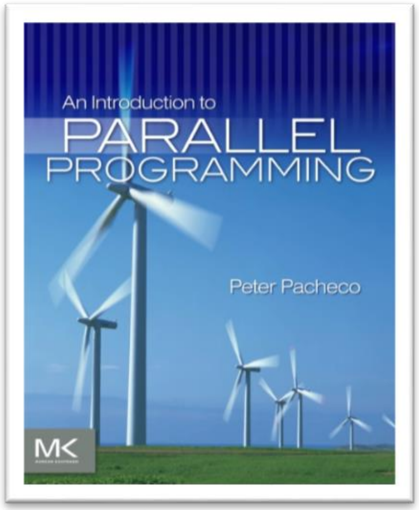
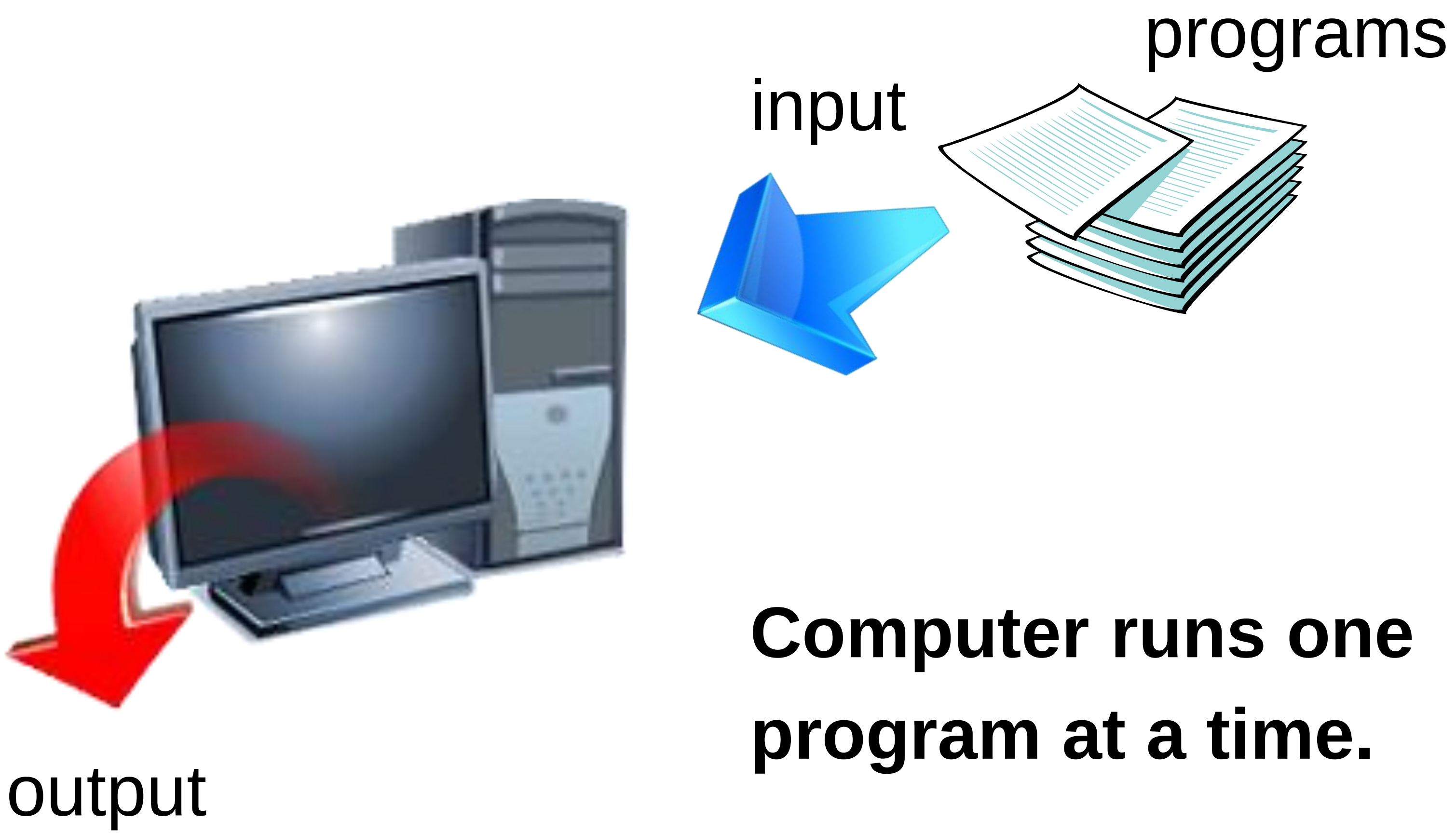
Introduction to Parallel Programming

Center for Institutional Research Computing



Slides for the book "An introduction to Parallel Programming", by Peter Pacheco (available from the publisher website): *http://booksite.elsevier.com/9780123742605/*

Serial hardware and software



Computer runs one program at a time.

Why we need to write parallel programs

- Running multiple instances of a serial program often isn't very useful.
- Think of running multiple instances of your favorite game.
- What you really want is for it to run faster.

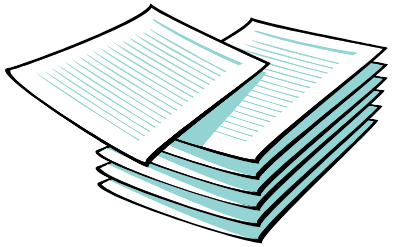
How do we write parallel programs?

- Task parallelism
  - Partition various tasks carried out solving the problem among the cores.
- Data parallelism
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on it’s part of the data.

Professor P

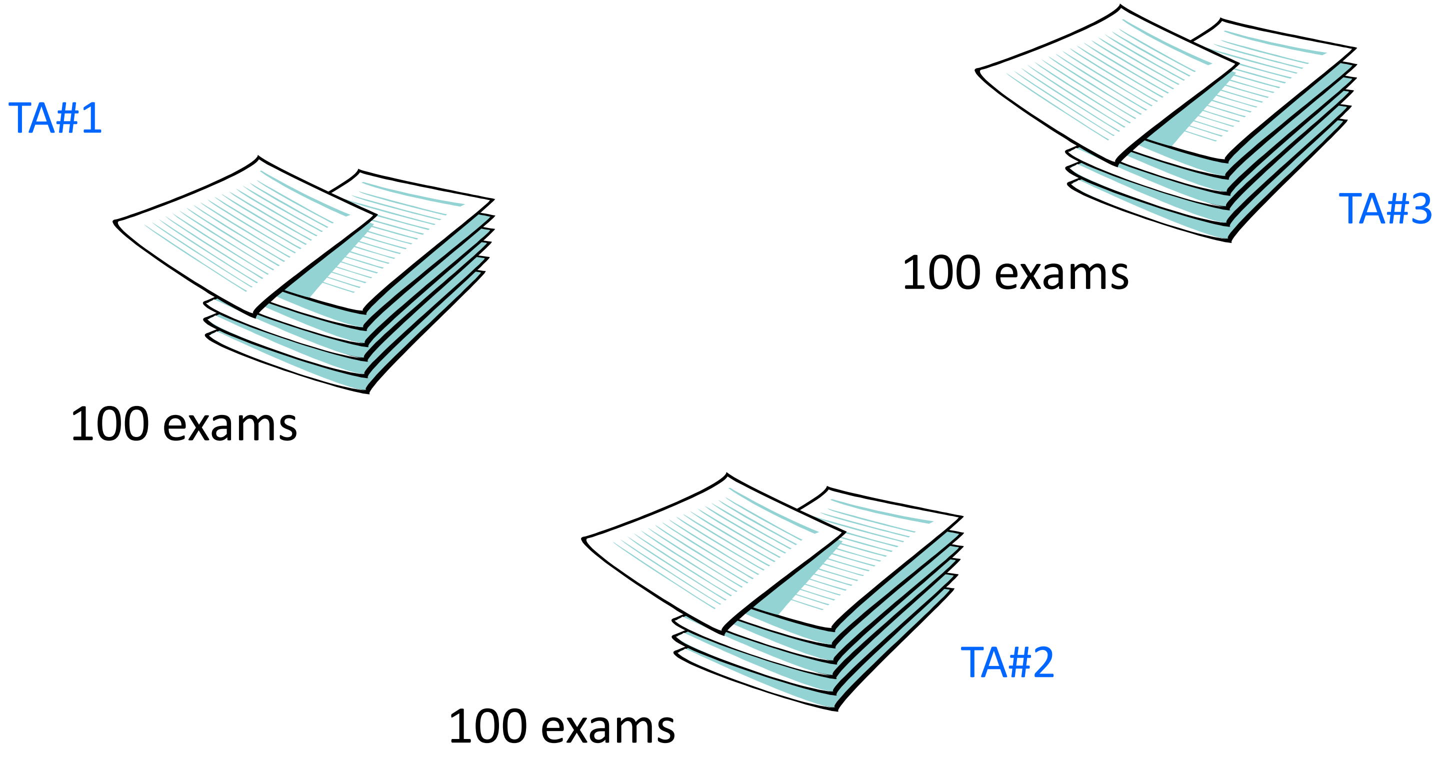
15 questions

300 exams



Professor P's grading assistants

Division of work – data parallelism



Division of work – task parallelism

TA#1

Questions 1 - 5

or Questions 1 - 7

TA#3

Questions 11 - 15

or Questions 12 - 15

TA#2

Questions 6 - 10

or Questions 8 - 11

Partitioning strategy:
- either by number
- Or by workload

Coordination

- Cores usually need to coordinate their work.
- Communication – one or more cores send their current partial sums to another core.
- Load balancing – share the work evenly among the cores so that one is not heavily loaded.
- Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.

What we’ll be doing

- Learning to write programs that are explicitly parallel.
- Using the C language.
- Using the OpenMP extension to C (multi-threading for shared memory)
  - Others you can investigate after this workshop:
    - Message-Passing Interface (MPI)
    - Posix Threads (Pthreads)

Essential concepts

- Memory
- Process execution terminology
- Configuration of Kamiak
- Coding concepts for parallelism - Parallel program design
- Performance
- OpenMP

.c programs we will use

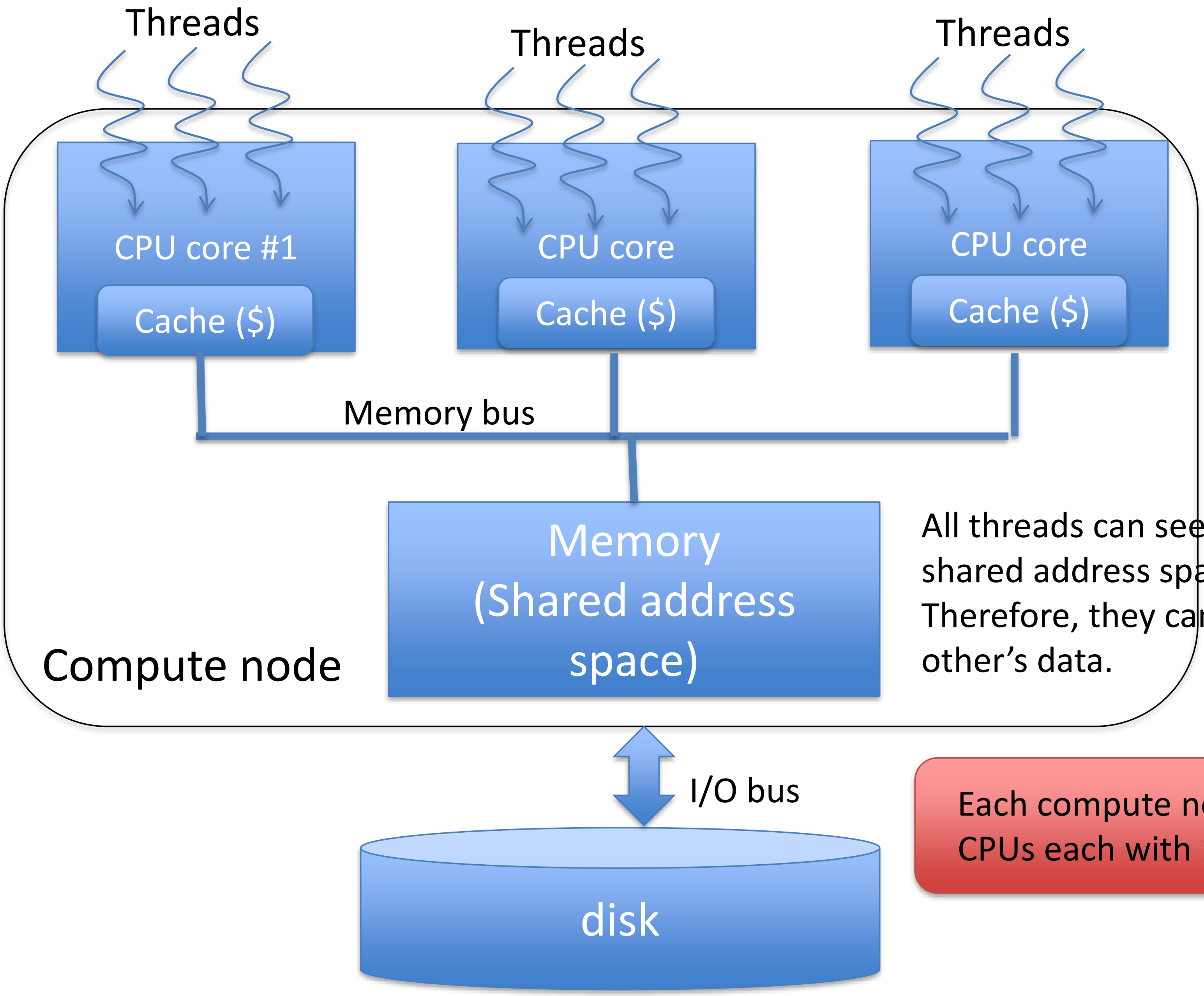
- loop.c
- sync.c
- sumcomp.c
- matrix_vector.c (for independent study)

- Step 1: log into kamiak
- Step 2: run command: training

Memory

- Two major classes of parallel programming models:
  - Shared Memory
  - Distributed Memory

Shared Memory Architecture

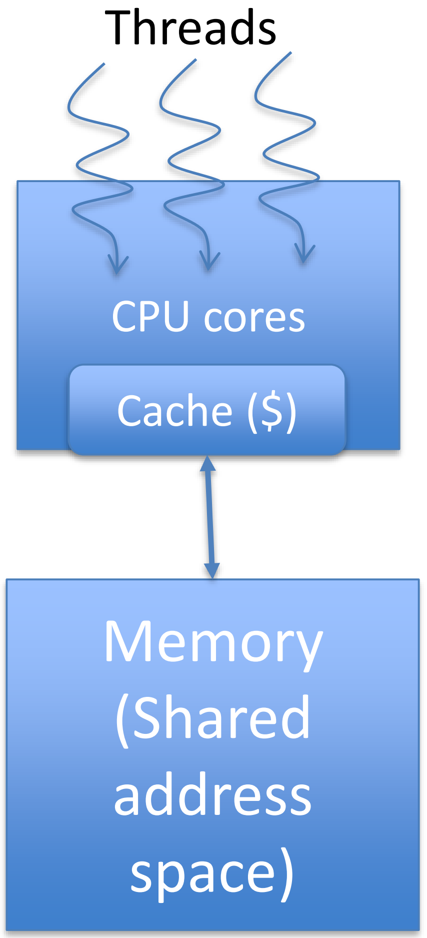


All threads can see a single shared address space. Therefore, they can see each other's data.

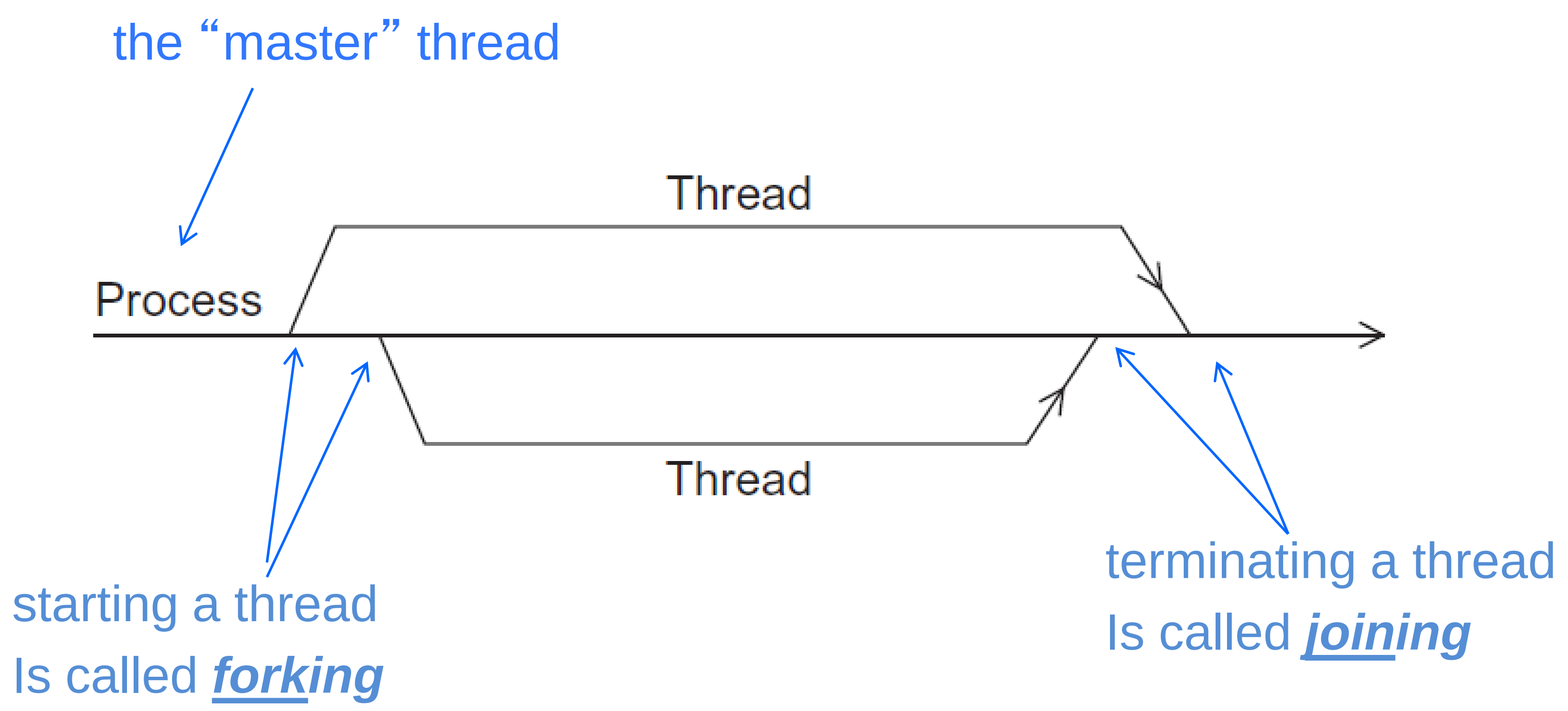
Each compute node has 1-2 CPUs each with 10-14 cores

Multi-Threading (for shared memory architectures)

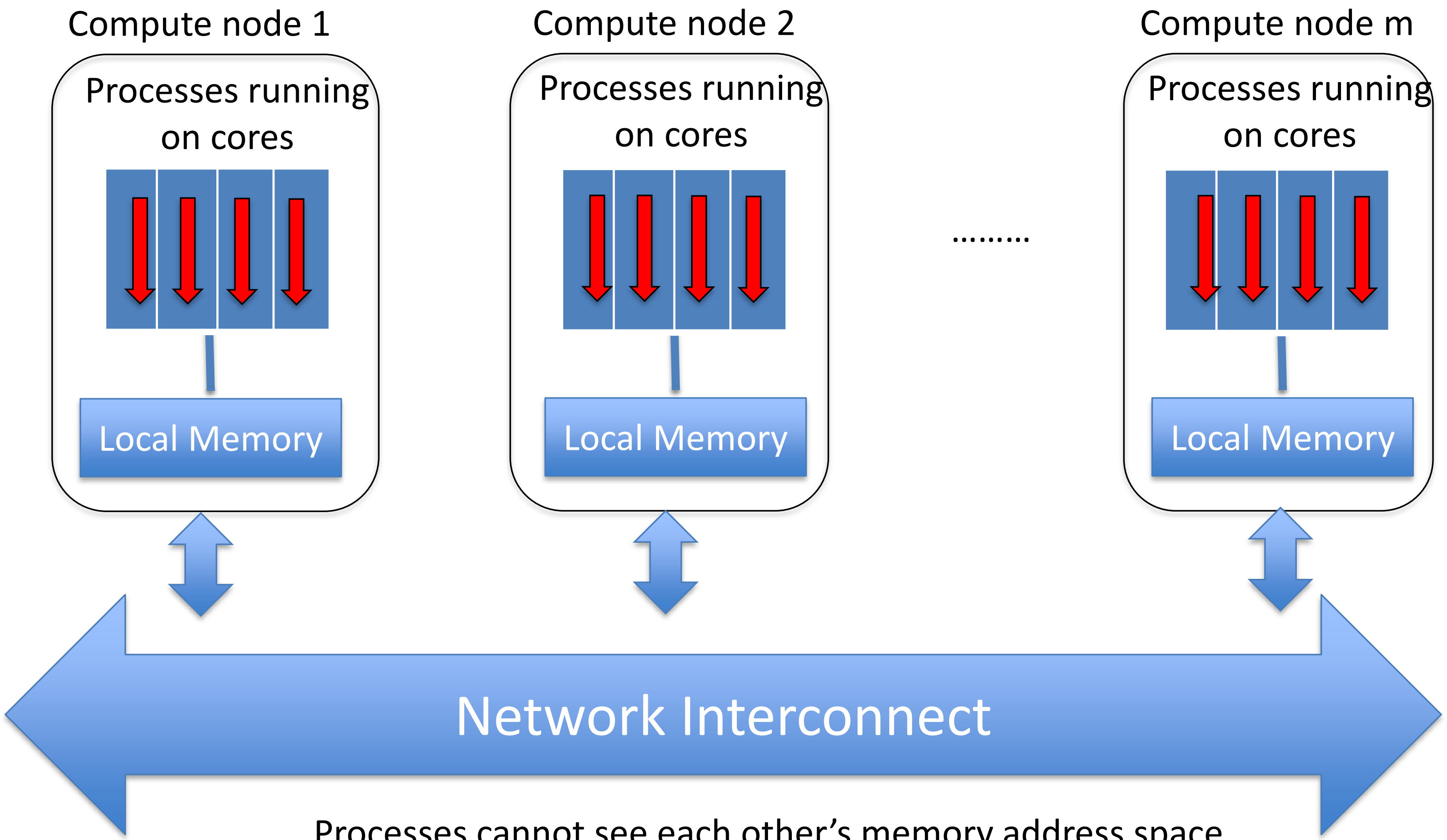
- Threads are contained within processes
  - One process => multiple threads
- All threads of a process share the same address space (in memory).
- Threads have the capability to run concurrently (executing different instructions and accessing different pieces of data at the same time)
- But if the resource is occupied by another thread, they form a queue and wait.
  - For maximum throughput, it is ideal to map each thread to a unique/distinct core



A process and two threads



Distributed Memory Architecture



Processes cannot see each other’s memory address space.
They have to send inter-process messages (using MPI).

Distributed Memory System

- Clusters (most popular)
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.
- Nodes of a cluster are individual computers joined by a communication network.

Kamiak provides an Infiniband interconnect between all compute nodes

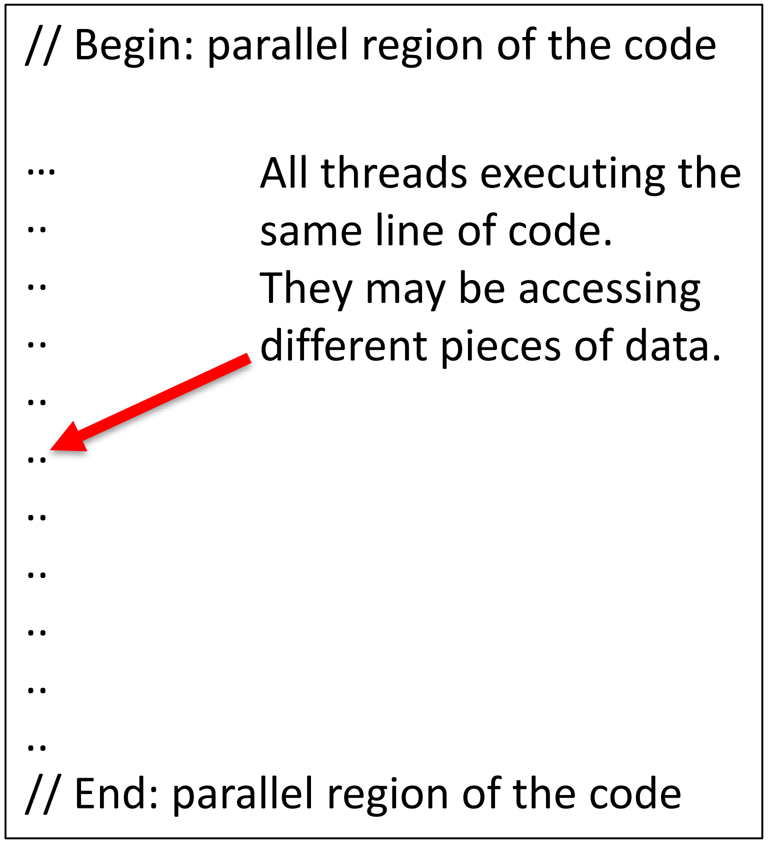
a.k.a. hybrid systems

Single Program Models: SIMD vs. MIMD

- **SP**: Single Program
  - Your parallel program is a single program that you execute on all threads (or processes)
- **SI**: Single Instruction
  - Each thread should be executing the same line of code at any given clock cycle.
- **MI**: Multiple Instruction
  - Each thread (or process) could be independently running a different line of your code (instruction) concurrently
- **MD**: Multiple Data
  - Each thread (or process) could be operating/accessing a different piece of the data from the memory concurrently

Single Program Models: SIMD vs. MIMD

SIMD

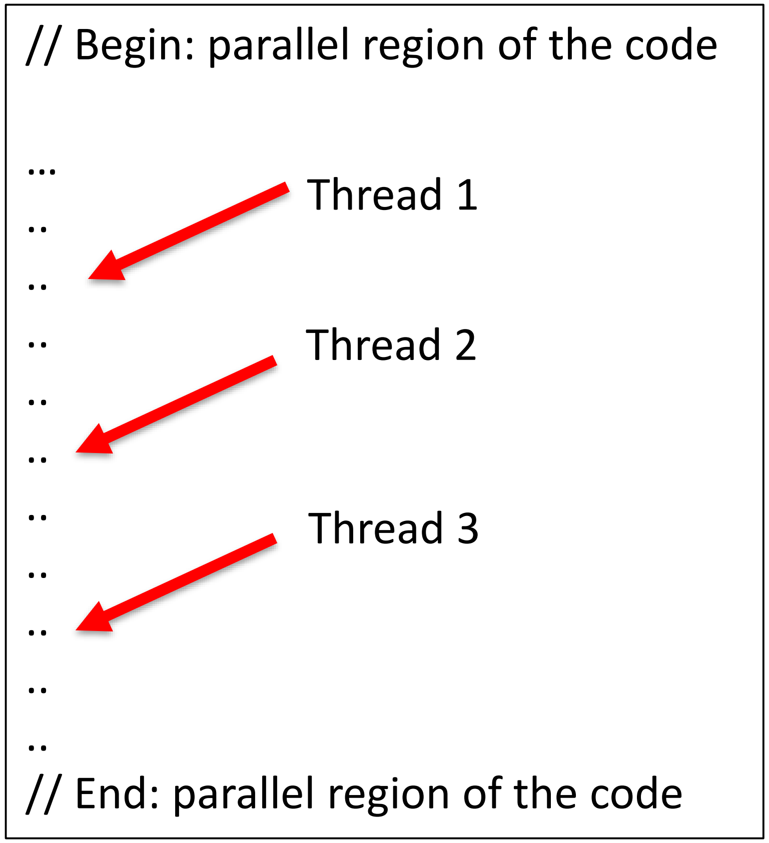


```
// Begin: parallel region of the code

…
..
..
..
..
..
..
..
..
..
..
// End: parallel region of the code
```

All threads executing the same line of code.
They may be accessing different pieces of data.

MIMD



```
// Begin: parallel region of the code

…
..
..
..
..
..
..
..
..
..
..
// End: parallel region of the code
```

Foster’s methodology

1. Partitioning: divide the computation to be performed and the data operated on by the computation into small tasks.

   The focus here should be on identifying tasks that can be executed in parallel.

Foster's methodology

2. Communication: determine what communication needs to be carried out among the tasks identified in the previous step.

Foster’s methodology

3. Agglomeration or aggregation: combine tasks and communications identified in the first step into larger tasks.

   For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

Foster’s methodology

4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

   This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

Example from sum.c

- Open sum.c
- What can be parallelized here?

OPENMP FOR SHARED MEMORY MULTITHREADED PROGRAMMING

Roadmap

- Writing programs that use OpenMP.
- Using OpenMP to parallelize many serial for loops with only small changes to the source code.
- Task parallelism.
- Explicit thread synchronization.
- Standard problems in shared-memory programming.

Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren’t part of the basic C specification.
- Compilers that don’t support the pragmas ignore them.

#pragma

OpenMp pragmas

- # pragma omp parallel

- # include omp.h

  - Most basic parallel directive.
  - The number of threads that run
    the following structured block of code
    is determined by the run-time system.

clause

- Text that modifies a directive.
- The num_threads clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp parallel num_threads ( thread_count )
```

Some terminology

- In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a team, the original thread is called the master, and the additional threads are called worker.

In case the compiler doesn’t support OpenMP

```
# include <omp.h>
```

$\searrow$

```
#ifdef _OPENMP
# include <omp.h>
#endif
```

In case the compiler doesn’t support OpenMP

```
# ifdef _OPENMP
   int my_rank = omp_get_thread_num ( );
   int thread_count = omp_get_num_threads ( );
# e l s e
   int my_rank = 0;
   int thread_count = 1;
# endif
```

Serial version of “hello world”

```
#include <stdio.h>

int main()
{
  printf("Hello world\n");
  return 0;
}
```

Compile it: gcc hello.c

How do we invoke omp in this case?

After invoking omp

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Compile it: gcc –fopenmp hello.c

Parallel Code Template (OpenMP)

```
#include <omp.h>

main(…) {
… // let p be the user-specified #threads

omp_set_num_threads(p);

#pragma omp parallel for
{
…. // openmp parallel region where p threads are
active and running concurrently
}
```

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.

- A variable that can only be accessed by a single thread has private scope.

- The default scope for variables declared before a parallel block is shared.

Loop.c

- Lets go over our first code – loop-serial.c

- Now edit to employ omp, save as loop-parallel.c

Performance

Taking Timings

- What is time?
- Start to finish?
- A program segment of interest?
- CPU time?
- Wall clock time?

Taking Timings

theoretical function

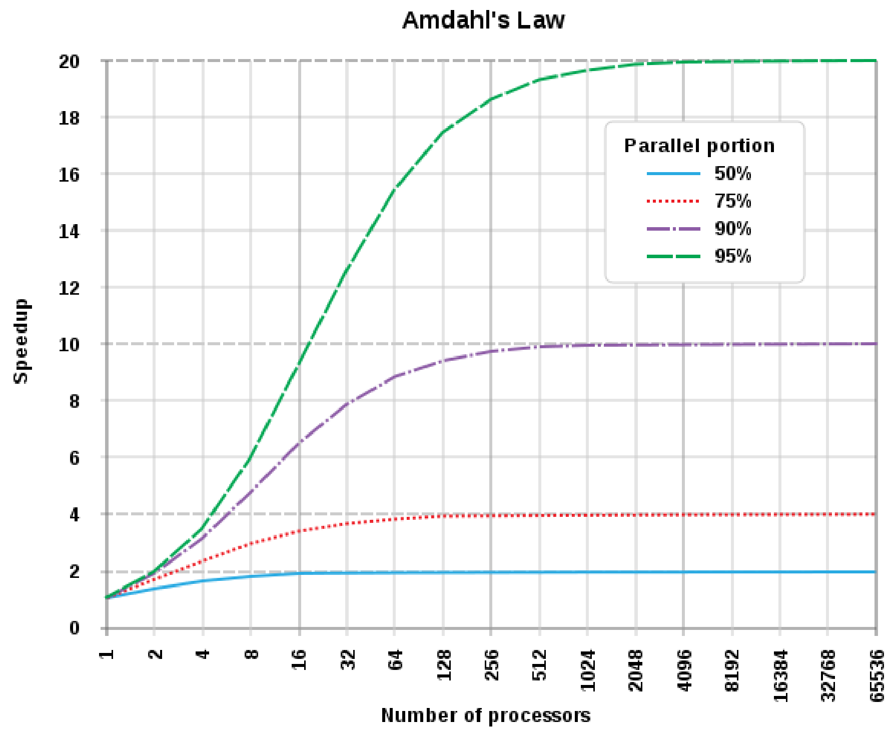
```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

MPI_Wtime

omp_get_wtime

Speedup

- Number of threads = p
- Serial run-time = T_{serial}
- Parallel run-time = $T_{parallel}$



$$T_{parallel} = T_{serial} / p$$

$$S = \frac{T_{serial}}{T_{parallel}}$$

Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.

Studying Scalability

Table records the parallel runtime (in seconds) for varying values of n and p.

Input size (n)	Number of threads (p)				
	1	2	4	8	16
1,000					
2,000					
4,000					
8,000					
16,000					

It is conventional to test scalability in powers of two (or by doubling n and p).

Studying Scalability

Table records the parallel runtime (in seconds) for varying values of n and p.

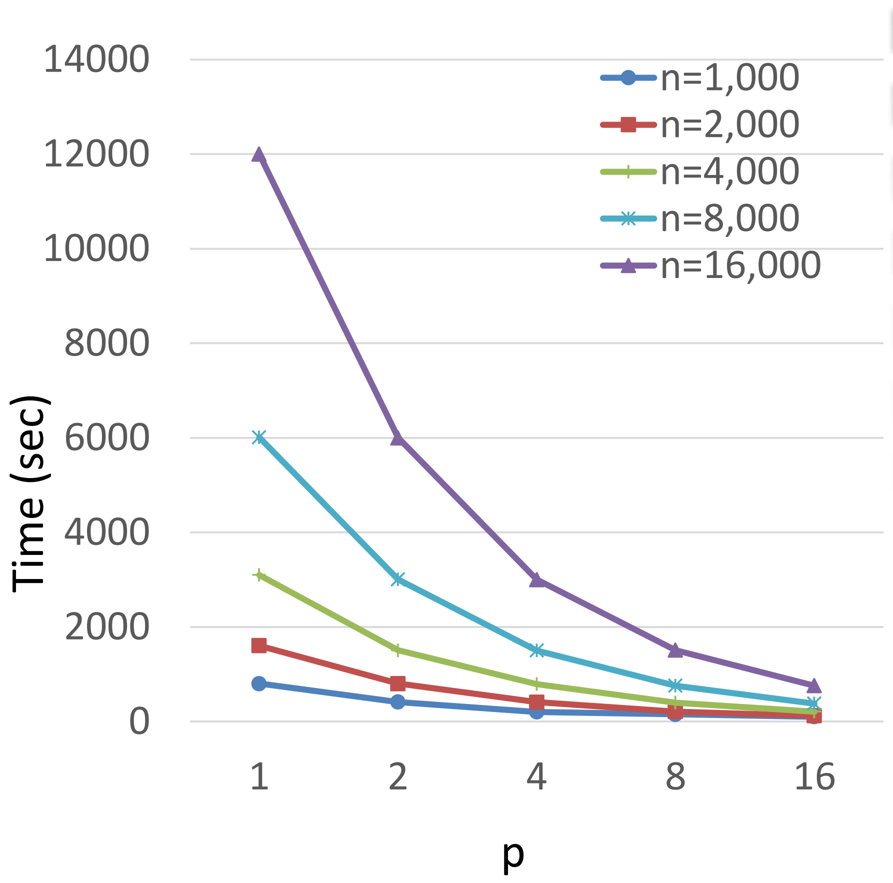
Input size (n)	Number of threads (p)				
	1	**2**	**4**	**8**	**16**
1,000	800	410	201	150	100
2,000	1,601	802	409	210	120
4,000	3,100	1,504	789	399	208
8,000	6,010	3,005	1,500	758	376
16,000	12,000	6,000	3,001	1,509	758

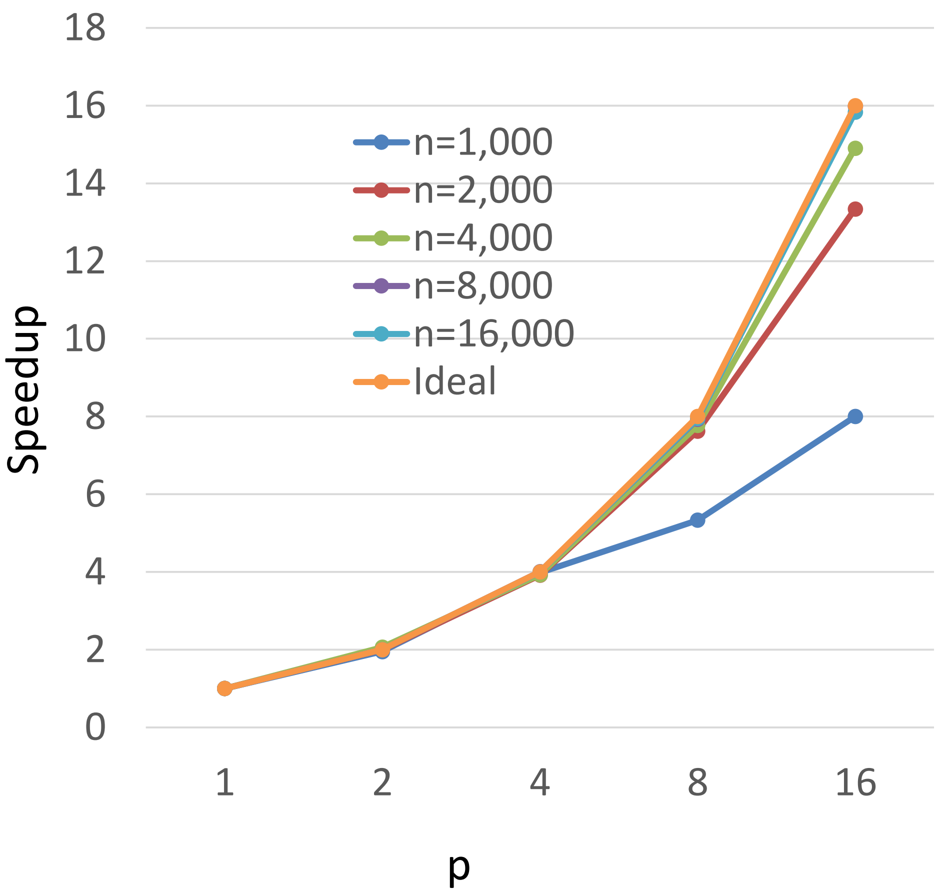
Strong scaling behavior

Weak scaling behavior

It is conventional to test scalability in powers of two (or by doubling n and p).

Studying Scalability





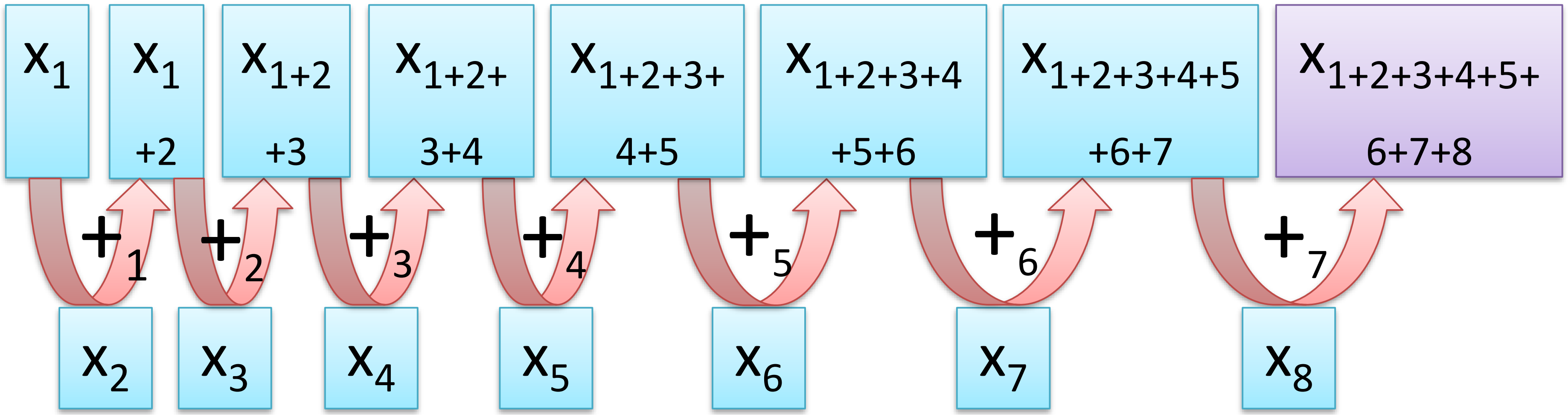
Timings with loop-parallel.c (your code) or loop.c (our code)

You try!

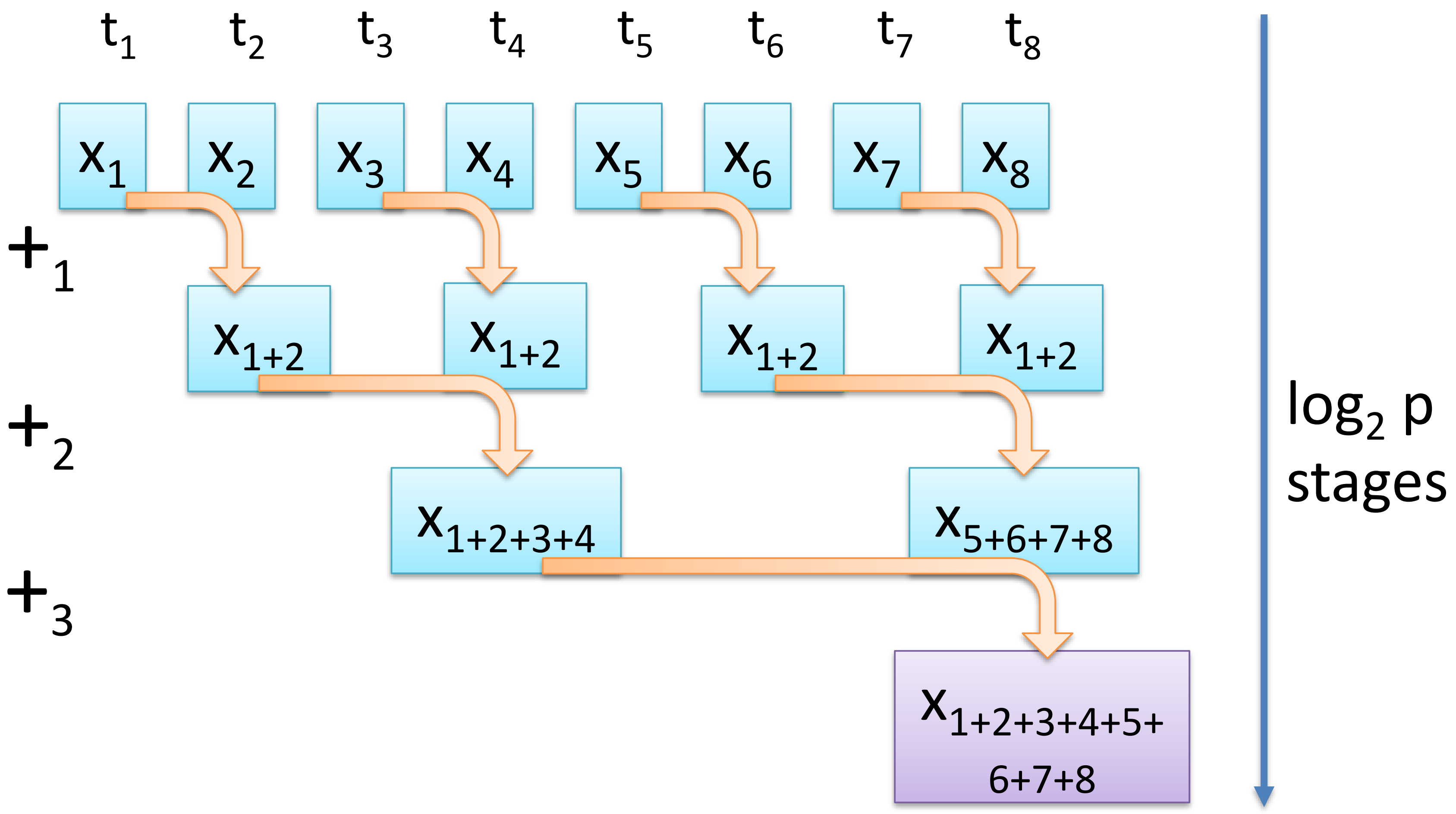
< 10 minutes

Serial vs. Parallel Reduction

Serial Process (1 thread, 7 operations)



Parallel Process (8 threads, 3 operations)



t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
$+_1$
x_{1+2} x_{1+2} x_{1+2} x_{1+2}
$+_2$
$x_{1+2+3+4}$ $x_{5+6+7+8}$
$+_3$
$x_{1+2+3+4+5+6+7+8}$
$\log_2 p$ stages

Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

Computing a sum

- **Open sumcompute-serial.c**
- **Lets go over it**

< 5 minutes

Mutual exclusion

```
# pragma omp critical
  global_result += my_result ;
```

only one thread can execute
the following structured block at a time

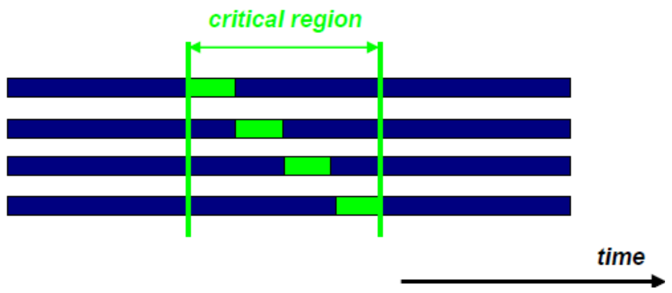
Example

- Open **sync-unsafe.c**

Synchronization

- Synchronization imposes order constraints and is used to protect access to shared data
- Types of synchronization:
  - *critical*
  - *atomic*
  - *locks*
  - *others (barrier, ordered, flush)*
- We will work on an exercise involving *critical, atomic, and locks*

Critical



```
#pragma omp parallel for schedule(static) shared(a)
for(i = 0; i < n; i++)
{
        #pragma omp critical
        {
                a = a+1;
        }
}
```

Threads wait here: only one thread at a time does the operation: “a = a+1”. So this is a piece of sequential code inside the for loop.

Atomic

- Atomic provides mutual exclusion but only applies to the load/update of a memory location
- It is applied only to the (single) assignment statement that immediately follows it
- Atomic construct may only be used together with an expression statement with one of operations: +, *, -, /, &, ^, |, <<, >>
- Atomic construct does not prevent multiple threads from executing the function() at the same time (see the example below)

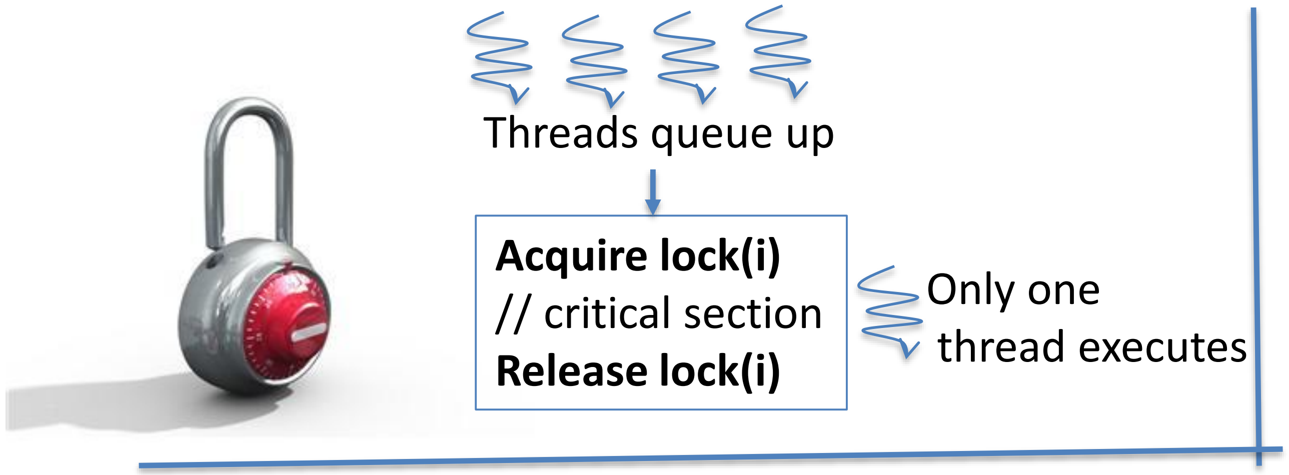
Code example:

```
int ic, i, n;
ic = 0;
#pragma omp parallel shared(n,ic) private(i)
   for (i=0; i++; i<n)
      {
          #pragma omp atomic
             ic = ic + function(c);
       }
```

Atomic only protects the update of ic

Locks

- A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

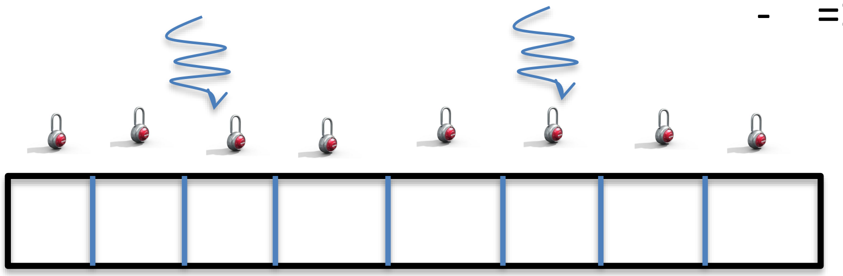


```
Acquire lock(i)
// critical section
Release lock(i)
```

<u>Difference from critical section:</u>

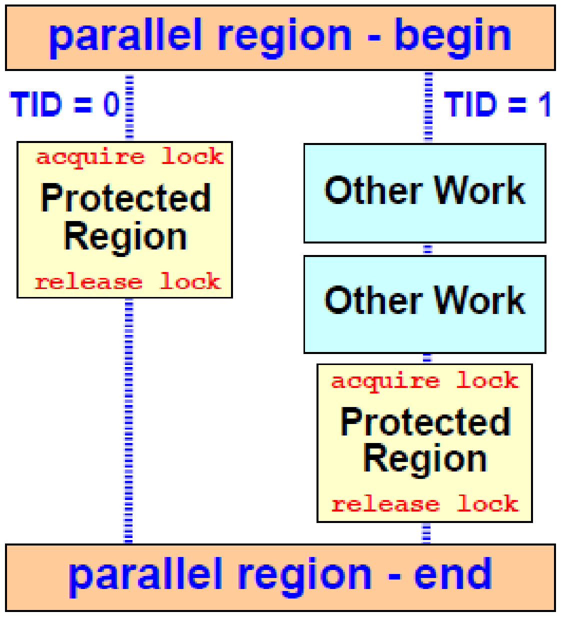
- You can have multiple locks
- A thread can try for any specific lock

- => we can use this to acquire data-level locks



e.g., two threads can access different array indices without waiting.

Illustration of Locking Operation



- The protected region contains the update of a shared variable
- One thread acquires the lock and performs the update
- Meanwhile, other threads perform some other work
- When the lock is released again, the other threads perform the update

A Locks Code Example

```
long long int a=0;
long long int i;

omp_lock_t my_lock;
// init lock
omp_init_lock(&my_lock);
#pragma omp parallel for
for(i = 0; i < n; i++)
{
      omp_set_lock(&my_lock);
      a+=1;
      omp_unset_lock(&my_lock);
}
omp_destroy_lock(&my_lock);
```

1. Define lock variable
2. Initialize lock
3. Set lock
4. Unset lock
5. Destroy lock

Compiling and running sync.c:

```
gcc −g −Wall −fopenmp −o sync sync.c
./sync #of-iteration #of-threads
```

Some Caveats

1. You shouldn't mix the different types of mutual exclusion for a single critical section.
2. There is no guarantee of fairness in mutual exclusion constructs.
3. It can be dangerous to "nest" mutual exclusion constructs.

Loop.c example

- Default schedule:

```
#pragma omp parallel for schedule(static)
private(a)//creates N threads to run the
next enclosed block
   for(i = 0; i < loops; i++)
   {
      a = 6+7*8;
   }
```

The Runtime Schedule Type

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.
- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.

schedule (type , chunksize)

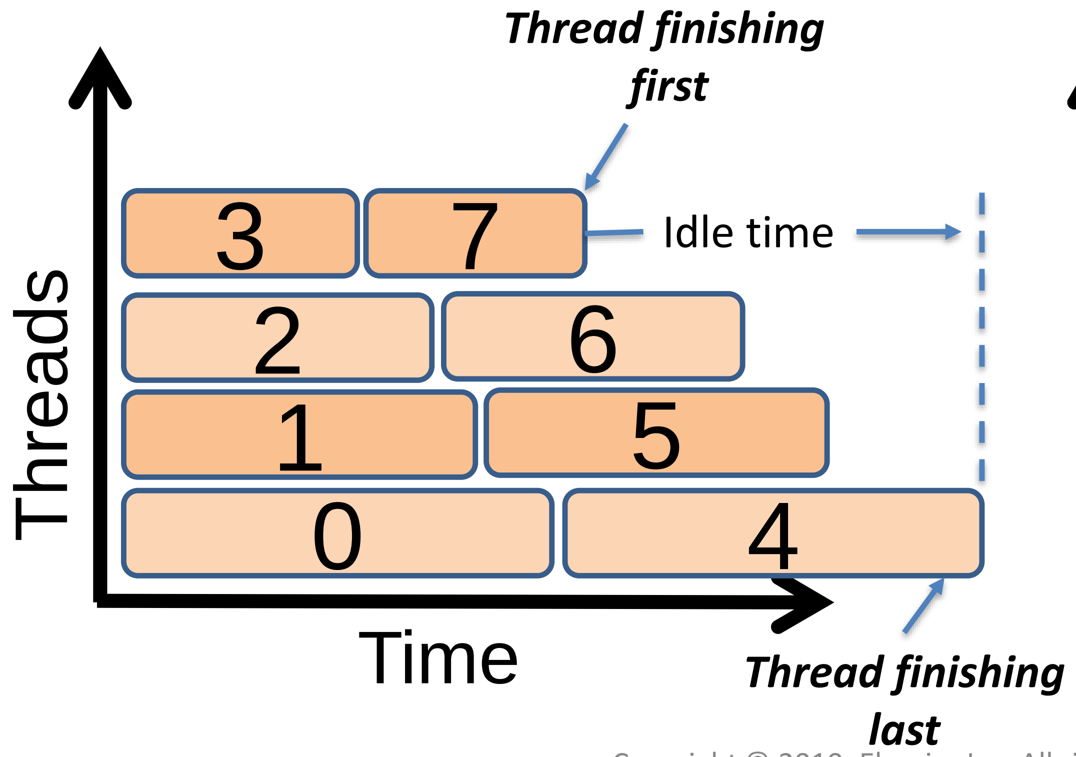
Controls how loop iterations are assigned

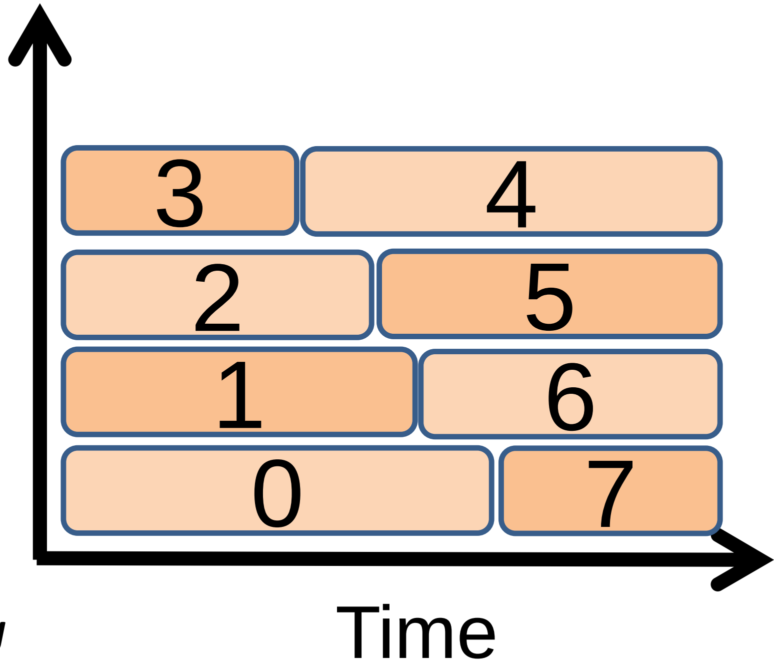
- Static: Assigned before the loop is executed.
- dynamic or guided: Assigned while the loop is executing.
- auto/ runtime: Determined by the compiler and/or the run-time system

- Consecutive iterations are broken into chunks
- Total number = chunksize
- Positive integer
- Default is 1

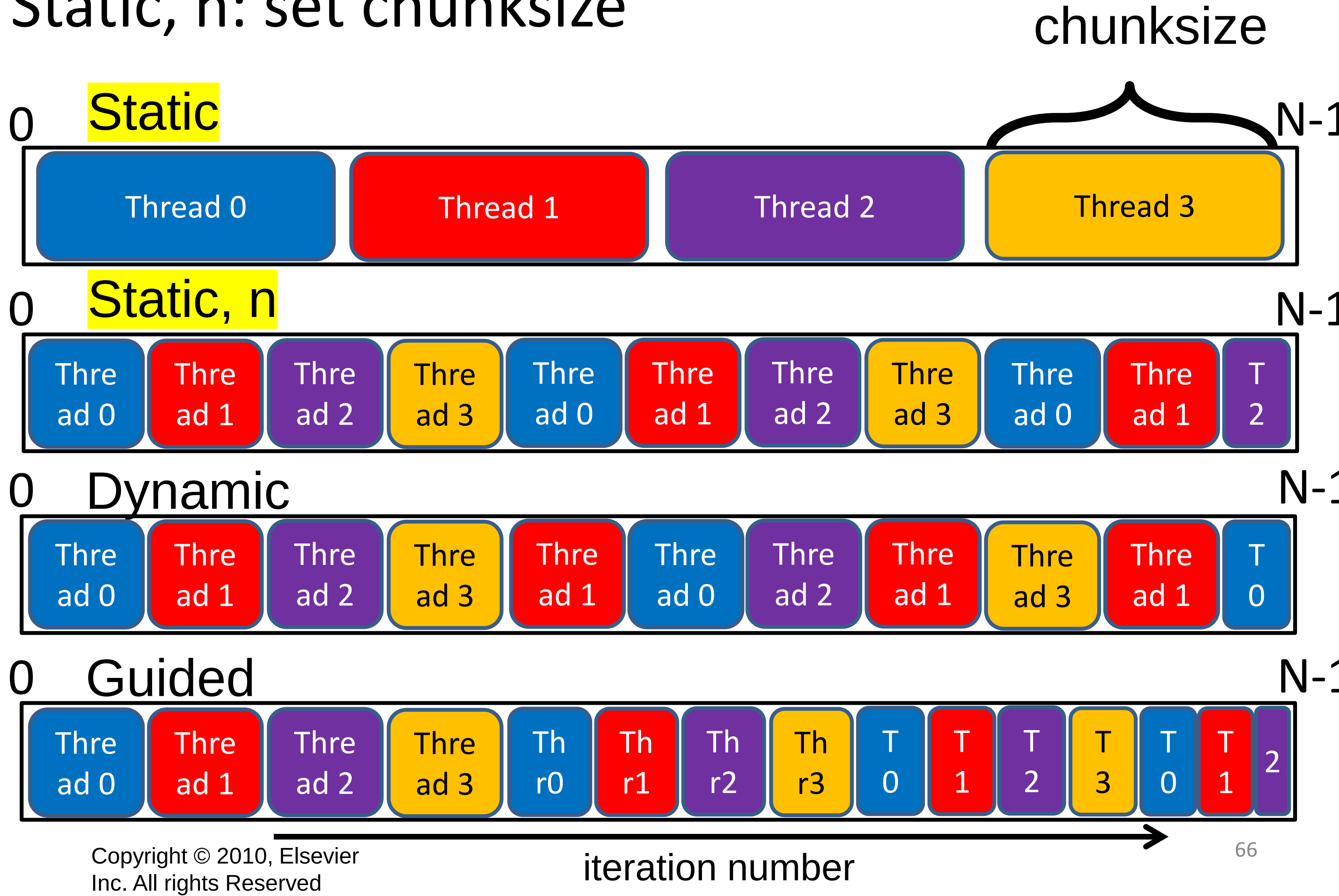
schedule types can prevent load imbalance

Static schedule vs Dynamic schedule

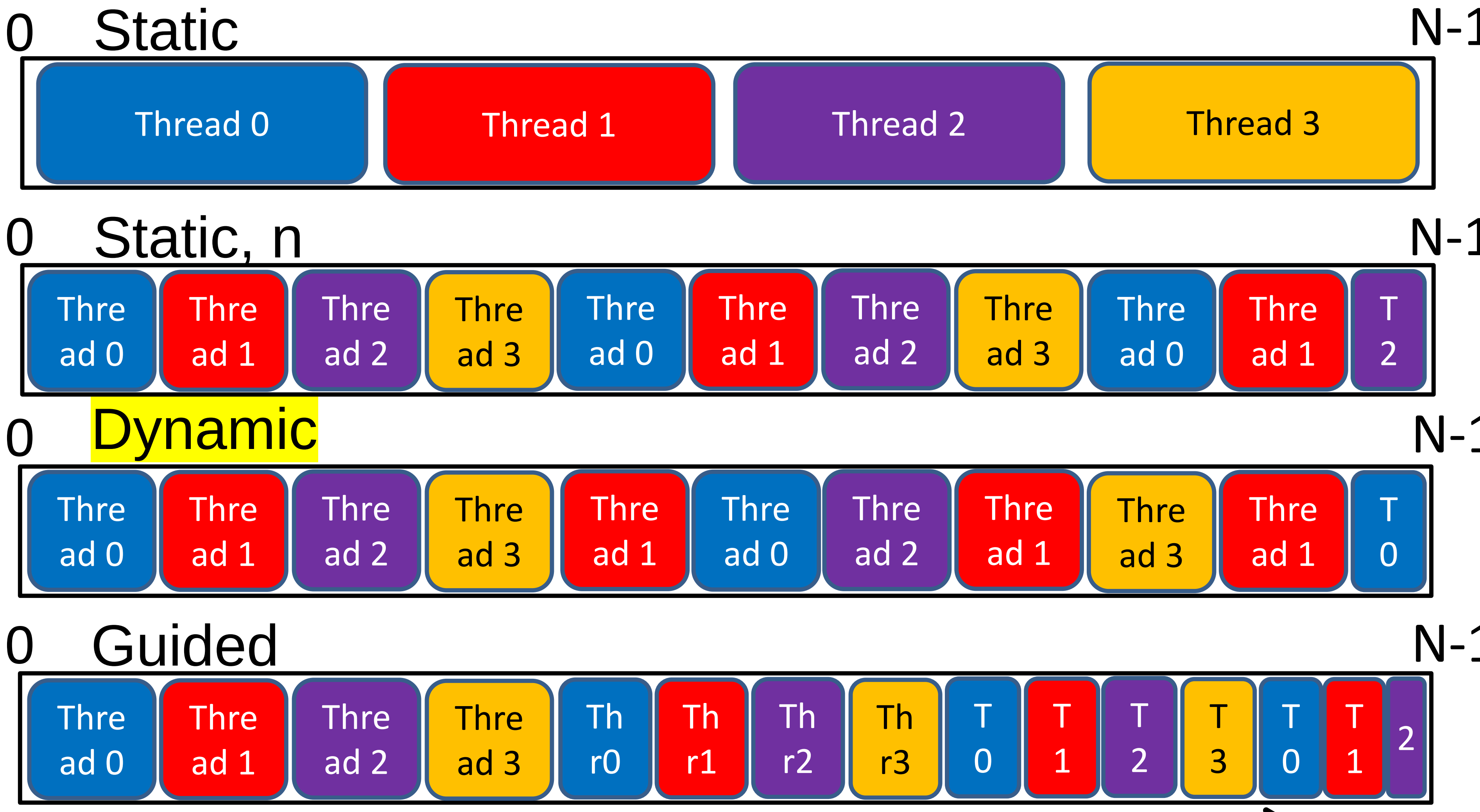




Static: default
Static, n: set chunksize

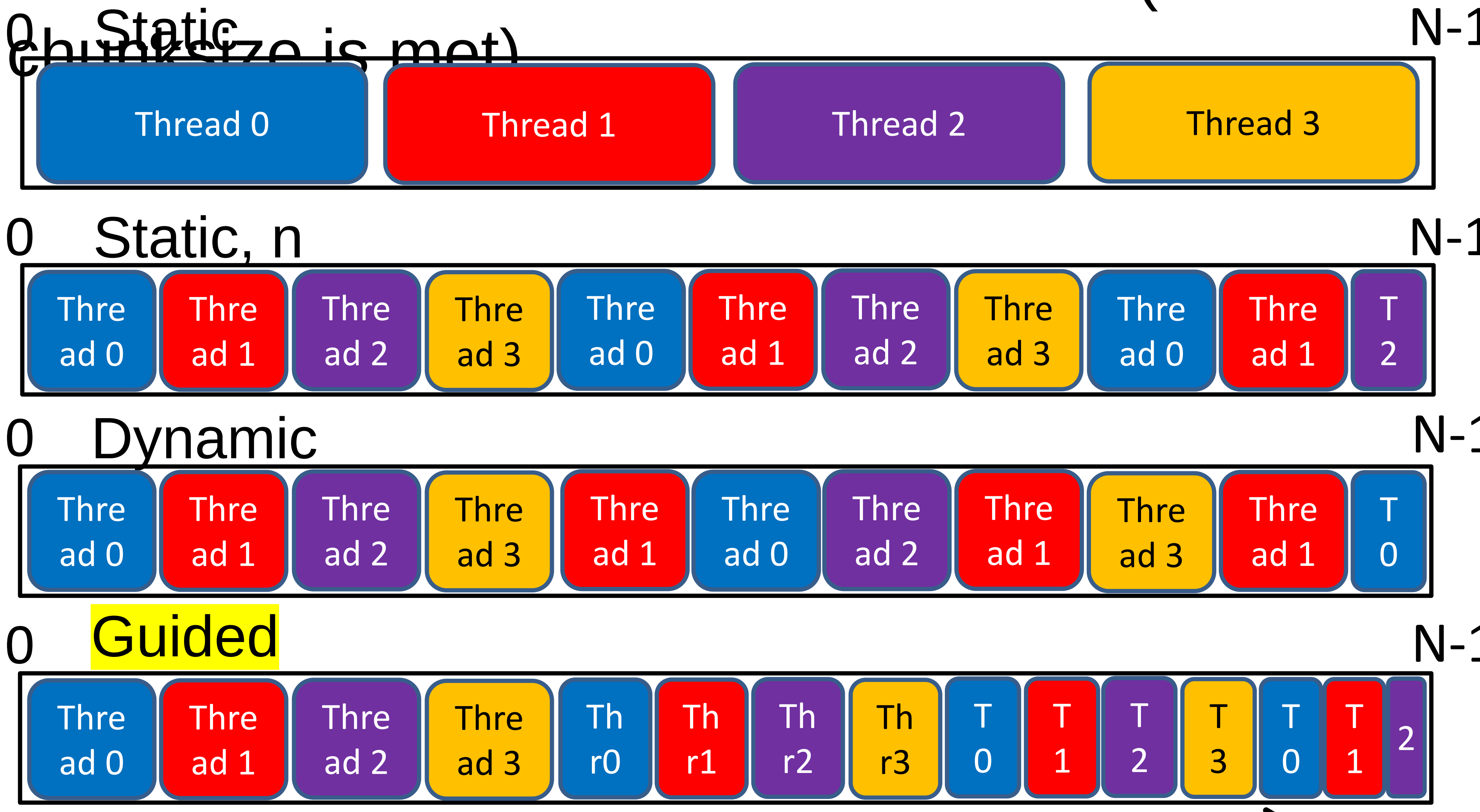


Dynamic: thread executes a chunk when done, it requests another one



iteration number

Guided: thread executes a chunk when done, it requests another one new chunks decrease in size (until chunksize is met)



multiplication.c

- Go over this code at a conceptual level – show where and how to parallelize

Matrix-vector multiplication

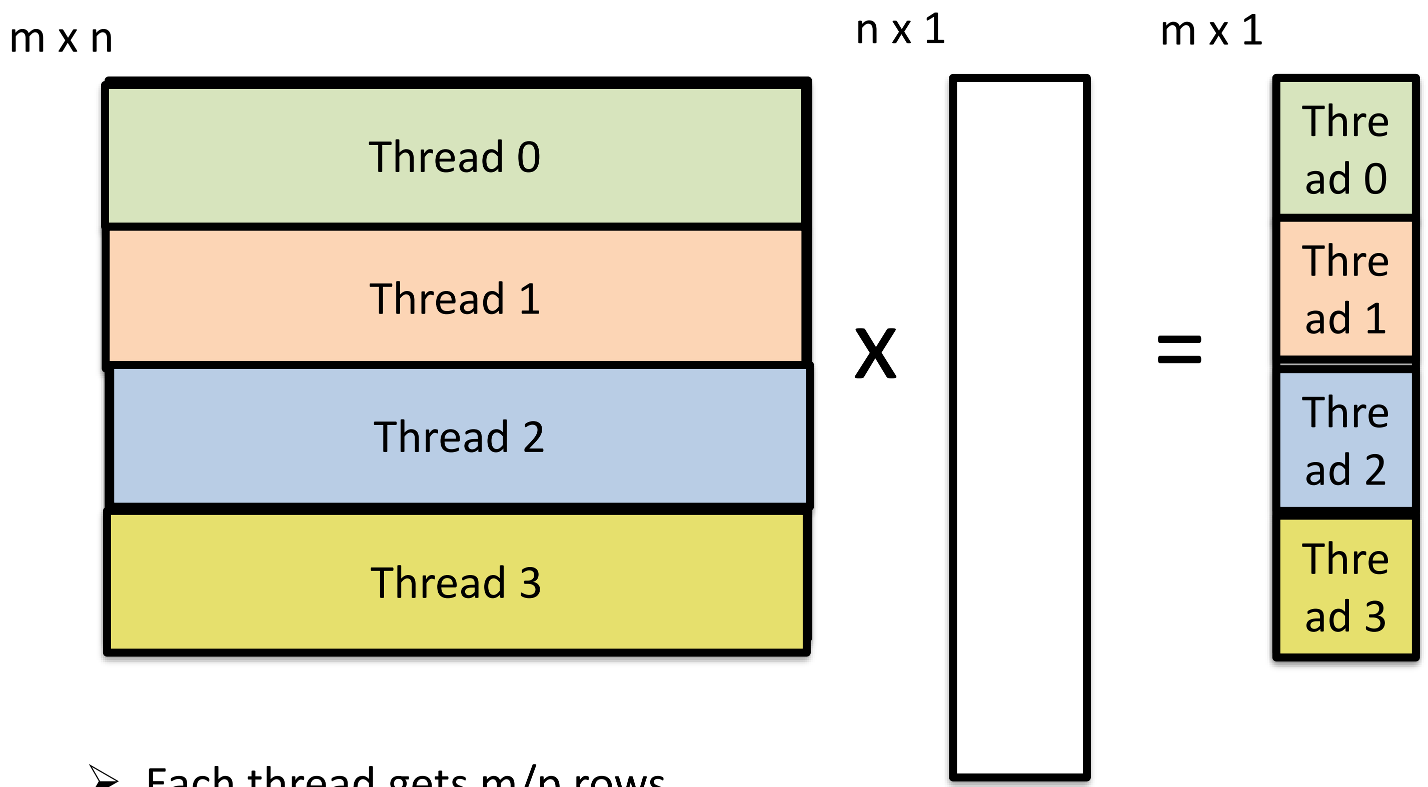
$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

$$\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}$$

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

M x V = X: Parallelization Strategies

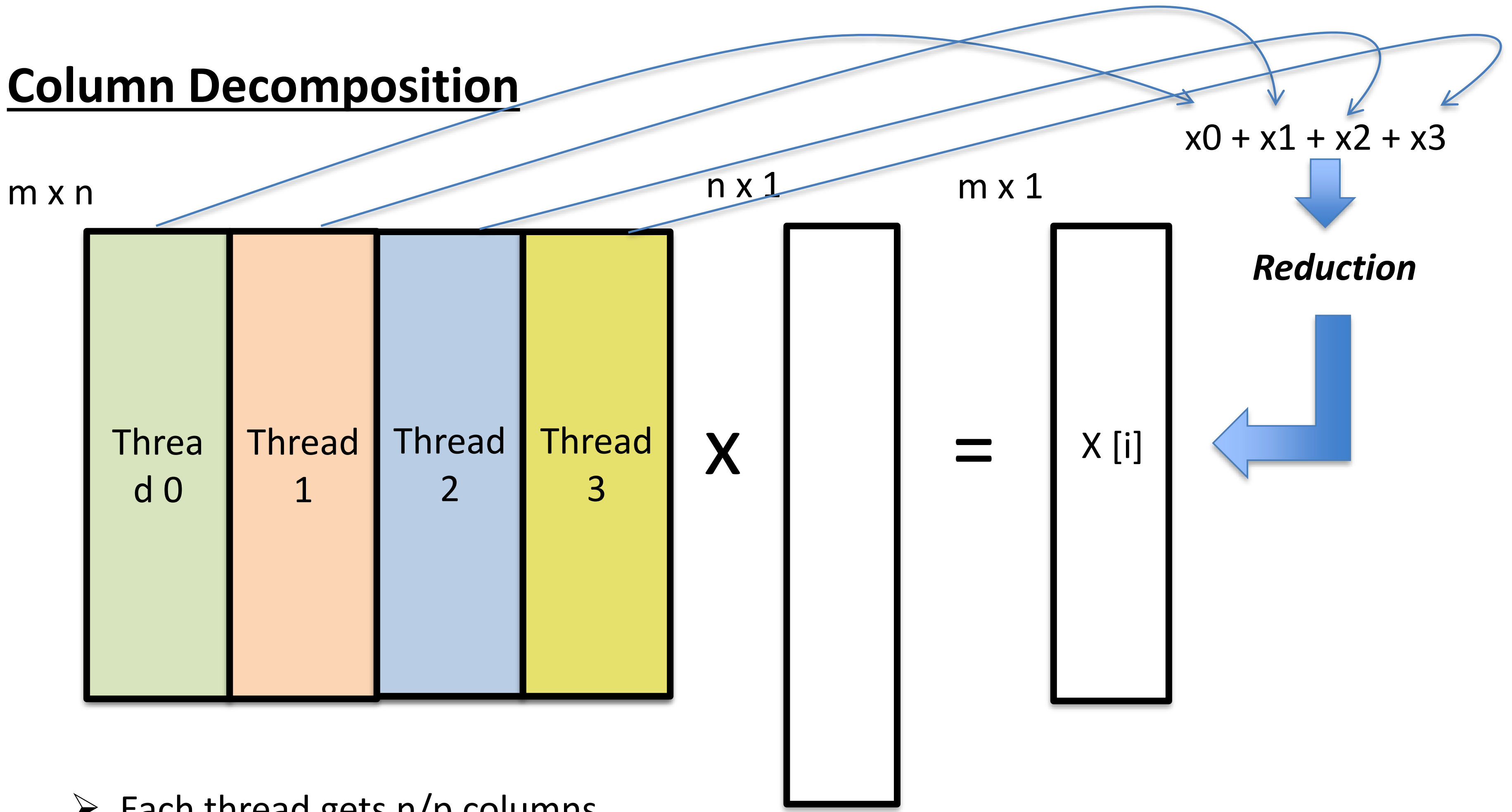
Row Decomposition



- Each thread gets m/p rows
- Time taken is proportional to: (mn)/p : per thread
- No need for any synchronization (static scheduling will do)

M x V = X: Parallelization Strategies

Column Decomposition



- Each thread gets n/p columns
- Time taken is proportional to: (mn)/p + time for reduction : per thread

Extra slides

What happened?

1. OpenMP compilers don’t check for dependences among iterations in a loop that’s being parallelized with a parallel for directive.
2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

Estimating π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
   sum += factor/(2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```

OpenMP solution #1

loop dependency

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           sum += factor/(2*k+1);
           factor = -factor;
        }
        pi_approx = 4.0*sum;
```

OpenMP solution #2

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Insures factor has private scope.

Thread-Safety

```
void Tokenize(
      char*   lines[]        /* in/out */,
      int     line_count     /* in     */,
      int     thread_count   /* in     */) {
   int my_rank, i, j;
   char *my_token;

#  pragma omp parallel num_threads(thread_count) \
      default(none) private(my_rank, i, j, my_token) \
      shared(lines, line_count)
   {
      my_rank = omp_get_thread_num();
#     pragma omp for schedule(static, 1)
      for (i = 0; i < line_count; i++) {
         printf("Thread %d > line %d = %s", my_rank, i, lines[i]);
         j = 0;
         my_token = strtok(lines[i], " \t\n");
         while ( my_token != NULL ) {
            printf("Thread %d > token %d = %s\n", my_rank, j, my_token);
            my_token = strtok(NULL, " \t\n");
            j++;
         }
      } /* for i */
   }  /* omp parallel */

}  /* Tokenize */
```

An operating system “process”

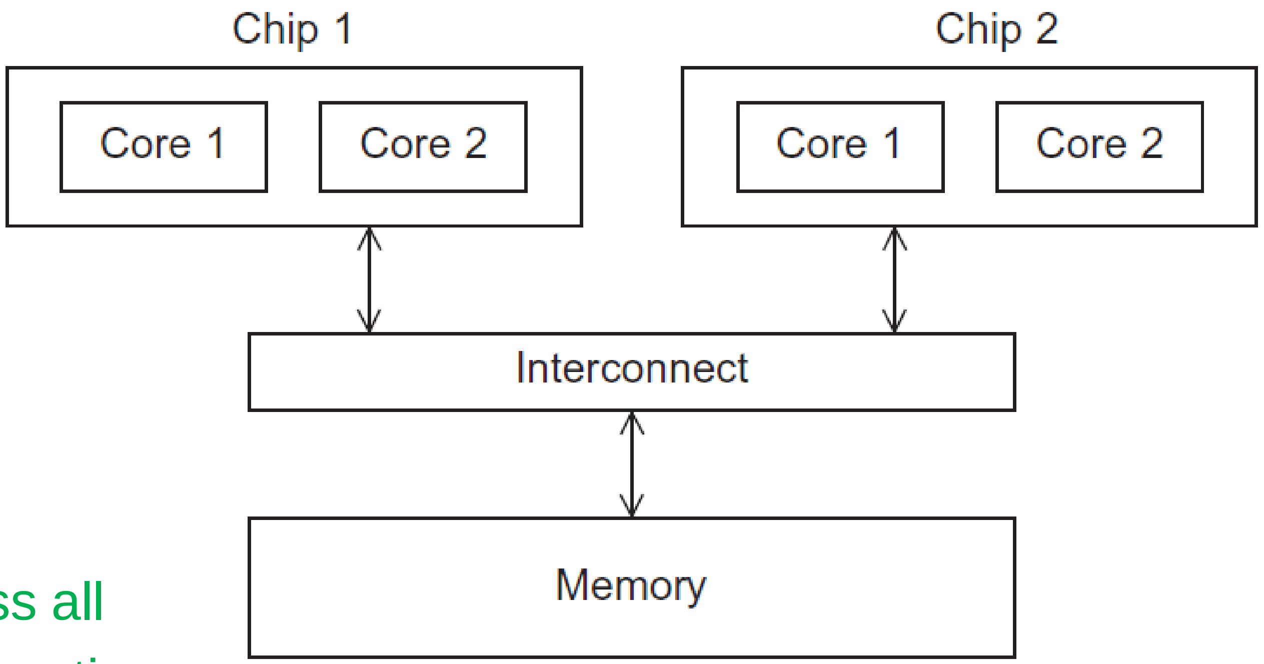
- An instance of a computer program that is being executed.
- Components of a process:
  - The executable machine language program.
  - A block of memory.
  - Descriptors of resources the OS has allocated to the process.
  - Security information.
  - Information about the state of the process.

Shared Memory System

- Each processor can access each memory location.
- The processors usually communicate implicitly by accessing shared data structures.
- Example: Multiple CPU cores on a single chip

Kamiak compute nodes have multiple CPUs each with multiple cores

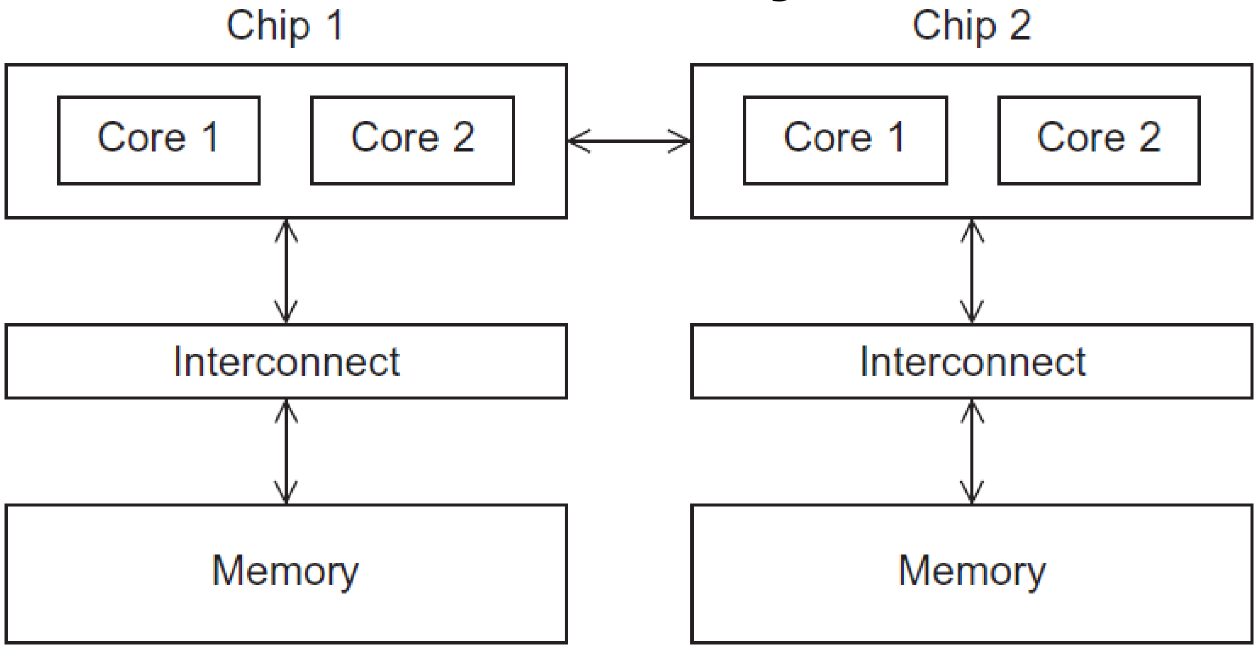
UMA multicore system



Time to access all the memory locations will be the same for all the cores.

Figure 2.5

NUMA multicore system



A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

Figure 2.6

Input and Output

- However, because of the indeterminacy of the order of output to *stdout*, in most cases only a single process/thread will be used for all output to *stdout* other than debugging output.

- Debug output should always include the rank or id of the process/thread that's generating the output.

Input and Output

- Only a single process/thread will attempt to access any single file other than *stdin*, *stdout*, or *stderr*. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.

Division of work – data parallelism

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Division of work – task parallelism

```
if (I'm the master core) {
    sum = my_x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_x to the master;
}
```

Tasks

1) Receiving
2) Addition

Distributed Memory on Kamiak

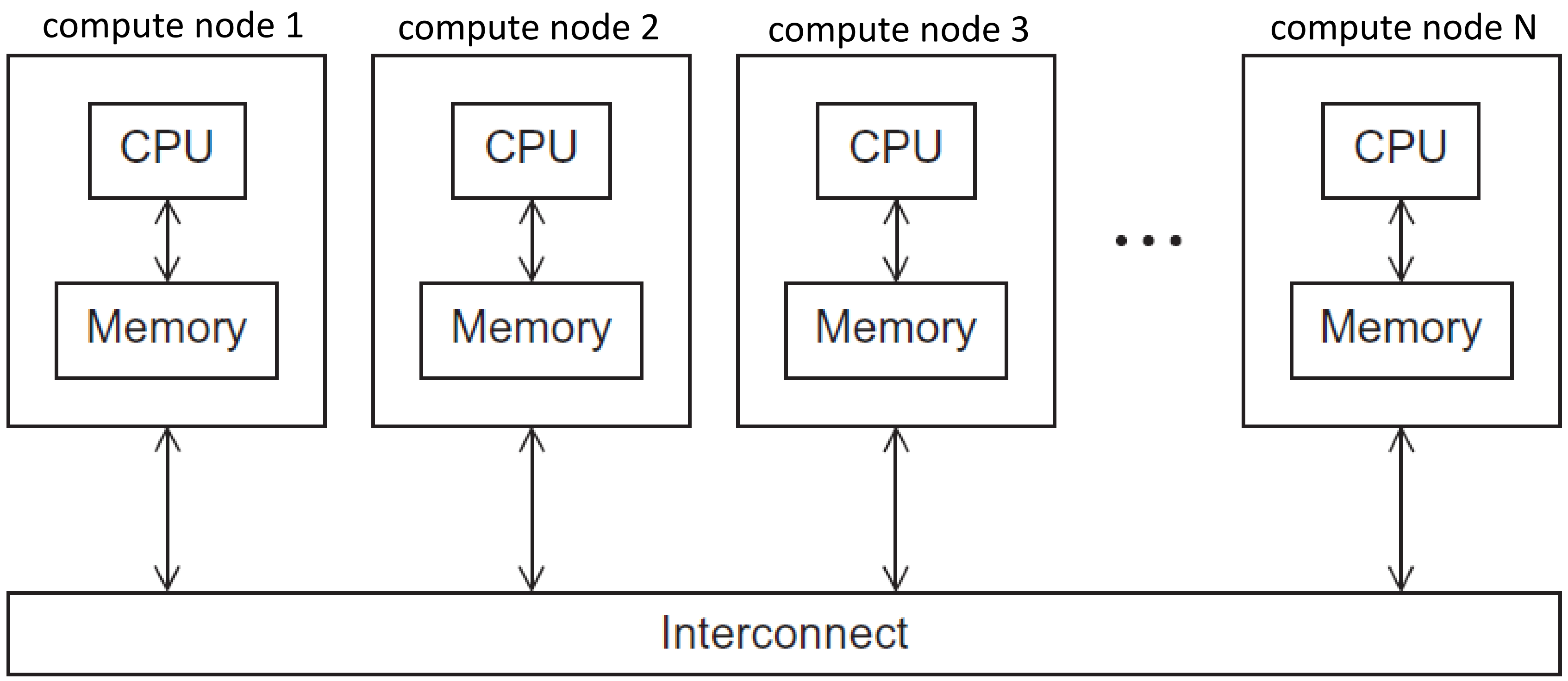


Figure 2.4

Each compute node has 1-2 CPUs each with 10-14 cores

The burden is on software

- Hardware and compilers can keep up the pace needed.
- From now on…
  - **In shared memory programs:**
    - **Start a single process and fork threads.**
    - **Threads carry out tasks.**
  - In distributed memory programs:
    - Start multiple processes.
    - Processes carry out tasks.

Writing Parallel Programs

1. Divide the work among the processes/threads
   (a) so each process/thread gets roughly the same amount of work
   (b) and communication is minimized.
2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among processes/threads.

```
double x[n], y[n];
...
for (i = 0; i < n; i++)
    x[i] += y[i];
```

I think we will eventually introduce this when doing sum. I would remove this slide from here.

Ananth can you put in a simple example here?

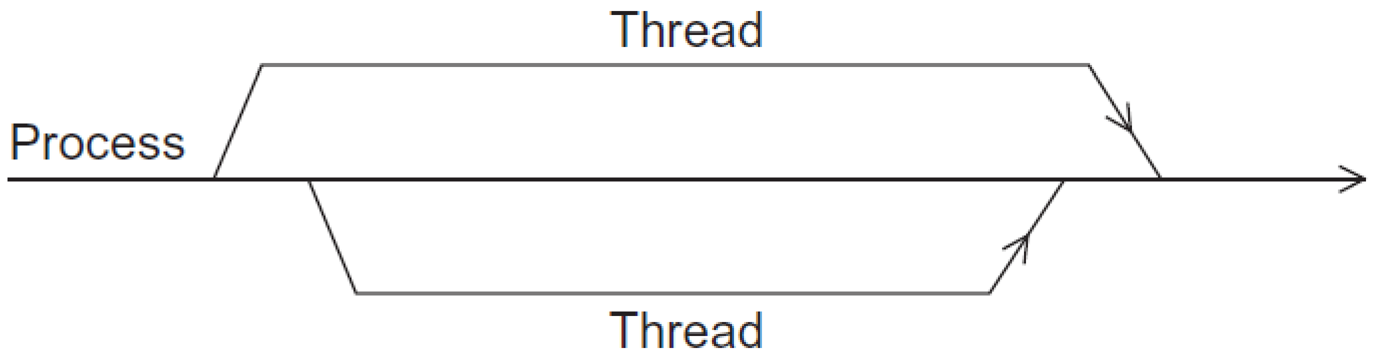
Remove Slide

OpenMP (ask ananth to tailor more for the loop.c code)

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for syste [illegible] each thread or process [illegible] lly have access to all availa [illegible] .
- System is viewed [illegible] on of cores or CPU' s, all of wh [illegible] ccess to main memory.

Remove Slide

A process forking and joining two threads



Of note…

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn’t guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we’re trying to start a lot of threads, we will almost always get the desired number of threads.

```
#include <stdio.h>
#include <stdl
#include <omp.

void Hello(voi

int main(int a
   /* Get num
   int thread_

#  pragma omp
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Have people take serial version and make into the openMP version of loop.c

-have a small example code that has the pragma and openMP calls in it

-then have them adapt the loop.c to have the openmp in it…give them 10 minutes to do while we go around the room and help….then walk them through our parallel version of loop.c

SCHEDULING LOOPS IN SYNC.C

Locks

- A lock implies a memory fence of all thread visible variables
- The lock routines are used to guarantee that only one thread accesses a variable at a time to avoid race conditions
- C/C++ lock variables must have type "omp_lock_t" or "omp_nest_lock_t" (will not discuss nested lock in this workshop)
- All lock functions require an argument that has a pointer to omp_lock_t or omp_nest_lock_t
- Simple Lock functions:
  - omp_init_lock(omp_lock_t*);
  - omp_set_lock(omp_lock_t*);
  - omp_unset_lock(omp_lock_t*);
  - omp_test_lock(omp_lock_t*);
  - omp_destroy_lock(omp_lock_t*);

How to Use Locks

1) Define the lock variables
2) Initialize the lock via a call to omp_init_lock
3) Set the lock using omp_set_lock or omp_test_lock. The latter checks whether the lock is actually available before attempting to set it. It is useful to achieve asynchronous thread execution.
4) Unset a lock after the work is done via a call to omp_unset_lock.
5) Remove the lock association via a call to omp_destroy_lock.

Matrix-vector multiplication

```
#   pragma omp parallel for num_threads(thread_count)  \
       default(none) private(i, j) shared(A, x, y, m, n)
    for (i = 0; i < m; i++) {
       y[i] = 0.0;
       for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
    }
```

Run-times and efficiencies of matrix-vector multiplication (times are in seconds)

	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275