2;
    It m2 = std::lower_bound(r.begin2, r.end2, *m1);
    begin1 = m1;
    begin2 = m2;
    end1 = r.end1;
    end2 = r.end2;
    out = r.out + (m1-r.begin1) + (m2-r.begin2);
    r.end1 = m1;
    r.end2 = m2;
}
}; // struct ParallelMergeRange
Pitfalls in Parallel Code

- non-scalable algorithm
  - re-think algorithm
- load imbalances
  - break work into smaller tasks, dynamically schedule these between threads
- task overhead: managing tasks takes more time than the actual work
  - set a minimum per-thread tasks size (not too small, not to large)
References