Shared-Memory Parallelism

ICS632: Principles of High Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

Outline

Parallelism

- The main concept of parallelism is that we have multiple devices on which we can execute tasks at the same time, and several tasks to execute
  - Computations on processing elements, or *processors*
- Sometimes the application is designed as a set of tasks
  - More common as parallelism awareness increases?
- More often we have to "extract" parallelism out of an application that is designed/implemented sequentially
  - Sometimes extracting parallelism can be straightforward, sometimes it can be more difficult
  - Conceptually simple ways to expose parallelism can be ineffective due to specifics of the underlying architecture
- Let's review fundamental terminology and concepts...

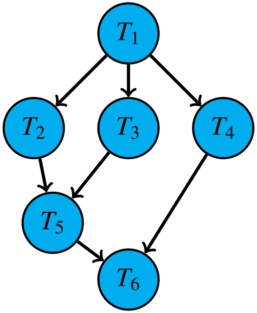
Concept #1: Work

- We consider a parallel computing platform that consists of p homogeneous, ideal processors
  - Heterogeneous parallel computing is an entire subject, which we'll talk about throughout the semester
- The work of an application is simply the amount of computation that has to be executed
  - Measured in flops, application-specific quantities, etc.
- Each processor has a compute speed measured in units of work performed per time unit
  - Also called compute rate, performance
- Sequential execution time is proportional to the work
  - Often one simply equates the work to the sequential execution time (1 unit of work = computation that can be done in 1 second on one processor)

Concept #2: Parallelism

- The parallelism of an application is the (maximum) number of processors that can be used concurrently
- If the parallelism is 1, then the application is sequential
  - Parallelism has not been "exposed", "extracted"
- If the parallelism is p throughout the whole execution then the application can use the whole platform at full performance

- Task graph with max parallelism = 3


$$\begin{tikzcd} & T_1 \arrow[dl] \arrow[d] \arrow[dr] & \\ T_2 \arrow[d] & T_3 \arrow[dl] & T_4 \arrow[ddl] \\ T_5 \arrow[dr] & & \\ & T_6 & \end{tikzcd}$$

Concept #3: Parallel Speedup

$$\text{parallel speedup} = \frac{\text{sequential wall-clock time}}{\text{parallel wall-clock time}}$$

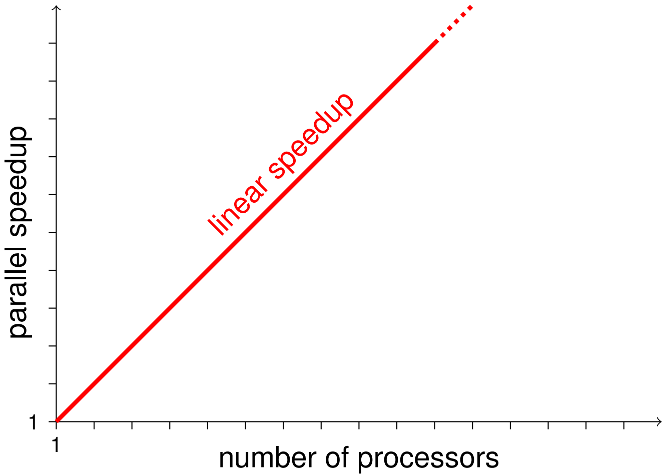
- Best case: parallel speedup = p
- If parallelism $< p$, then parallel speedup = $< p$
- Question: what is sequential wall-clock time?
  - The original purely sequential application?
  - The application once re-designed to expose parallelism, but executed with one processor?
    - Considered cheating by many, but useful to see how scalable your parallelization is

Concept #4: Parallel Efficiency

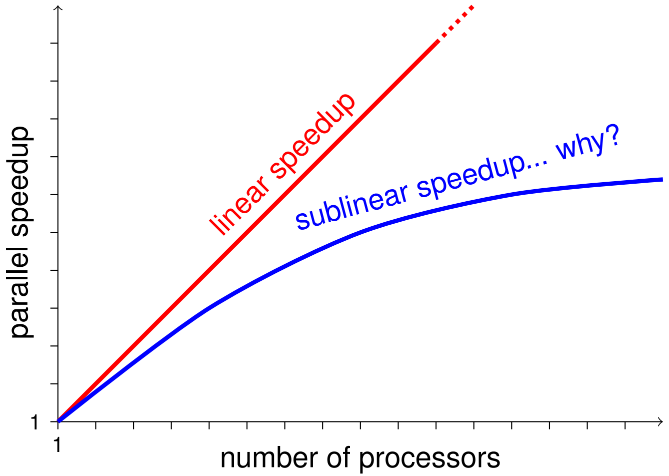
$$\text{parallel efficiency} = \frac{\text{parallel speedup}}{\text{number of processors}}$$

- Parallel efficiency quantifies how well the processors are being utilized by the parallel application
- Example: if with 4 processors the application goes 2x as fast, then the parallel efficiency is 50%
  - You can think of 50% of the processors being wasted when compared to a perfect execution with speedup = 4
  - Waste of hardware, power, heat,...
- Scalability: the ability to maintain high efficiency as the number of processors increases
  - Difficult to achieve to many applications

Speedup Curve



Speedup Curve



Amdahl's Law

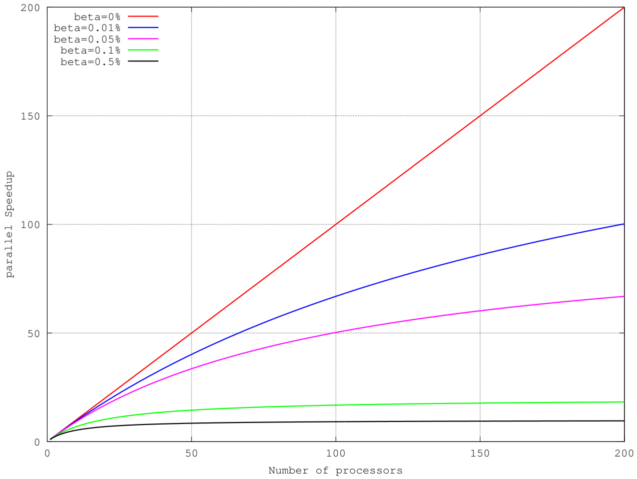
- Consider an application with sequential wall-clock time T_1
- A fraction α of this time is spent in a part of the code that cannot be parallelized
- A fraction $1 - \alpha$ of this time is spent in a part of the code that can be parallelized perfectly
- Therefore, the wall-clock time on p processors is:

$$T_p = \alpha \times T_1 + (1 - \alpha) \times \frac{T_1}{p}$$

- Which gives the parallel speedup:

$$\frac{T_1}{T_p} = \frac{1}{\alpha + (1 - \alpha)/p} \leq \frac{1}{\alpha}$$

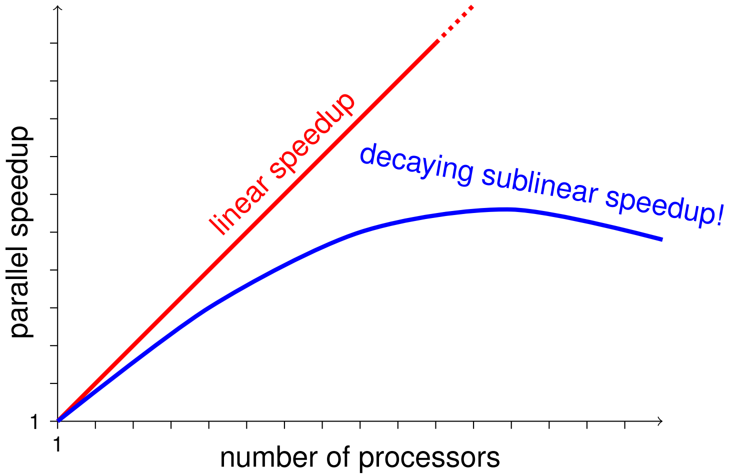
Amdahl's Speedups



Concept #5: Overhead / Idle Time

- Parallelization often comes with an overhead during which no useful or parallel work is accomplished
- Part of the execution is necessarily sequential (contributing to the α in Amdahl's law)
  - e.g., creating/terminating parallel tasks
  - e.g., serialized memory accesses
  - e.g., serialized access to critical sections of the code
- Tasks share resources thus leading to slow downs
  - e.g., tasks competing for cache space
- Tasks need to synchronize leading to wait time (idle time)
  - Inherent behavior with task graphs

The Effect of Overhead

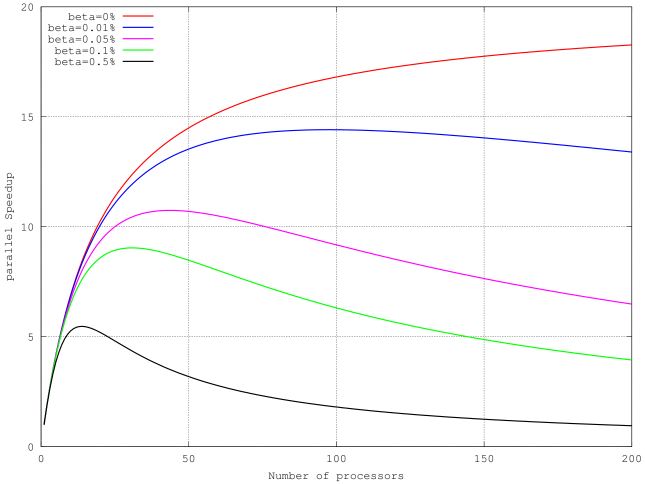


Parallelism - Overhead Tradeoff

- Well-known tradeoff for a given amount of work:
  - Having many tasks is great for parallelism
  - Having many tasks is not good for overhead
- You can think of overhead as an increasing $\alpha(p)$ in Amdahl's law
- Example:
  - 0.1% of the sequential execution time is due to code that is not parallelizable
  - For each processor beyond the first processor, an extra β% of the execution time is sequential
  - $\alpha(p) = 0.001 + (p-1) \times \beta$

$$\frac{T_1}{T_p} = \frac{1}{0.001 + (p-1) \times \beta + (1 - 0.001 + (p-1) \times \beta)/p}$$

Parallelism - Overhead Example



Concept #6: Load-Balancing

- A common cause of idle time (high β) is load imbalance
  - The term "load" refers to the portion of work that each processor accomplishes
- It is clear that evenly loaded processors are a good idea
  - Waiting for the last processor(s) to finish is not a good idea
- In some cases it is straightforward to balance the load
  - Matrix multiplication: each processor computes the same number of elements of the result matrix
- In some cases it is difficult
  - Applications whose work depends on the data, and thus cannot be precisely forecasted
  - e.g., n-body simulations, branch-and-bound searches
- Decades of load-balancing theory and practice
- A common practical solution: *work stealing*

Data- vs. Task-Parallelism

- Data-parallelism: multiple tasks all do the same thing on different parts of the same data
  - The Single Instruction Multiple Data (SIMD) model is quintessential data-parallelism
  - Example: adding two vectors together
- Task-parallelism: multiple tasks do different things on different data
  - Multiple Instructions Multiple Data (MIMD) model is quintessential task-parallelism
  - Example: a task graphs
- An application can have both task- and data-parallelism
  - Example: a task graph of data-parallel tasks
- These two terms are a bit fuzzy
  - Data-parallelism can look like task-parallelism (instruction-level? procedure-level? higher level?)

Recap

- Concepts:
  1. Work
  2. Parallelism
  3. Parallel speedup
  4. Parallel efficiency
  5. Overhead / Idle Time
  6. Load Balancing
- Amdahl's Law
  - A law of diminishing return for increasing parallelism
  - A law of decaying speedup if overhead is taken into account
- Data-parallelism and Task-parallelism

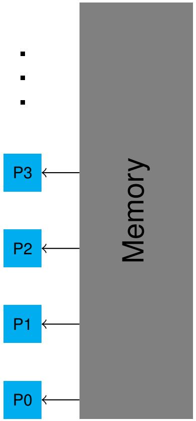
Outline

The PRAM Model

- In these lecture notes, we focus on processors that all have access to single (hopefully large) shared memory
- The oldest parallel computing model is PRAM
  - Parallel Random Access Machine
- An imperfect model that makes it possible to reason about parallel algorithms
  - A bit like a Turing Machine makes it possible to reason about algorithms
- The goal: obtain complexity results
- The goal is not to be representative of a real system
  - But results on an "ideal" system provide useful bounds

The PRAM Model

- Infinite memory, processors
- Processors read/write anywhere in memory on 1 cycle
  - Exclusive Read (ER): p procs can simultaneously read the content of p distinct memory locations
  - Concurrent Read (CR): p procs can simultaneously read the content of $p' < p$ memory locations
  - Exclusive Write (EW): p procs can simultaneously write the content of p distinct memory locations.
  - Concurrent Write (CW): p procs can simultaneously write the content of $p' < p$ memory locations

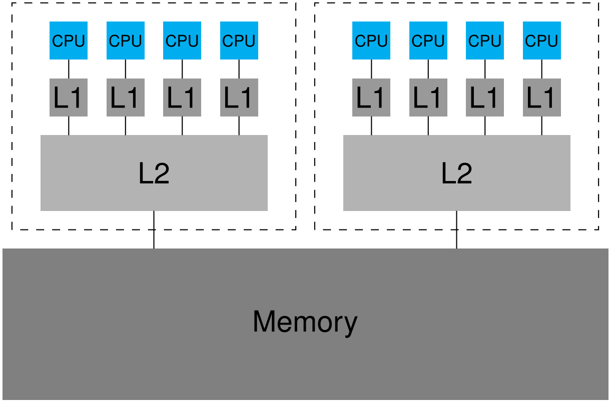


PRAM Algorithms

- Many algorithms have been developed assuming the PRAM model for fundamental operations
  - Prefix sum, scans, sorting, etc.
- There are fundamental "simulation" theorem
  - Example: Let A be an algorithm that runs in time t on a PRAM with p processors. One can simulate A on a PRAM with p' processors in time $O(t.p/p')$
  - Makes it possible to design an algorithm assuming any number of processors and "fold it" back to some reasonable number

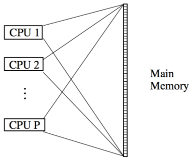
Modeling Locality

- PRAM assumes perfect locality, but reality is complex

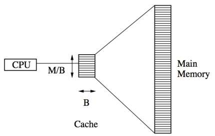


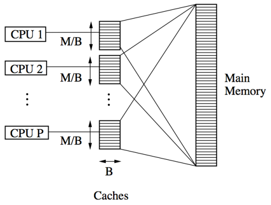
The Parallel External Memory (PEM) Model

PRAM Model



External Memory (EM) Model





Figures courtesy of Nodari Sitchinava

- The complexity metric is the amount of data transferred between the caches and the main memory
- Many algorithms have been developed and their complexity analyzed in the PEM model

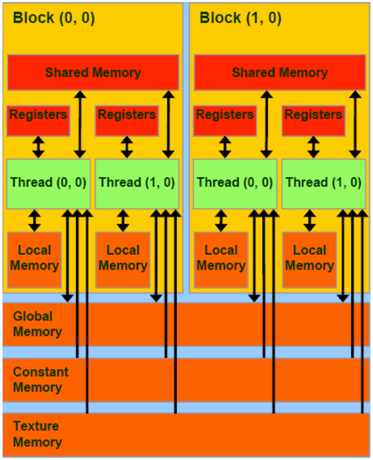
Cache-oblivious vs. Cache-aware

- The PEM model corresponds to a cache-aware approach
  - You know the size of the cache M and the size of the cache line (B)
  - You use M and B as parameters to design efficient algorithms and to analyze their performance
    - In practice code would discover these parameters at compile time
- The other approach is the cache-oblivious approach
  - Parameters M and B do not occur at all in the algorithm
  - But the algorithm is design such that it will achieve good locality regardless the values of M and B!
- Both cache-aware and cache-oblivious approaches have been investigated actively
  - In sequential and parallel contexts

Even More Complexity: GPUs

- The GPU architecture has many kinds of memory
- Processors (threads, really) are structured in groups
- Each memory has "features"
  - e.g., coalesced accesses to Global Memory are fast
  - e.g., shared memory has bank conflicts

The CUDA memory model:



Theory vs. Practice

- From PRAM, to PEM, to GPU models, there are plenty of fascinating theoretical results
  - See Dr. Sitchinava's courses for the theory
  - I will highlight some theoretical results now and again
- In this course we take a resolutely hands-on practical approach, through which we rediscover fundamental theoretical results through practice and experience
  - We write code, see what it does, and try to make it better
- Theory and practice are absolutely linked
  - Although there is a often a sharp and regrettable divide in the research communities

Outline

Threads

- A threads is a stream of instructions that can be scheduled as an independent unit
- Threads can be thought of as being "inside" a process
  - But Linux, for instance, doesn't really make a difference
- Threads created by the same process share the data segment and the heap
  - So they can "see the same memory"
- But each threads has its own program counter and stack
  - So they can "do different things"
- Threads are thus the way to achieve shared-memory parallelism (e.g., on multi-core platforms)
- Threads are scheduled by the O/S (we can run n threads on p cores, some possibly time-sharing cores)

Threads

- A threads is a stream of instructions that can be scheduled as an independent unit
- Threads can be thought of as being "inside" a process
  - But Linux, for instance, doesn't really make a difference
- Threads created by the same process share the data segment and the heap
  - So they can "see the same memory"
- But each threads has its own program counter and stack
  - So they can "do different things"
- Threads are thus the way to achieve shared-memory parallelism (e.g., on multi-core platforms)
- Threads are scheduled by the O/S (we can run n threads on p cores, some possibly time-sharing cores)

Thread Pitfalls

- Race Conditions: Program gives wrong results non-deterministically
  - Non-atomic executions problem
- Deadlocks: Program cannot make any progress as no thread executes instructions
  - Circular wait problem
- Livelocks: Program ping-pong's forever without making any progress
  - "Shall we dance" problem

Race Conditions

- Race conditions occur because of multi-step operations that update state (i.e., memory) that should not be interrupted
  - Including at the hardware level (e.g., `x++` is multi-step!)
- And yet the OS decides on context-switching, the cores run simultaneously
- How do we deal with this?
- We have CPUs that provide us with atomic instructions
  - Allow for synchronization abstractions (locks, semaphores, etc.) to create *critical sections*
  - Allow for efficient "lock free" implementations (try, check if it worked, if not retry)

Race Conditions

- Race conditions occur because of multi-step operations that update state (i.e., memory) that should not be interrupted
  - Including at the hardware level (e.g., `x++` is multi-step!)
- And yet the OS decides on context-switching, the cores run simultaneously
- How do we deal with this?
- We have CPUs that provide us with atomic instructions
  - Allow for synchronization abstractions (locks, semaphores, etc.) to create *critical sections*
  - Allow for efficient "lock free" implementations (try, check if it worked, if not retry)

Threads and Hardware

- We typically think of multi-threading as an "OS thing"
- But in fact processors have been designed with hardware support for threads
- One such support is to allow multiple threads to run on a single CPU core together in parallel

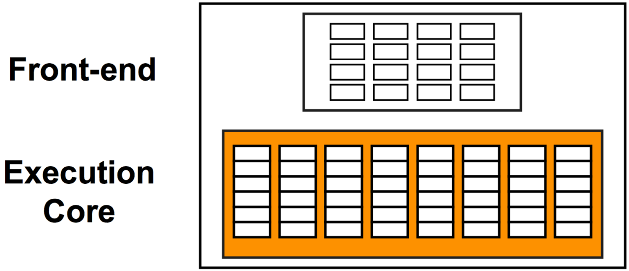
- Let's look at a simplified multi-issue (superscalar) CPU
  - Think of it as pipelining on steroids

Threads and Hardware

- We typically think of multi-threading as an "OS thing"
- But in fact processors have been designed with hardware support for threads
- One such support is to allow multiple threads to run on a single CPU core together in parallel

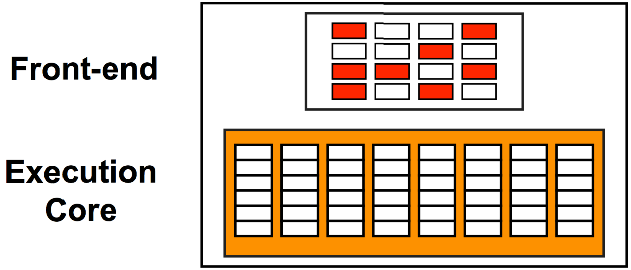
- Let’s look at a simplified multi-issue (superscalar) CPU
  - Think of it as pipelining on steroids

Superscalar Processor



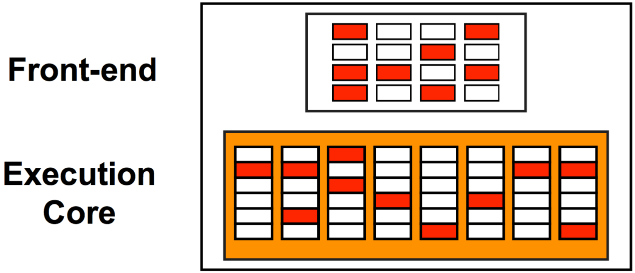
- Front-end: of 4 pipelines, each with 4 stages
    - 4 instructions can theoretically be issued to the execution core at each cycle
- Execution core: 8 pipelined execution units (each with 6 stages)
    - 8 instructions can theoretically be completed at each cycle

Superscalar Processor



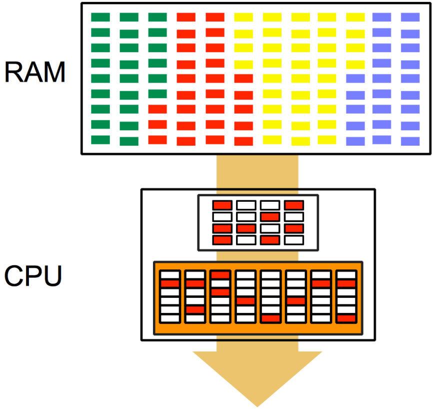
- The front-end is about to issue 2 instructions
- The cycle after it will issue 3
- The cycle after it will issue only 1
- The cycle after it will issue 2

Superscalar Processor



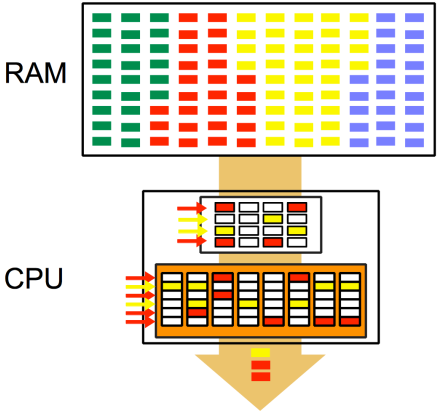
- At the current cycle, two functional units are used
- Next cycle one will be used
- And so on
- The while slots are "pipeline bubbles": lost opportunity for doing useful work
  - Due to low instruction-level parallelism in the program

Timesharing on Superscalar Processor



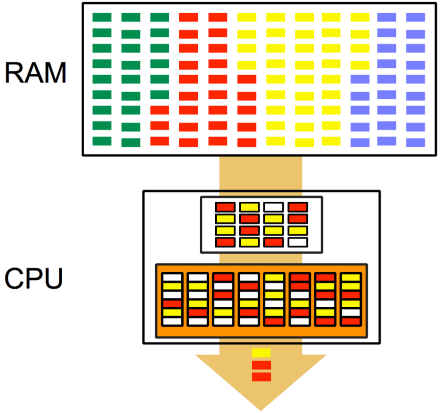
- 4 threads in RAM, context-switched in and out of the processor
- Only one threads on the CPU at a time
- A lot of wasted hardware due to pool Instruction Level Parallelism (ILP)

Super-threading



- Multiple threads in the processors at a time
- But only one thread per pipeline stage at each cycle
- More hardware complexity, less hardware waste, but still limited by ILP

Simultaneous Multi-threading



- Multiple threads in the processors even at the same pipeline stage
- More hardware complexity (a few % larger die), less hardware waste
- Marketed by Intel as "Hyperthreading" (2 threads together)
  - Mileage varies w.r.t. speedup
  - Can be enabled/disabled

POSIX Threads

- Standard C library (first standardized around 1995)
- Used for task-parallelism
- 60+ functions to do what you expect
  - Create, destroy, manage threads
  - Mutual exclusion, synchronization
  - Condition variables, semaphores for signaling
  - Many configuration bells and whistles
- Show of hands: should we go over Pthreads quickly
  - Some of you may already have experience, or may have seen this in an undergraduate/graduate course already
  - Typical ICS 432 material in our undergraduate curriculum
  - Linked off the course Web site

OpenMP (Open Multi-Processing)

- A higher level of abstraction than Pthreads
  - Much simpler and more convenient (parallelization can require adding only 1 line to the code)
  - Not explicitly task-oriented
  - Not as powerful (in fact, built on top of Pthreads)
- Used mostly for data-parallelism
- Supported by GCC since version 4.4
- Based on *directives* (`#pragma` clauses)
- Show of hands: should we go over OpenMP programming basics?
  - Some of you may already have experience, or may have seen this in an undergraduate/graduate course already
  - Typical ICS 432 material in our undergraduate curriculum
  - Linked off the course Web site

Cilk Threads

- Cilk is originally MIT research project
- Objective:
  - Convenient way to expose parallelism in the program
  - The runtime system does all sorts of optimization based on work-stealing, cache efficiency (all the research in Cilk)
- Cilk Plus is a commercial version implemented by Intel
  - Supported by GCC as of version 4.9
- Has features of both Pthreads and OpenMP, all better integrated into the language
  - Used both for task- and data-parallelism
- Let’s look at on-line Cilk Plus material...

Cilk Documentation/Tutorials/Examples

- Cilk Plus Keywords: https://www.cilkplus.org/tutorial-cilk-plus-keywords
- Cilk Plus Reducers: https://www.cilkplus.org/tutorial-cilk-plus-reducers
- Cilk Plus Array Notation: https://www.cilkplus.org/tutorial-array-notation
- Cilk Plus SIMD pragma: https://www.cilkplus.org/tutorial-pragma-simd
- Important to know that these are all "old" (but good!) concepts

Outline

Sequential Sorting

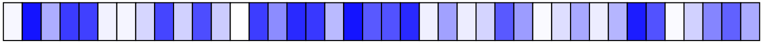
- We have an array of n elements that we wish to sort
- We can use the in-place sequential quick sort implementation in the standard C library
- Complexity $O(n \log n)$ on average

Sequential sorting

```
#include <stdlib.h>

#define SIZE 200
char array[SIZE];

int compare_ascending(const void *a, const void*b) {
  return (*((char *)a) - *((char *)b));
}

int main() {
  qsort(array, SIZE, sizeof(char), compare_ascending);
}
```

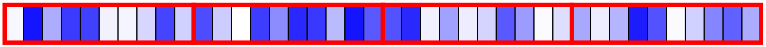
Simple Parallelization

- N elements to sort with p threads (on p cores)
    - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
    - Selecting the smallest element of each partial result



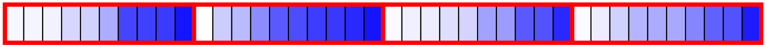
Simple Parallelization

- N elements to sort with p threads (on p cores)
  - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
  - Selecting the smallest element of each partial result



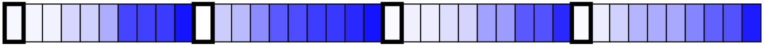
Simple Parallelization

- N elements to sort with p threads (on p cores)
  - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
  - Selecting the smallest element of each partial result



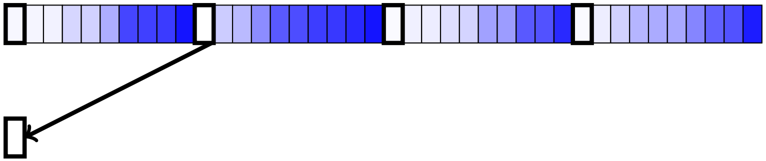
Simple Parallelization

- N elements to sort with p threads (on p cores)
  - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
  - Selecting the smallest element of each partial result



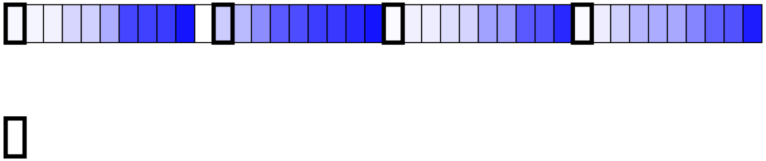
Simple Parallelization

- N elements to sort with p threads (on p cores)
  - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
  - Selecting the smallest element of each partial result



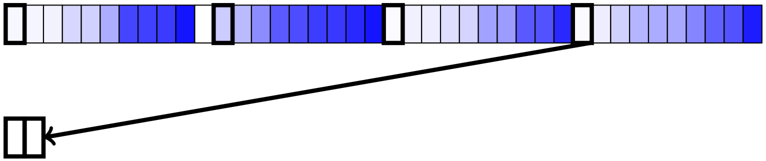
Simple Parallelization

- N elements to sort with p threads (on p cores)
  - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
  - Selecting the smallest element of each partial result



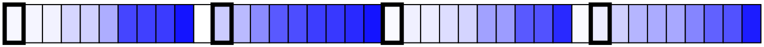
Simple Parallelization

- N elements to sort with p threads (on p cores)
  - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
  - Selecting the smallest element of each partial result



Simple Parallelization

- N elements to sort with p threads (on p cores)
  - For simplicity, assume p divides N
- Each thread sorts a N/p-element chunk of the array
- Then one thread does a multi-way merge of the n chunks
  - Selecting the smallest element of each partial result



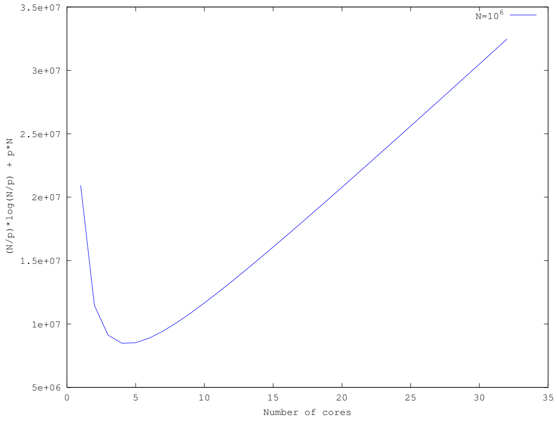
Implementing it in Practice

- Let’s look at the code with Pthreads...
  - OpenMP or Cilk wouldn’t be very different

Complexity Analysis

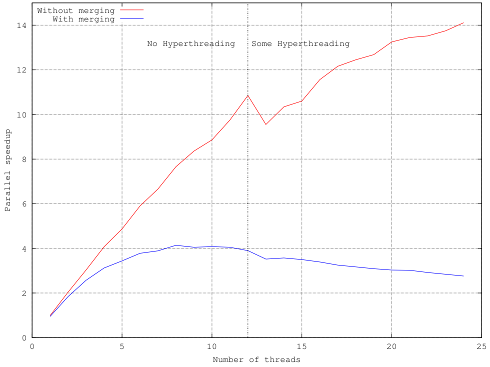
- Sort an array of p arrays of N/p elements with p cores: $O((N/p)\log(N/p))$
  - Assumes no overhead, no memory bottleneck, no locality problems, etc.
- Merge the results into the result array: $O(p \times N)$
  - Each of the N elements in the result array is determined as the smallest element among p elements
- Finally: $O((N/p)\log(N/p) + O(p \times N)$
  - If you think of Amdahl's Law, the second term looks bad: overhead that increases with parallelism!
    - And not much smaller than the 1st term unless N is huge
- Let's plot this function without the big-O's just for kicks...

"Complexity Plot"



Practical results

- Speedup on a 12-core Xeon 2.8GHz Linux box



- As expected, the sequential merging is a problem

Outline

Shared-Memory Programming

- Shared-memory programming is known to be difficult
  - How to ensure that programs will run correctly
  - How to achieve performance in a non-PRAM world?
  - How to expose all parallelism?
  - How to do all this in a way that's convenient?
  - Couldn't the compiler / language do all this for us?
- This class is mostly about distributed-memory programming, but we still have a short shared-memory programming assignment...