Logistics

- OpenMP exercise is posted
  - Deadline Friday, February 4, midnight
Shared Memory Parallel Programming

- Based on the ability of “threads of execution” to access shared memory
  - Shared addresses used in instructions as part of address space
  - Map to same physical address in physical memory

- Processes can do this with shared segments
  - OS calls set up shared segments in address space
  - Then can reference shared addresses (in the segment)

- Threads are more natural for shared memory programming
  - Threads of a process share process memory by default
  - Threads are lighterweight with respect to scheduling
  - Threads can be “user-level”

- Modern shared memory parallel programming is based on thread-based parallel (multi-threading) implementations
Shared Memory Languages / Libraries

- Early programming systems supported “co-routines”
- Cilk / Cilk++ ([https://www.cilkplus.org](https://www.cilkplus.org))
- Thread Building Blocks (TBB)
- OpenMP
- OpenCL
- Modern C++ (C++11)
  - Threading library
  - Parallel implementation of many standard algorithms
- Language-level based on “spawn” and “fork-join” constructs with hierarchical thread relationships
- Languages utilize multi-threading runtime libraries
  - POSIX threads (pthreads)
OpenMP

- An API for writing multi-threaded (parallel) applications
  - Set of compiler directives and library routines
  - Greatly simplifies writing multi-threaded code
  - Standardizes last 30 years of SMP practice

Goals of OpenMP

- Standardized
  - provide a parallelization standard among shared memory architectures
  - defined & endorsed by a number of hardware and software vendors

- Lean
  - only requires a few lines of directives to parallelize your code

- Easy to use
  - simple concept (as we will see later)
  - allows both fine-grained and coarse-grained parallelism

- Portable

- Supported by most major vendors
OpenMP Solution Stack
OpenMP Fork-Join Story

- OpenMP uses a fork-join model of parallel execution as a fundamental basis of the language
- All OpenMP programs begin as a single process
  - Master thread executes until a parallel region is encountered
- OpenMP runtime systems executes the parallel region by forking a team of (Worker) parallel threads
  - Statements in parallel region are executed by worker threads
- Team threads join with master at parallel region end
OpenMP – General Rules

- Most OpenMP constructs are compiler directives
- Directives inform the compiler
  - Provide compiler with knowledge
  - Usage assumptions
- Directives are ignored by non-OpenMP compilers!
  - Essentially act as comment for backward compatibility
- Most OpenMP constructs apply to structured blocks of code
  - A block of code with one point of entry at the top and one point of exit at the bottom
  - Loops are a common example of structured blocks
    - excellent source of parallelism
OpenMP PARALLEL Directive

- Specifies what should be executed in parallel:
  - A program section (structured block)
  - If applied to a loop, what happens is:
    - iterations are executed in parallel
    - do loop (Fortran)
    - for loop (C/C++)

- PARALLEL DO is a “worksharing” directive
  - Causes work to be shared across threads
  - More on this later
PARALLEL DO: Syntax

- **Fortran**
  - !$omp parallel do [clause [,] [clause ...]]
  - do index = first, last [, stride]
    - body of the loop
  - enddo
  - !$omp end parallel do

  The loop body executes in parallel across OpenMP threads

- **C/C++**
  - #pragma omp parallel for [clause [clause ...]]
  - for (index = first; text_expr; increment_expr) {
    - body of the loop
  }

- The loop body executes in parallel across OpenMP threads
Example: PARALLEL DO

- Single precision $a \times x + y$ (saxpy)

```
subroutine saxpy (z, a, x, y, n)
  integer i, n
  real z(n), a, x(n), y(n)
  !$omp parallel do
  do i = 1, n
    z(i) = a * x(i) + y(i)
  enddo
  return
end
```

What is the degree of concurrency?
What is the degree of parallelism?
Execution Model of PARALLEL DO

Master thread executes serial portion of code

Master thread enters *saxpy* routine

Master thread encounters *parallel do* directive

Creates slave threads (How many?)

Master and slave threads divide iterations of parallel do loop and execute them concurrently

Implicit synchronization: wait for all threads to finish their allocation of iterations

Master thread resumes execution after the do loop

Slave threads disappear

☐ Abstract execution model – a Fork-Join model!!!
Loop-level Parallelization Paradigm

- Execute each loop in parallel
  - Where possible
- Easy to parallelize code
- Incremental parallelization
  - One loop at a time
  - What happens between loops?
- Fine-grain overhead
  - Frequent synchronization
- Performance determined by sequential part (Why?)

```c
C$OMP PARALLEL DO
do i=1,n
............
enddo

alpha = xnorm/sum

C$OMP PARALLEL DO
do i=1,n
............
enddo

C$OMP PARALLEL DO
do i=1,n
............
enddo
```
Example: PARALLEL DO – Bad saxpy

- Single precision a\times x + y (saxpy)

```fortran
subroutine saxpy (z, a, x, y, n)
  integer i, n
  real z(n), a, x(n), y(n)
  !$omp parallel do
  do i = 1, n
    y(i) = a * x(i+1) + y(i+1)
  enddo
  return
end
```

What happens here?
How Many Threads?

- Use environment variable
  - `setenv OMP_NUM_THREADS 8` (Unix machines)
- Use `omp_set_num_threads()` function

```plaintext
subroutine saxpy (z, a, x, y, n)
  integer i, n
  real z(n), a, x(n), y(n)
  call omp_set_num_threads(4)
  !$omp parallel do
  do i = 1, n
    z(i) = a * x(i) + y(i)
  enddo
  return
end
```

Not a directive, but a call to the OpenMP library.
Assigning Iterations to Threads

- A parallel loop in OpenMP is a worksharing directive.
- The manner in which iterations of a parallel loop are assigned to threads is called the loop’s schedule.
- Default schedule assigns iterations to threads as evenly as possible (good enough for saxpy).
- Alternative user-specified schedules possible.
- More on scheduling later.
PARALLEL DO: The Small Print

- The programmer has to make sure that the iterations can in fact be executed in parallel
  - No automatic verification by the compiler

```fortran
subroutine noparallel (z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)
!
!$omp parallel do
!
!
do i = 2, n
  z(i) = a * x(i) + y(i) + z(i-1)
enddo
!
return
!
end
```

Do you see any problems here?
PARALLEL DO: The Small Print (2)

- The programmer has to make sure that the iterations can in fact be executed in parallel
  - No automatic verification by the compiler

subroutine noparallel (z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)
!
$omp parallel do
!$omp end parallel do
do i = 2, n
    z(i) = a * x(i) + y(i) + z(i-1)
enddo
return
end
PARALLEL Directive

- **Fortran**
  
  ```fortran
  !$omp parallel [clause [,] [clause ...]]
  structured block
  !$omp end parallel
  ```

- **C/C++**
  
  ```c
  #pragma omp parallel [clause [clause ...]]
  structured block
  ```
Parallel Directive: Details

- When a parallel directive is encountered, threads are spawned which execute the code of the enclosed structured block (i.e., the parallel region).
- The number of threads can be specified just like for the PARALLEL DO directive.
- The parallel region is replicated and each thread executes a copy of the replicated region.
Example: Parallel Region

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

```c
pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,A)
```

Is this ok?
Parallel versus Parallel Do

- Arbitrary structured blocks versus loops
- Coarse grained versus fine grained
- Replication versus work division (work sharing)

```$omp parallel do```
do I = 1,10
   print *, 'Hello world', I
```enddo```

PARALLEL DO is a work sharing directive

```$omp parallel```
do I = 1,10
   print *, 'Hello world', I
```enddo```

Output: 10 Hello world messages

```$omp parallel do```
do I = 1,10
   print *, 'Hello world', I
```enddo```

Output: 10*T Hello world messages

where T = number of threads
omp_set_num_threads(2);
#pragma omp parallel private(i, j, x, y, my_width, my_thread, i_start, i_end)
{
    my_width = m/2;
    my_thread = omp_get_thread_num();
    i_start = 1 + my_thread * my_width;
    i_end = i_start + my_width - 1;
    for (i = i_start; i <= i_end; i++)
        for (j = 1; j <= n; j++) {
            x = i/ (double) m;
            y = j/ (double) n;
            depth[j][i] = mandel_val(x, y, maxiter);
        }
    for (i = i_start; i <= i_end; i++)
        for (j = 1; j <= n; j++)
            dith_depth[j][i] = 0.5*depth[j][i] + 0.25*(depth[j-1][i] + depth[j+1][i])
}

What is going on here?
Work Sharing in Parallel Regions

- Manual division of work (previous example)

- OMP *worksharing* constructs
  - Simplify the programmers job in dividing work among the threads that execute a parallel region
    - *do* directive
      - have different threads perform different iterations of a loop
    - *sections* directive
      - identify sections of work to be assigned to different threads
    - *single* directive
      - specify that a section of code is to be executed by one thread only
        (remember default is replicated)
**DO Directive**

- **Fortran**
  
  ```fortran
  !$omp parallel [clause [,] [clause ...]]
  ...
  !$omp do [clause [,] [clause ...]]
  do loop
  !$omp enddo [nowait]
  ...
  !$omp end parallel
  ```

- **C/C++**
  
  ```c
  #pragma omp parallel [clause [clause ...]]
  {
  ...
  #pragma omp for [clause [clause] ... ]
  for-loop
  }
  ```
DO Directive: Details

- The DO directive does not spawn new threads!
  - It just assigns work to the threads already spawned by the PARALLEL directive
- The work→thread assignment is identical to that in the PARALLEL DO directive

```c
!$omp parallel do
  do I = 1,10
      print *, 'Hello world', I
  enddo
!$omp end parallel

!$omp parallel
  !$omp do
    do I = 1,10
      print *, 'Hello world', I
    enddo
  !$omp end do
!$omp end parallel
```
Coarser-Grain Parallelism

What’s going on here? Is this possible? When?
Is this better? Why?
SECTIONS Directive

- **Fortran**
  ```fortran
  !$omp sections [clause [,] [clause ...]]
  !$omp section
  code for section 1
  !$omp section
  code for section 2
  ...
  !$omp end sections [nowait]
  ```

- **C/C++**
  ```c
  #pragma omp sections [clause [clause ...]]
  {
      [#pragma omp section]
      block
      ...
  }
  ```
SECTIONS Directive: Details

- Sections are assigned to threads
  - Each section executes once
  - Each thread executes zero or more sections
- Sections are not guaranteed to execute in any order

```c
#pragma omp parallel
#pragma omp sections
{
    X_calculation();
    #pragma omp section
    y_calculation();
    #pragma omp section
    z_calculation();
}
```
OpenMP Fork-Join Summary

- OpenMP parallelism is Fork-Join parallelism
- Parallel regions have logical Fork-Join semantics
  - OMP runtime implements a Fork-Join execution model
  - Parallel regions can be nested!!
    - can create arbitrary Fork-Join structures
- OpenMP tasks are an explicit Fork-Join construct
OpenMP Overview: What is OpenMP?

- Model for parallel programming
  - Shared-memory parallelism
  - Supports loop, task, and data parallelism
  - Incremental parallelization

- Extensions to existing programming languages
  - Mainly by directives with a few library routines
  - Compiler-based
  - Fortran and C/C++

- Portable across shared-memory architectures

http://www.openmp.org
Motivation: Why should I use OpenMP?

Performance

Scalar Program Parallel Program

OpenMP

OpenMP + MPI

MPI only

Larger scales with multiple nodes

Time/Effort
Where should I use OpenMP?
Simple OpenMP Program

- Most OpenMP constructs are compiler directives
- Main focus of OpenMP is to parallelize loops
- OpenMP offers incremental parallelism

```
Serial Program:
void main()
{
    double Res[1000];
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

```
Parallel Program:
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```
Outline

- Execution Model
  - Parallel regions: team of threads
  - Syntax
  - Environment variables
  - Runtime library routines

- Work-sharing directives
  - Which thread executes which statement or operation?
  - Synchronization constructs, e.g., critical sections

- Data environment and combined constructs
  - Private and shared variables
  - Combined parallel work-sharing directives

- Summary of OpenMP API
OpenMP Programming Model

- OpenMP is a shared memory model.
- Workload is distributed between threads
- Threads interact through shared memory
  - Threads can have local data
- Threads synchronize implicitly and explicitly
  - Implicit synchronization built into parallel constructs
  - Explicit thread synchronization to protect shared data
- OpenMP implementations utilize a runtime system
  - Thread management
  - Runtime library
OpenMP Execution Model

- Fork-join model of parallel execution
- Begin execution as a single thread (master thread)
- Start of a parallel construct:
  - Master thread creates team of threads
- Completion of a parallel construct:
  - Threads in the team synchronize with implicit barrier
- Only master thread continues execution
OpenMP Execution Model (2)
OpenMP Parallel Region

- Indicates that block of code in the region should be executed by multiple threads in parallel
  - Each thread executes the same code redundantly!

- C/C++:
  
  #pragma omp parallel [ clause [ clause ] ... ] new-line
  - Structured-block
  - Clause can be one of the following:
    - private( list)
    - shared( list)
**OpenMP Parallel Region (2)**

C / C++:

```c
#pragma omp parallel
structured block
/* omp end parallel */
```
OpenMP #pragma Directives for C

- **Format:**
  
  ```c
  #pragma omp directive_name [ clause [ clause ] ... ] new-line
  ```

- **Conditional compilation**
  
  ```c
  #ifdef _OPENMP
      block,
  #endif
  ```

- **Case sensitive**

- **Header file for library routines:**
  
  ```c
  #ifdef _OPENMP
      #include <omp.h>
  #endif
  ```
OpenMP Data Scope Clauses

- private ( list )
  - Variables in list are private to each thread in a team

- shared ( list )
  - Variables in list are shared among all the threads in a team

- Default is shared
  - Stack (local) variables are PRIVATE

- Loop control variable of parallel OMP
  - DO (Fortran)
  - for (C)
  - is PRIVATE
OpenMP Environment Variables

OMP_NUM_THREADS
   – sets the number of threads to use during execution
   – when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use

```bash
setenv OMP_NUM_THREADS 16 [csh, tcsh]
export OMP_NUM_THREADS=16 [sh, ksh, bash]
```

OMP_SCHEDULE
   – applies only to `do/for` and `parallel do/for` directives that have the schedule type `RUNTIME`
   – sets schedule type and chunk size for all such loops

```bash
setenv OMP_SCHEDULE GUIDED,4 [csh, tcsh]
export OMP_SCHEDULE= GUIDED,4 [sh, ksh, bash]
```
OpenMP Runtime Library

- Query functions
- Runtime functions
  - Run mode
  - Nested parallelism
- Lock functions
- C/C++
  - Add #include <omp.h>
OpenMP Runtime Library (2)

- `omp_get_num_threads()`
  - Returns the number of threads currently in the team executing the parallel region from which it is called

- `omp_get_thread_num()`
  - Returns the thread number, within the team, that lies between 0 and `omp_get_num_threads() - 1`, inclusive
  - Master thread of the team is thread 0
Work Sharing Directives

- Which thread executes which statement or operation?
- And when?
  - Work-sharing constructs
  - Master and synchronization constructs
- Organization of the parallel work!!!
OpenMP Work Sharing Constructs

- Divide the execution of the enclosed code region among the members of the team
- Must be enclosed dynamically within a parallel region
- They do not launch new threads
- No implied barrier on entry
  - sections directive
  - do directive (Fortran)
  - for directive (C/C++)
OpenMP for Directive

- Immediately following loop executed in parallel
- C/C++:
  
  #pragma omp for [ clause [ clause ] ... ] new-line

- For loop
#pragma omp parallel private(f) 
{ 
  f=7; 
#pragma omp for 
  for (i=0; i<20; i++) 
    a[i] = b[i] + f * (i+1); 
} /* omp end parallel */
OpenMP for Directive (3)

- Clause can be one of the following:
  - `private( list)`
  - `reduction( operator: list)`
  - `schedule( type [ , chunk ] )`
  - `nowait (C/C++: on #pragma omp for)`

- Implicit barrier at end of unless nowait is specified
  - If nowait is specified, threads do not synchronize at the end of the parallel loop

- schedule clause specifies how iterations of the loop are divided among the threads of the team.
  - Default is implementation-dependent
OpenMP schedule Clause

- **static**: Iterations are divided into pieces of a size specified by chunk.
  - The pieces are statically assigned to threads in the team in a roundrobin fashion in the order of the thread number.
  - Default chunk size: one contiguous piece for each thread.

- **dynamic**: Iterations are broken into pieces of a size specified by chunk.
  - As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. Default chunk size: 1.

- **guided**: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  - chunk specifies the smallest piece (except possibly the last).
  - Default chunk size: 1. Initial chunk size is implementation dependent.

- **runtime**: The decision regarding scheduling is deferred until run time.
  - The schedule type and chunk size can be chosen at run time by setting the OMP_SCHEDULE environment variable.
  - Default schedule: implementation dependent.
Loop Scheduling Types

- **Static**
- **Dynamic (3)**
- **Guided (1)**
OpenMP Synchronization

- Implicit Barrier
  - Beginning and end of parallel constructs
  - End of all other control constructs
  - Implicit synchronization can be removed
  - With nowait clause

- Explicit

- Critical
OpenMP Atomic

- OpenMP atomic operations allow multiple threads to safely update a shared numeric variable.

- An atomic operation applies only to the single assignment statement that immediately follows it.

C / C++:
```c
cnt = 0;
f = 7;
#pragma omp parallel
{
#pragma omp for
    for (i = 0; i < 20; i++) {
        if (b[i] == 0) {
            #pragma omp atomic
            cnt = cnt + 1;
        } /* endif */
        a[i] = b[i] + f * (i + 1);
    } /* end for */
} /* omp end parallel */
```
OpenMP Control Structures Summary

- Parallel region construct
  - parallel

- Work-sharing constructs
  - sections
  - do (Fortran)
  - for (C/C++)

- Combined parallel work-sharing constructs
  - parallel do (Fortran)
  - parallel for (C/C++)

- Synchronization constructs
Shared Memory Parallelism and Performance

- Shared memory parallel programming model creates thread-level concurrency and uses synchronization to ensure correctness.

- To achieve performance, it is necessary to think about how memory is being accessed.

- Hierarchical memory with caches:
  - Locality of reference is important.
  - Maximize cache performance.
  - Depends on knowledge of cache hierarchy.
Memory Hierarchies

- L3 cache is also referred to as last level cache (LLC)
- L1 cache is divided between instruction (L1i) and data (L1d) cache
- Inside the core, data is stored in registers
- Traditionally, in multi-core processors, L3 and L2 are (typically) shared among all cores, and L1 is local to each core
Modern Cache

- Intel Skylake architecture
- L1i/L1d cache
  - 32 KB per core, 8-way set associative
  - 64 Byte/cycle load, 32 Byte/cycle store
  - 4 or 5 cycle access (depending on how address is calculated)
- L2 cache
  - 256 KB per core (unified), 4-way set associative
  - 64 Byte/cycle (to L1 cache)
  - 12 cycle access
- L3 cache
  - 2 MB per core (but shared), 16-way associative
  - 32 Byte/cycle
  - 42 cycle access
- L4 cache (or embedded DRAM (eDRAM), side cache).
  - 64/128 MB per package
  - 32 Byte/cycle read/write (but runs on a separate eDRAM clock)
Locality

- Data locality
- Temporal
  - A data is reused within a short time frame
- Spatial
  - If data in location $i$ is used (e.g., array), data in location $i+1$ is also likely to be used
Temporal Locality

\( \tau = \) some factor which captures how much faster cache is vs. DRAM (includes both bandwidth and latency)
\( \kappa = \) cache “reuse” rate

\( S(\tau, \kappa) = \frac{T_{\text{mem}}}{T_{\text{avg}}} \)

\( T_{\text{cache}} = \frac{T_{\text{mem}}}{\tau} \rightarrow T_{\text{mem}} = T_{\text{cache}} \tau \)

\( T_{\text{avg}} = \kappa T_{\text{cache}} + (1 - \kappa) T_{\text{mem}} \)

\( S(\tau, \kappa) = \frac{T_{\text{cache}} \tau}{(\kappa T_{\text{cache}} + (1 - \kappa) T_{\text{cache}} \tau)} = \frac{\tau}{(\kappa + (1 - \kappa) \tau)} \)

\( \square \) What does this remind you of?
Temporal Locality

![Graph showing temporal locality with curves for different τ values.](image)
Locality

- How do we increase temporal locality?
- Cache “blocking” (or tiling).
  - Minimizes the chance that data loaded into cache will not be evicted
- What if data is “streamed?”
- Spatial locality?
Spatial Locality (cache lines)

- Data transfer between DRAM and cache occurs in cache line granularity
- This is done to reduce latency and to take advantage of spatial locality
  - $T = \alpha + \beta L$
  - $\alpha$ = time between data request and delivery; $L$ = cache line size; $\beta = 1 / \text{bandwidth}$
  - Also known as alpha beta model
- Let’s say latency is 80 ns and bandwidth = 40 GB/s
  - Typical cache line size is 64 Bytes
  - Time to load a cache line = 81.6 ns
  - Without cache lines (and reading 8 bytes at a time)?
    - $8 \times (80 \text{ ns} + 8 \text{ Bytes} / 40 \text{GB/s}) = 641.6 \text{ ns}$
Spatial Locality

- What if we don’t need all 64 Bytes?
  - Let’s say we need to read 64 bytes but they are separated by 56 Bytes - i.e., first 8 bytes of 8 cache lines
  - \[ T = 8 \times (80 \, \text{ns} + 64 \, \text{Bytes} / 40 \, \text{GB/s}) = 652.8 \, \text{ns} \, (\text{vs.} \, 641.6 \, \text{ns}) \]

- Now, let’s look at the streaming case.
  - What is the cache hit rate? (this is not the same as the reuse rate described above)
  - Assuming cache line arrives in cache before the 2nd data access
  - \[ \gamma = (64 - 8) / 64 = 0.875 \]

- If you have enough parallelism to hide the latency (i.e., you are limited by only the bandwidth), your potential performance is \( \frac{1}{8} \) what it could be if you don’t use all 64 bytes
Write Allocate

- What happens in writing data (as opposed to reading)?
  - Most LLC use write-back - if hit in cache, cache line is modified and written to memory only when evicted
  - On a cache miss - entire cache line is brought in (write allocate) which causes 2x data traffic

- Write allocate is required due to hardware design - registers can only communicate with L1

- If data is not re-used (i.e., no read from the cache after writing), it’s an unnecessary penalty - is there something that can be done?
  - Non-temporal stores - special store instructions that bypass all cache levels and directly write to memory. There is also typically a write combine buffer that bundles non-temporal stores to better utilize the memory system
  - Cache line zero - zero-out a cache line and mark it as modified without a read, and data is written to memory when evicted
**Associativity**

- Direct-mapped - memory location that are multiple of cache line size apart are always mapped to the same cache location
  - Easy to implement - just mask out the most significant bits
  - Prone to cache thrashing (i.e., conflict misses) - applications with strided access map to the same cache location at every iteration

- Fully-associative - any memory location can be mapped to any location in the cache
  - Difficult to build large, fast, and fully-associative caches due to book-keeping
  - Every entry in the cache must be checked to see if a new memory request is already in cache
  - More flexible usage
Direct-Mapped Cache
N-Way Set-Associative

- Reduce conflict misses without having huge bookkeeping overhead
- Cache is divided into N direct-mapped caches (equal in size).
- Set associative typically between 2 to 48 on modern processors.
2-Way Set-Associative
Prefetching

- Even if you improve spatial locality using cache lines, latency still exists on the first miss

- Making the cache lines longer will help reduce the latency occurrence
  - Counterproductive when there is irregular access pattern
  - Current “sweet spot” seem to be 64 or 128 Byte cache lines
- Prefetching can help with this.
Prefetching

- “Fetch” the cache line before it is requested
  - Prefetching instructions (software) inserted by the programmer or the compiler
  - Hardware prefetcher tries to “predict” by studying the access pattern

- Software prefetching
  - Instructions are “hints” to the architecture - not guaranteed to work
  - Increases instruction count - could degrade performance (instruction cache misses)
  - “Timing” the prefetch is difficult

- Hardware prefetching
  - Specialized hardware
  - Trades bandwidth for latency
Prefetching

- Hardware Prefetching
  - Prefetching requires resources that are limited by design
  - The memory system must be able to sustain a certain number of outstanding prefetches (i.e., pending prefetch requests)
  - Otherwise, the memory pipeline will stall and the latency cannot be hidden completely
  - How many outstanding prefetches are required? (Hint: Little’s Law)

- Number of cache lines that can be transferred during time T is the number of prefetches that the processor must be able to sustain

\[ L = \text{cache line size} \quad \beta = 1 / \text{bandwidth} \quad \alpha = \text{latency} \]
\[ t = \alpha + L \beta \quad \text{(time it takes to fetch a cache line)} \]
\[ \text{# of cache lines transferred in time } t = \left( t \times \text{bandwidth} / L \right) = t / L \beta \]
\[ P = t / (L \beta) = (\alpha + L \beta) / L \beta = 1 + \alpha / (L \beta) = \approx \alpha / (L \beta) \]

Concurrency \[ P = \text{Latency} \times \text{bandwidth} = \alpha / (L \beta) \quad \text{(Little’s Law)} \]
Prefetching

- If you also doing some amount of compute, some of that can be used to hide the memory latency.
- In such a case, fewer outstanding fetches will suffice to saturate the bandwidth.
RISC vs. CISC

- CISC (Complex Instruction Set Computer)
  - Complex, more powerful instructions
  - Requires larger hardware for decoding the instructions
  - Reduces the number of instructions

- RISC (Reduced Instruction Set Computer)
  - Simple instructions that can be decoded quickly and executed rapidly (i.e., fewer cycles per instruction, therefore higher clock rate possible)

- Intel is a well-known CISC architecture
  - This is only partially true. Machine code is CISC, but internally, they are converted to a set of micro-ops, which resemble RISC instructions