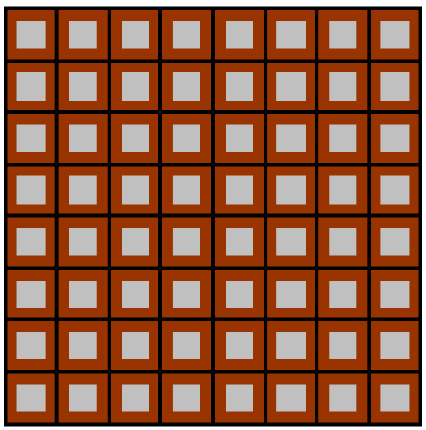
How is work split up across multiple cores?

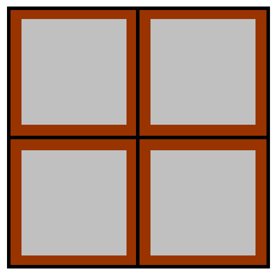
- Recall from section that the work per iteration can be badly imbalanced:

```
parallel_for(int i = 0; i < N; ++i)
  loop_body(i);   // Time(i) not constant
```

- How finely should the range (0, N) be split?   (grainsize)

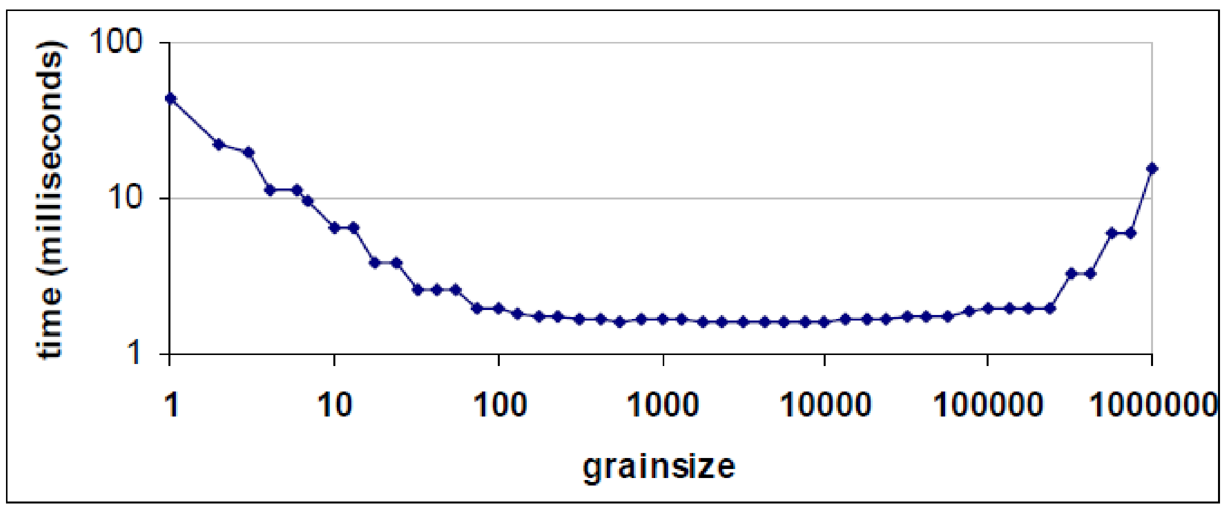


Case A



Work vs.
Overhead

Too fine ⇒ much overhead
Coarse ⇒ little parallelism
and imbalances hurt more



Task Scheduler: Dynamic Load Balancing

- The task scheduler's job is to keep all the processors busy
- Given a set of jobs J_1, J_2, …, J_n, and associated times t_1, t_2, …, t_n , and k processors, it is NP-hard to find the best way to assign jobs to processors
- The job is even harder for the task scheduler:
  - Job times are unknown (and hard to estimate)
  - Jobs may arrive over a period of time, not all at once
- TBB uses a heuristic called work-stealing
  - you can replace the default heuristic with others or your own
- CS225 concept: Deque (double-ended queue)

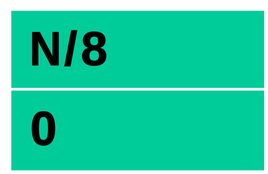
TBB Task Scheduling Heuristic

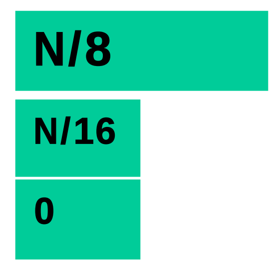
- Each available processor will maintain a deque of tasks (sub-ranges)
- P_1's deque:

- P_1 processes its deque bottom-up
  – if P_1 done, it steals from *top* of P_{random}'s deque

$\leq$ grainsize

- An improvement:



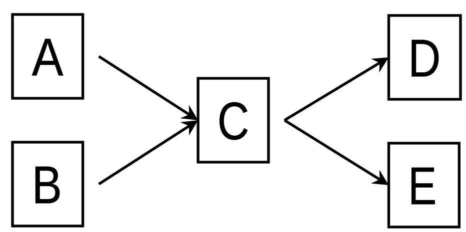


Other ways to exploit parallelism

- Suppose the iterations of a loop *must* be done sequentially
  — but tasks within each iteration can be done in parallel

```
for(int t = 0; t < N; ++t) {
  taskA(t);
  taskB(t);        in_parallel(taskA(t),taskB(t),taskC(t));
  taskC(t);
}
```

- What if tasks must also be in order: A → B → C → ... ?
  — Or, more generally:



- Answer: Pipelining!

The hazards of parallelism

- We have already seen how race conditions lead to incorrect behavior
- Consider the following example:

```
for(int i = 0; i < N; ++i)
  result = result ⊗ f(A[i]);  // ⊗ = some operation
```

- Correct approach to parallelization: reduction
- Alternate approach:

```
temp[NUM_THREADS] = {0, 0, ..., 0};
parallel_for(int i = 0; i < N; ++i)
  temp[thread_id()] = temp[thread_id()] ⊗ f(A[i]);
// in serial, merge temp array into a single result
```

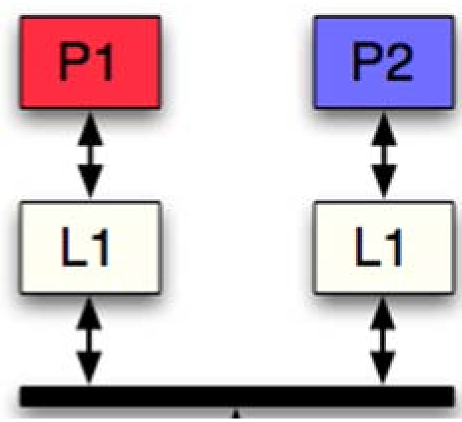
False Sharing

- The statement causing the problem is:

```
temp[thread_id()] = temp[thread_id()] ⊗ f(A[i]);
```

- There is *no race condition* here, but **`temp[0], temp[1], ...`** are all spatially local, and hence *within the same cache block*!

- When **`temp[0]`** is changed by thread 0, the entire block is marked dirty
  - this block must be sent to other processors that use this block



MESI protocol (a.k.a. *Illinois Protocol*)
- Modified
- Exclusive
- Shared
- Invalid

Multiple bouncing slows performance

Tips for MP6

- Do Task 1 first: make sure you have a fast version of the *serial* code!

- CSIL is running slow
  - the problem is not CPU utilization, it is I/O

- Give the following commands once you are in csil-linux-ts1:

```
cd /scratch
mkdir yourNetID
chmod 700 yourNetID
cd yourNetID
cp ~/mp6*.cxx ./
```

- Be sure to make your directory unreadable (chmod operation)!

- Be sure to copy your work over to your home folder regularly:

```
cp ./mp6*.cxx ~/
```