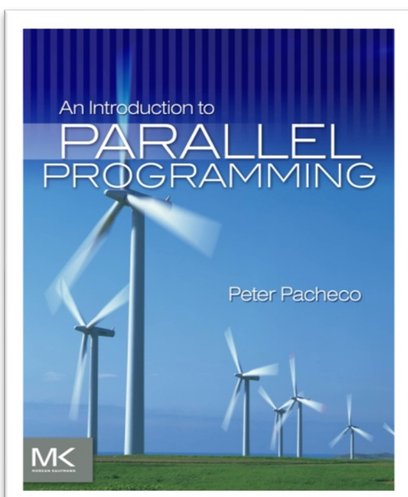
Introduction to Parallel Programming III

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/*

Taking Timings

- What is time?

Taking Timings

- What is time?
- Start to finish?

Taking Timings

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

Taking Timings

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

Taking Timings

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

user-cpu:

time spent in user code

Taking Timings

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

user-cpu:
    time spent in user code
system-cpu time:
    time spent in kernel cod

Taking Timings

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

user-cpu:
  time spent in user code
system-cpu time:
  time spent in kernel cod

- Wall clock time?

Taking Timings

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

  user-cpu:
  time spent in user code
  system-cpu time:
  time spent in kernel code

- Wall clock time?

  Actual time elapsed between the start of the process and 'now'

Taking Timings

```
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);
```

Taking Timings

```
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);
```

Taking Timings

```
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);
```

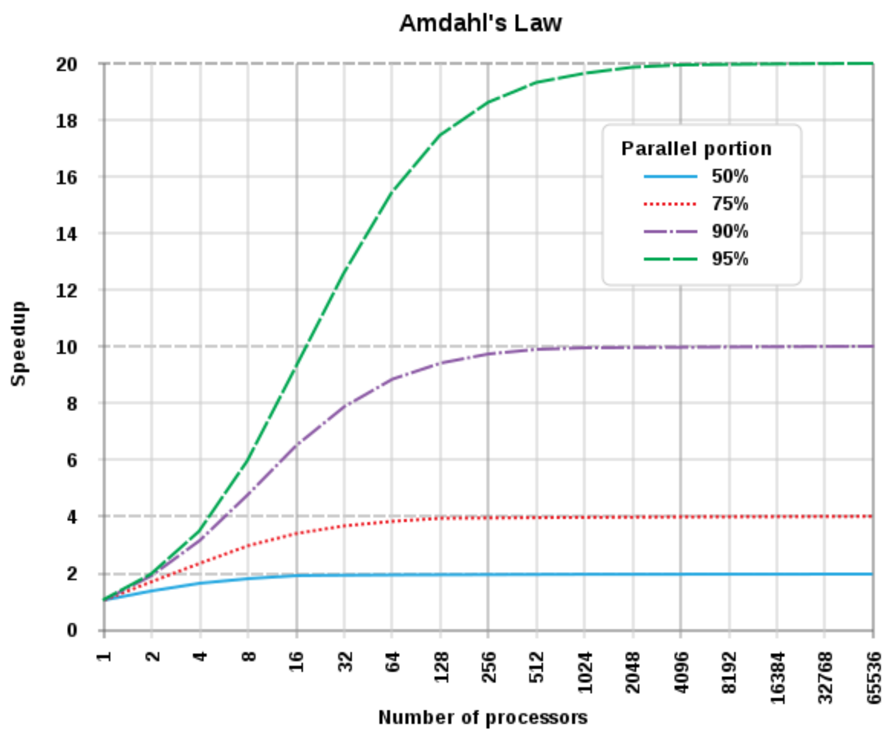
Taking Timings

```
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);
```

wall clock time rather than CPU time:
study scalability and speedup

Speedup

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



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

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.

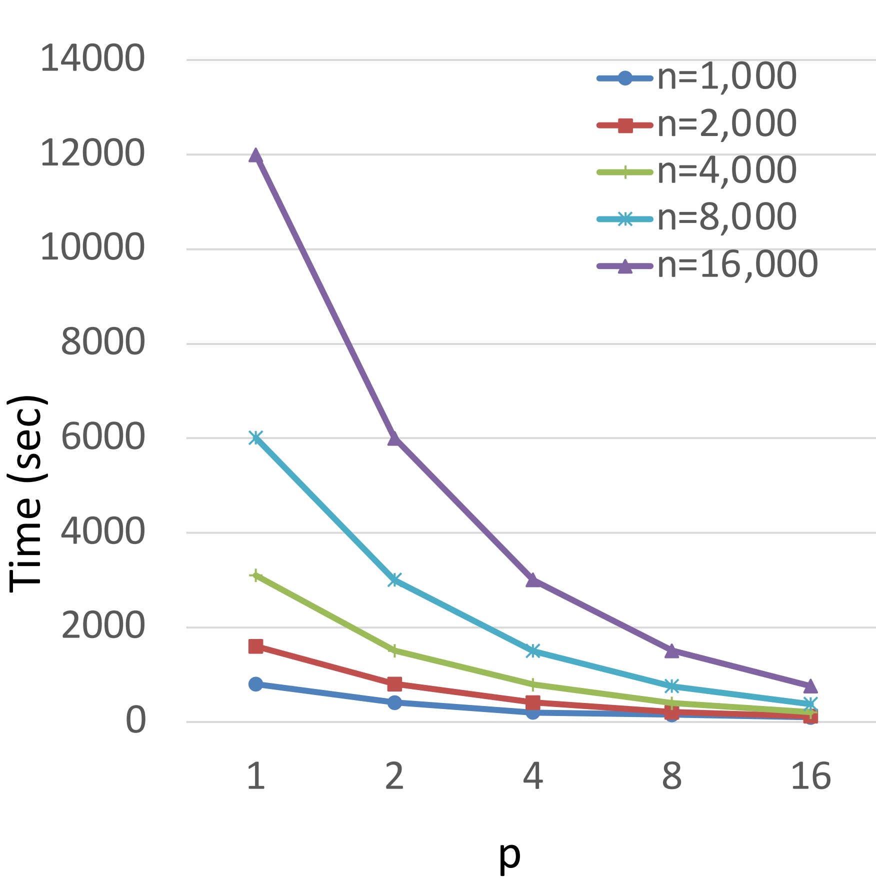
Input size (n)	Number of threads (p)				
	1	2	4	8	16
1,000	800	410	201	150	100
2,000	1,601	802	409	210	120
4,000	3,100	1,504	789	399	208
8,000	6,010	3,005	1,500	758	376
16,000	12,000	6,000	3,001	1,509	758

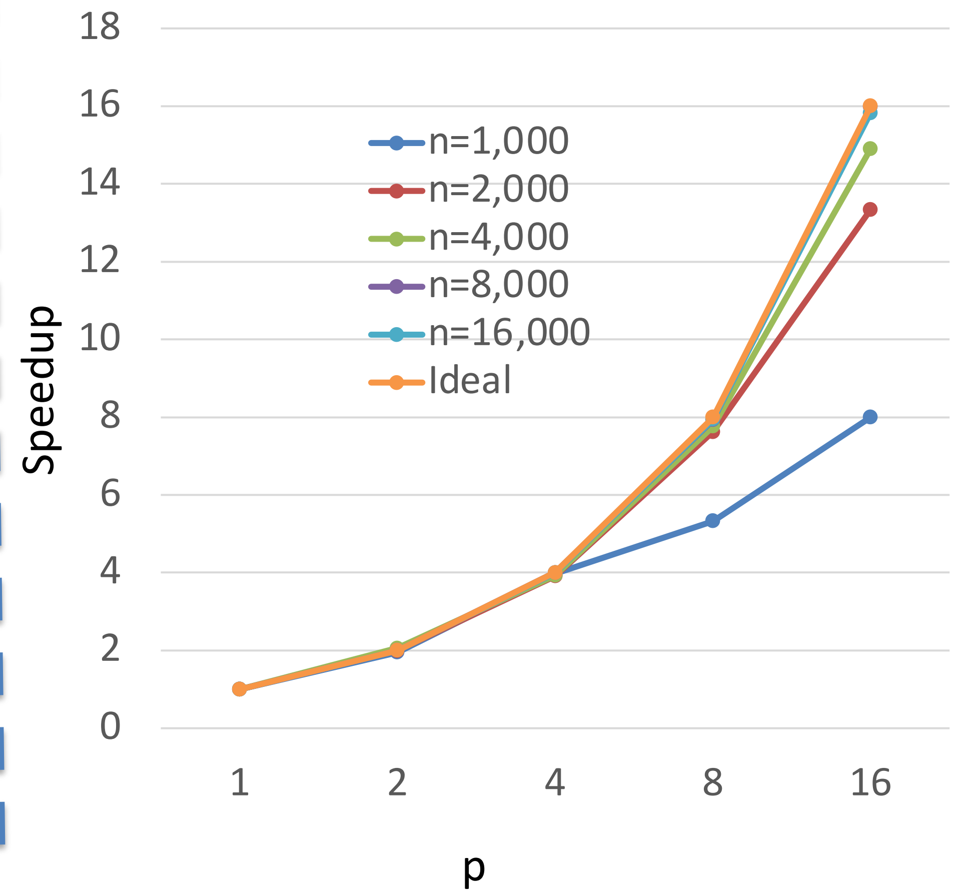
Strong scaling behavior

Weak scaling behavior

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

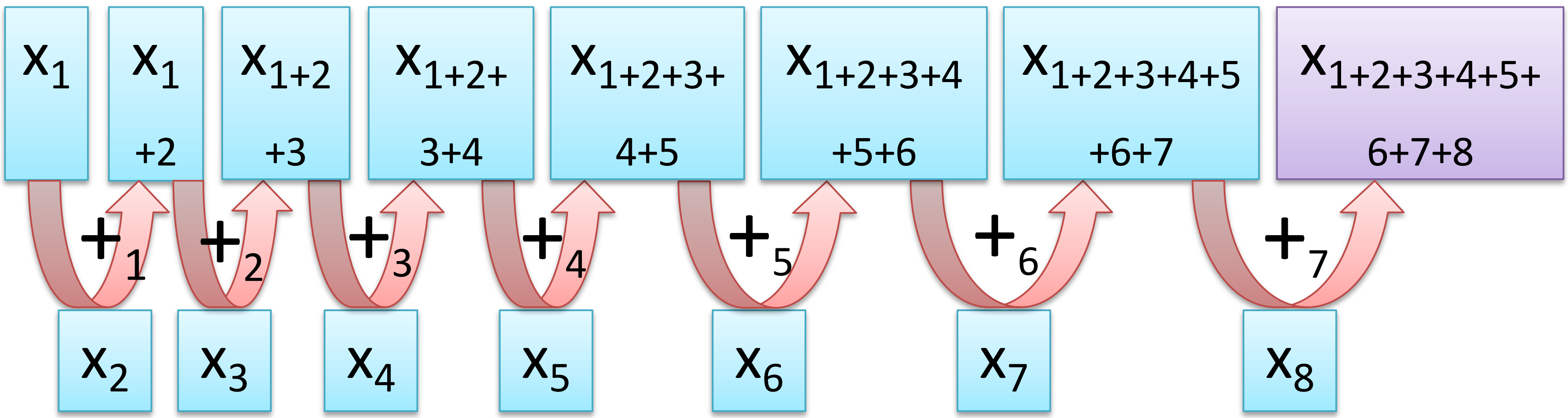
Studying Scalability



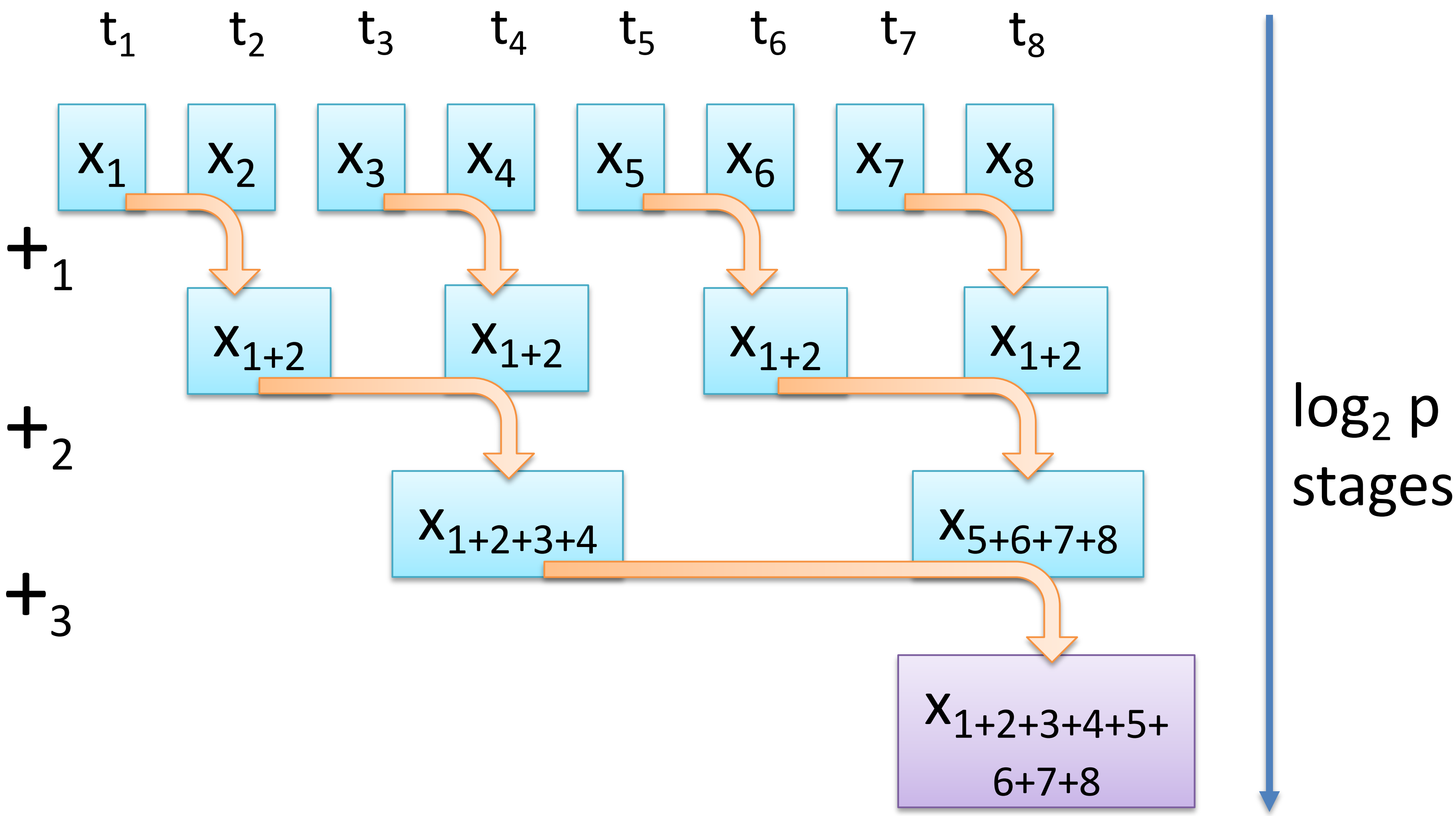


Serial vs. Parallel Reduction

Serial Process (1 thread, 7 operations)



Parallel Process (8 threads, 3 operations)



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.

Mutual exclusion

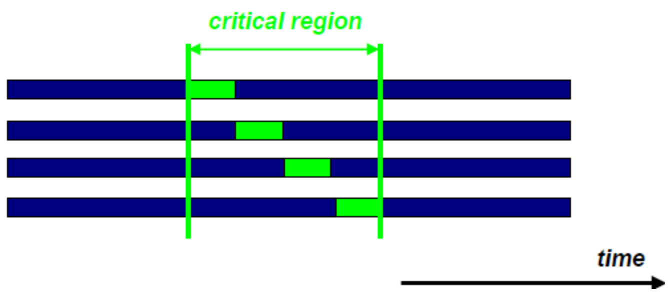
```
# pragma omp critical
  global_result += my_result ;
```

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

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

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.

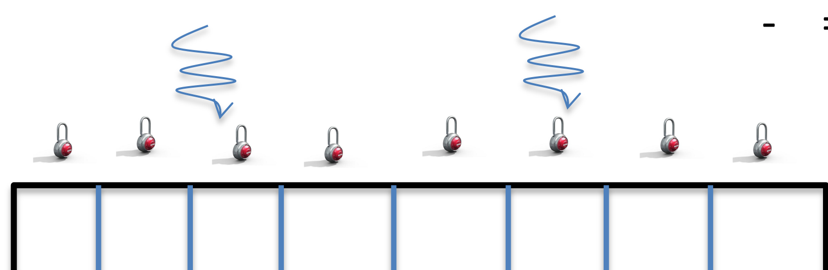
Threads queue up

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

Only one thread executes

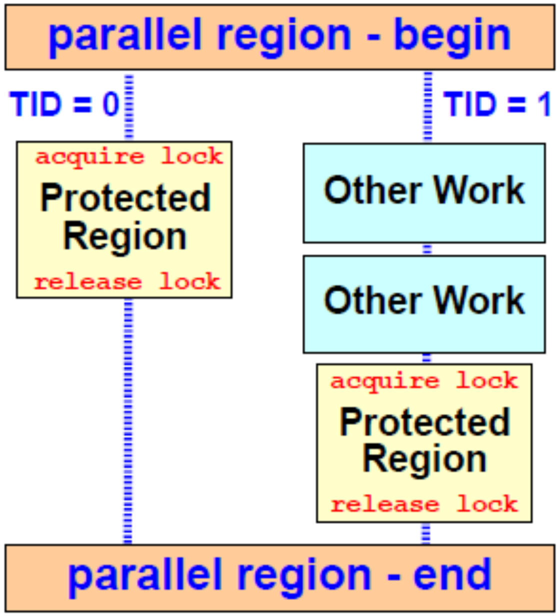
<u>Difference from critical section:</u>

- 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.

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.

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;
   }
```

schedule (type , chunksize)

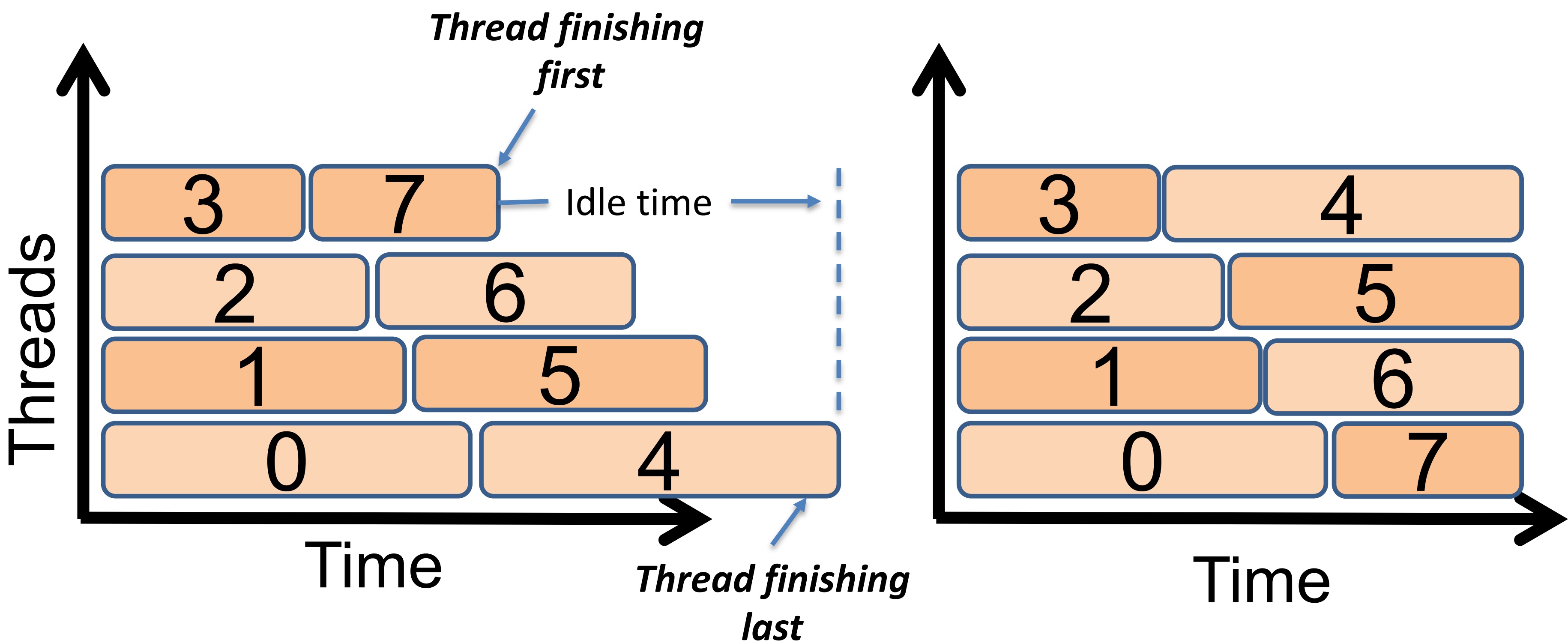
Controls how loop iterations are assigned

- 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

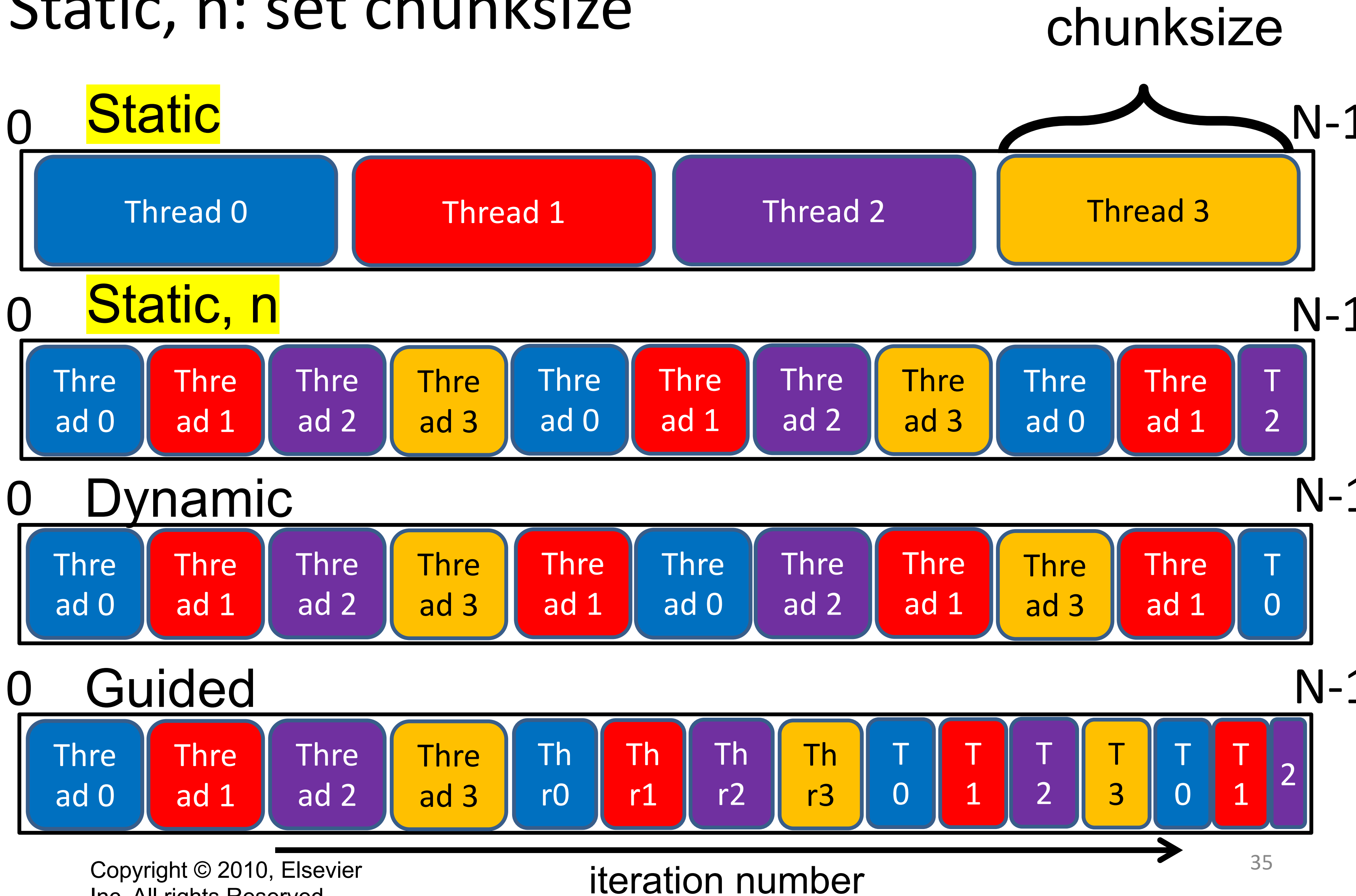
• Consecutive iterations are broken into chunks
• Total number = chunksize
• Positive integer
• Default is 1

schedule types can prevent load imbalance

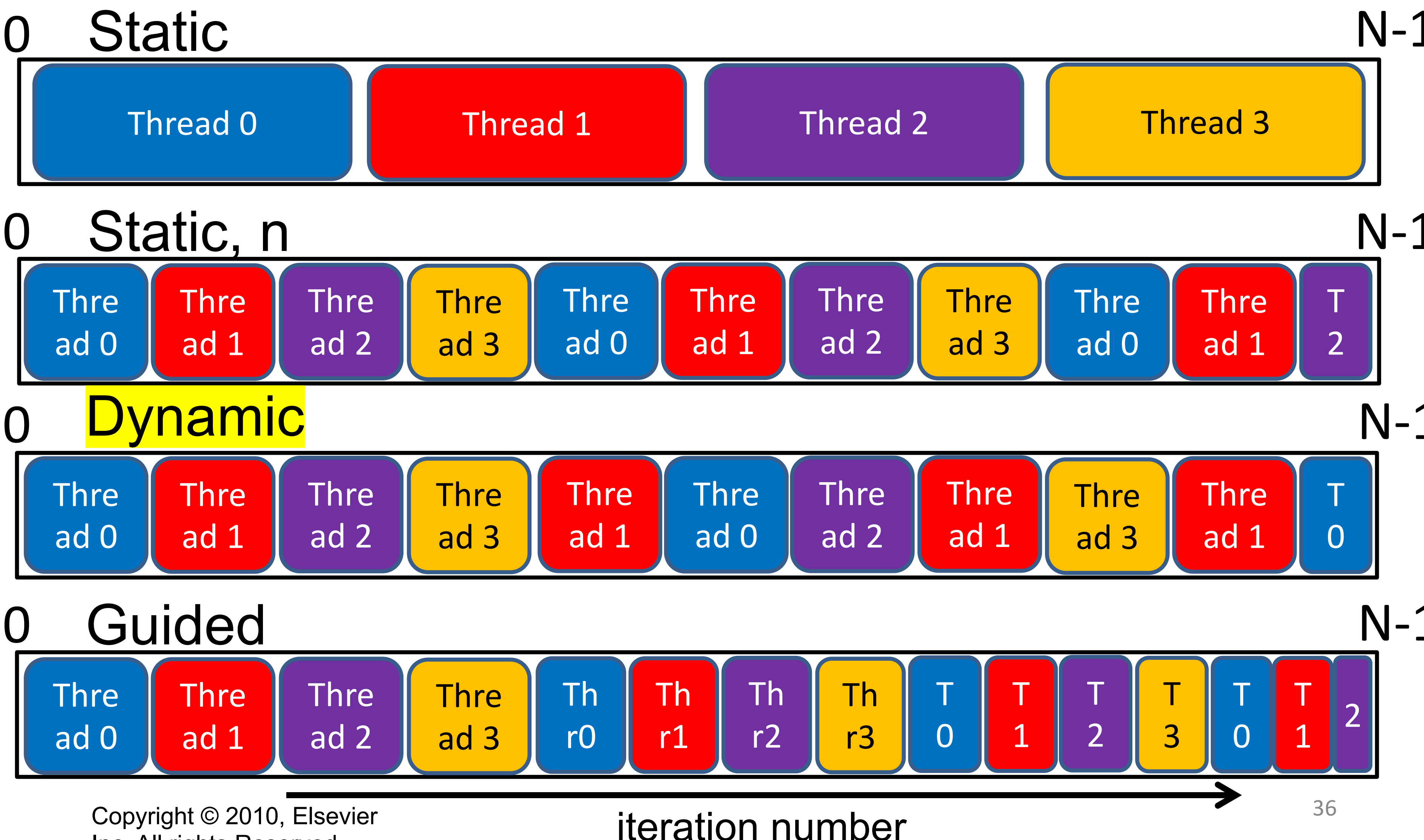
Static schedule vs Dynamic schedule



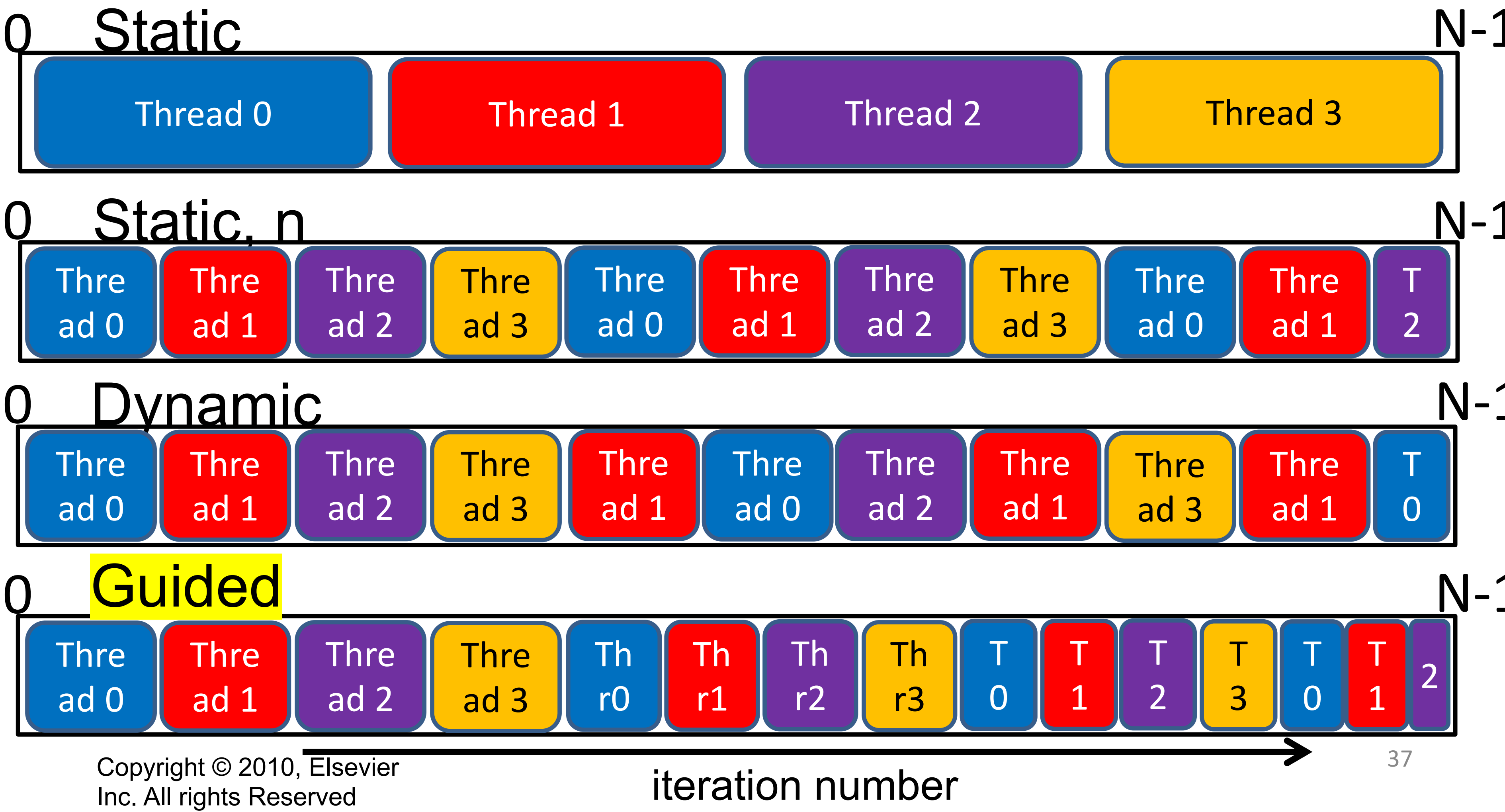
Static: default
Static, n: set chunksize



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



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



Matrix-vector multiplication

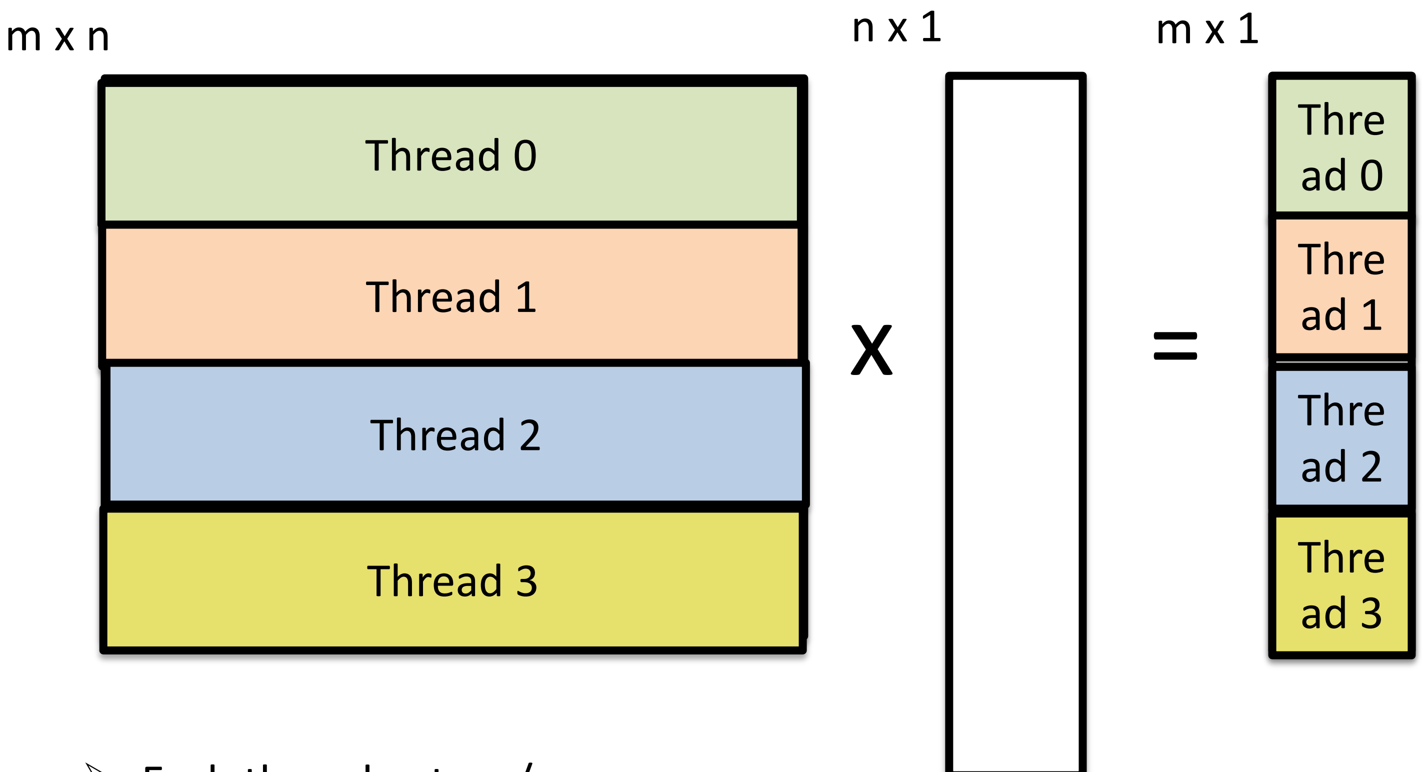
$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

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

```
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

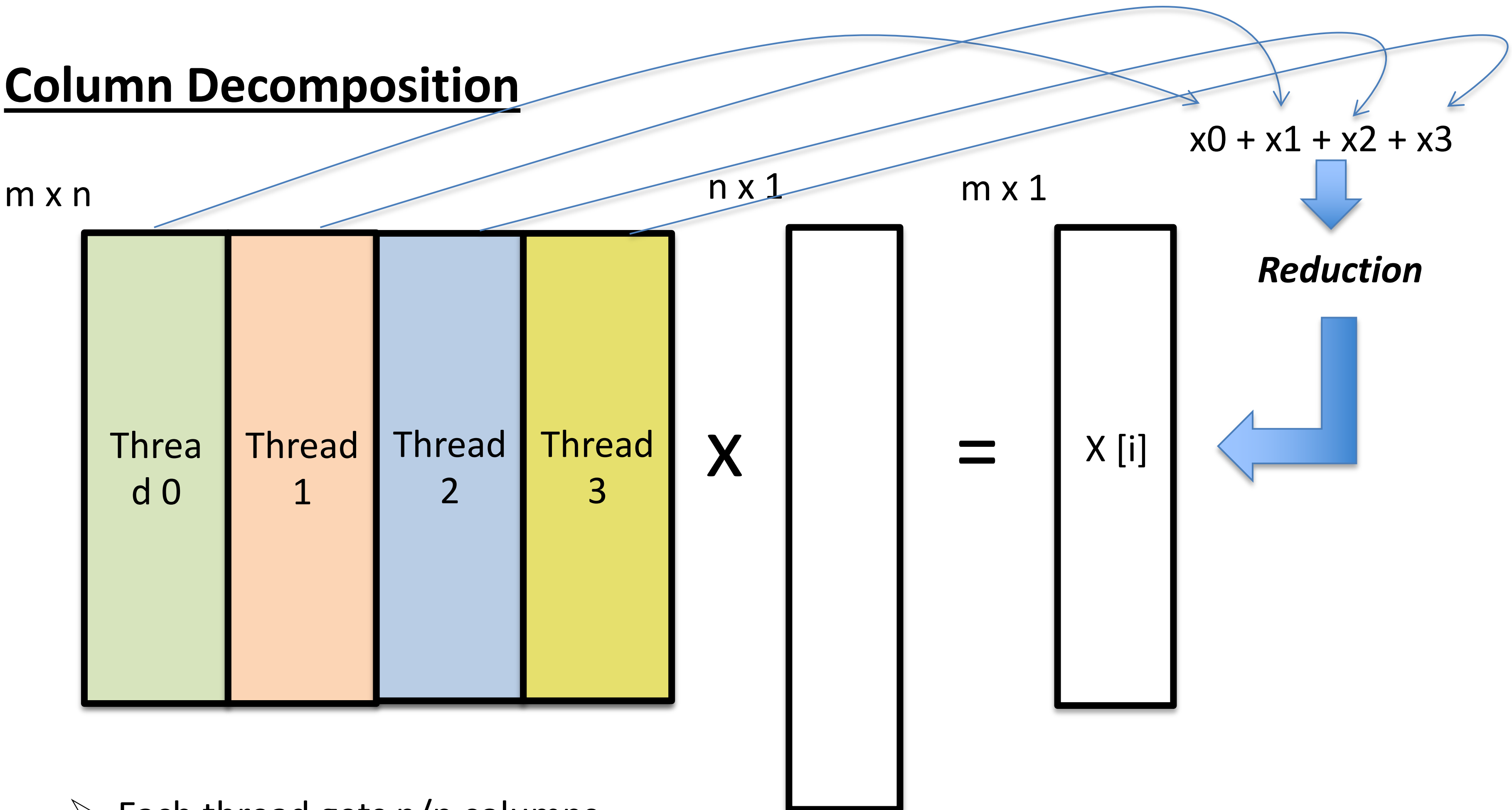
Row Decomposition



- 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