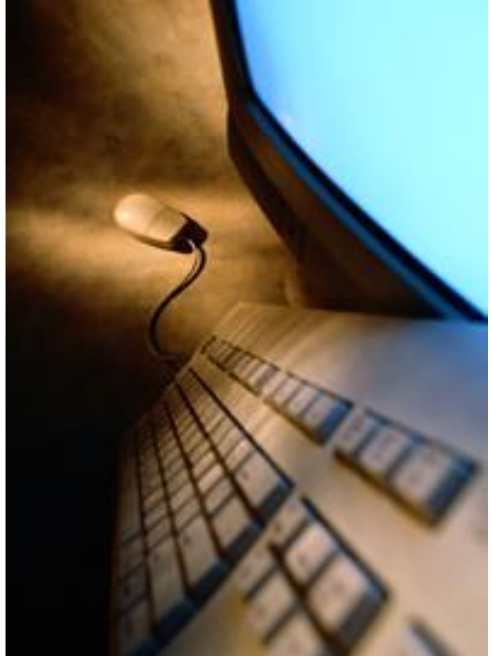
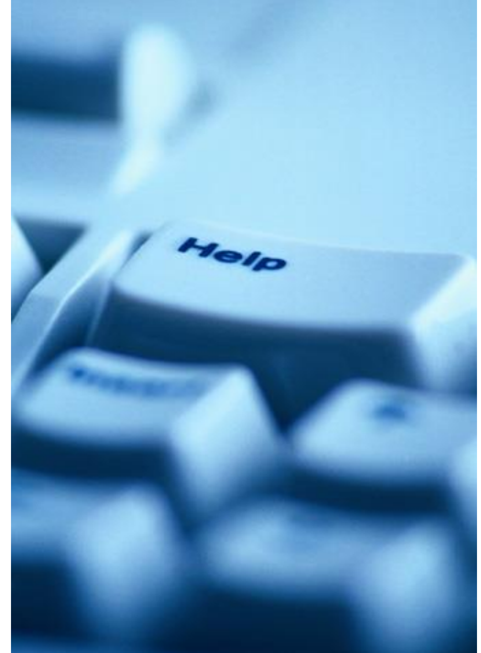
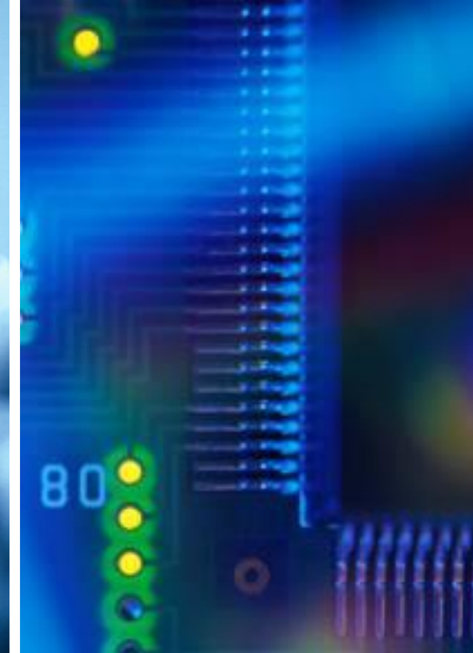






并行与分布式计算

Parallel & Distributed Computing

陈鹏飞

数据科学与计算机学院

2018-04-27

Lecture 7 — Performance Optimization in OpenMP

Pengfei Chen

School of Data and Computer Science

April 27, 2018

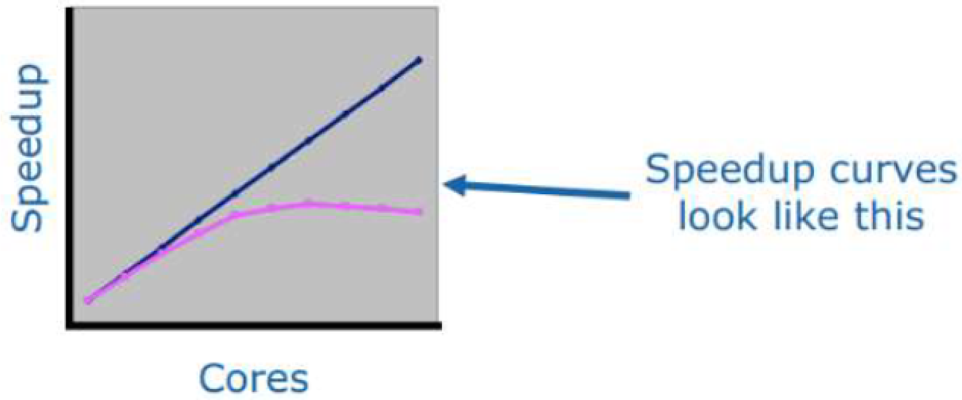
Outline:

- **Define speedup and efficiency**
  - **Use Amdahl's Law to predict maximum speedup**
- **Techniques for Performance Optimization of Parallel Programs**
  - **Rule of thumb**
    - **Start with best sequential algorithm**
    - **Maximize locality**
  - **Scheduling**
  - **Loop transformations**
    - **Loop fission (裂变、分离)**
    - **Loop fusion （聚变，合并）**
    - **Loop exchange （交换）**

https://en.wikipedia.org/wiki/Loop_fission_and_fusion

Speedup

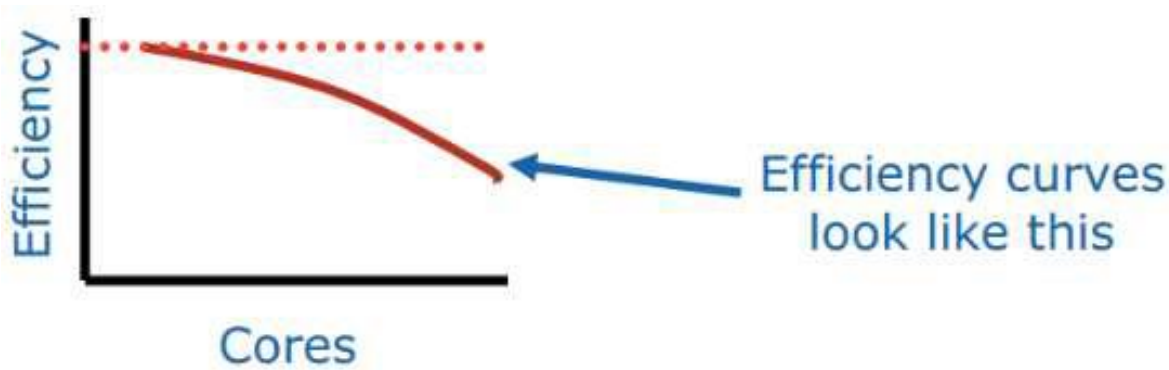
- Speedup is the ratio between sequential execution time and parallel execution time
- For example, if the sequential program executes in 6 seconds and the parallel program executes in 2 seconds, the speedup is 3X



Intel, "Introduction to Parallel Programming," 2009

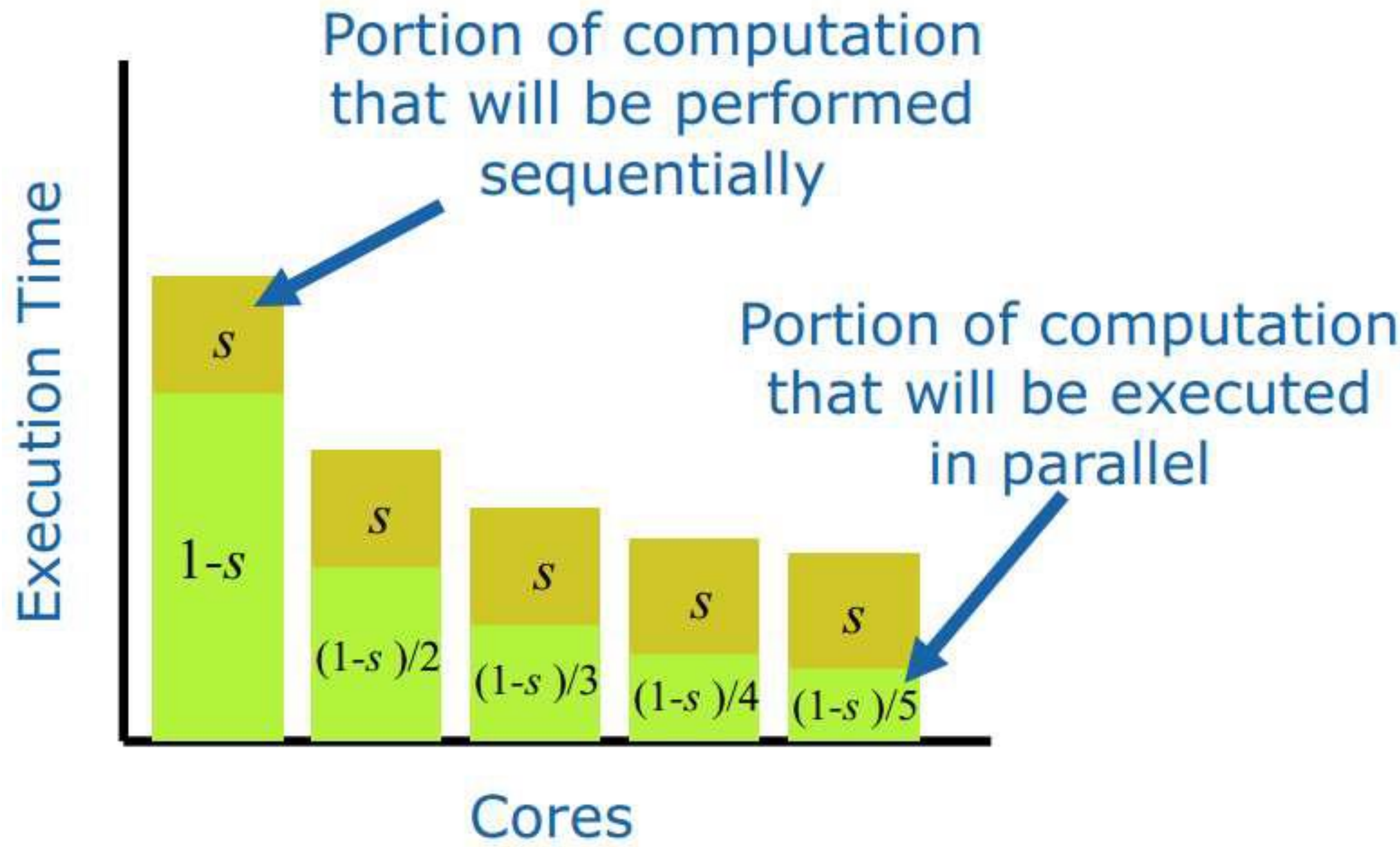
Efficiency (效率)

- **Efficiency**
  - **A measure of core utilization**
  - **Speedup divided by the number of cores**
- **Example**
  - **Program achieves speedup of 3 on 4 cores**
  - **Efficiency is 3/4 = 75%**



Intel, "Introduction to Parallel Programming," 2009

Idea Behind Amdahl's Law

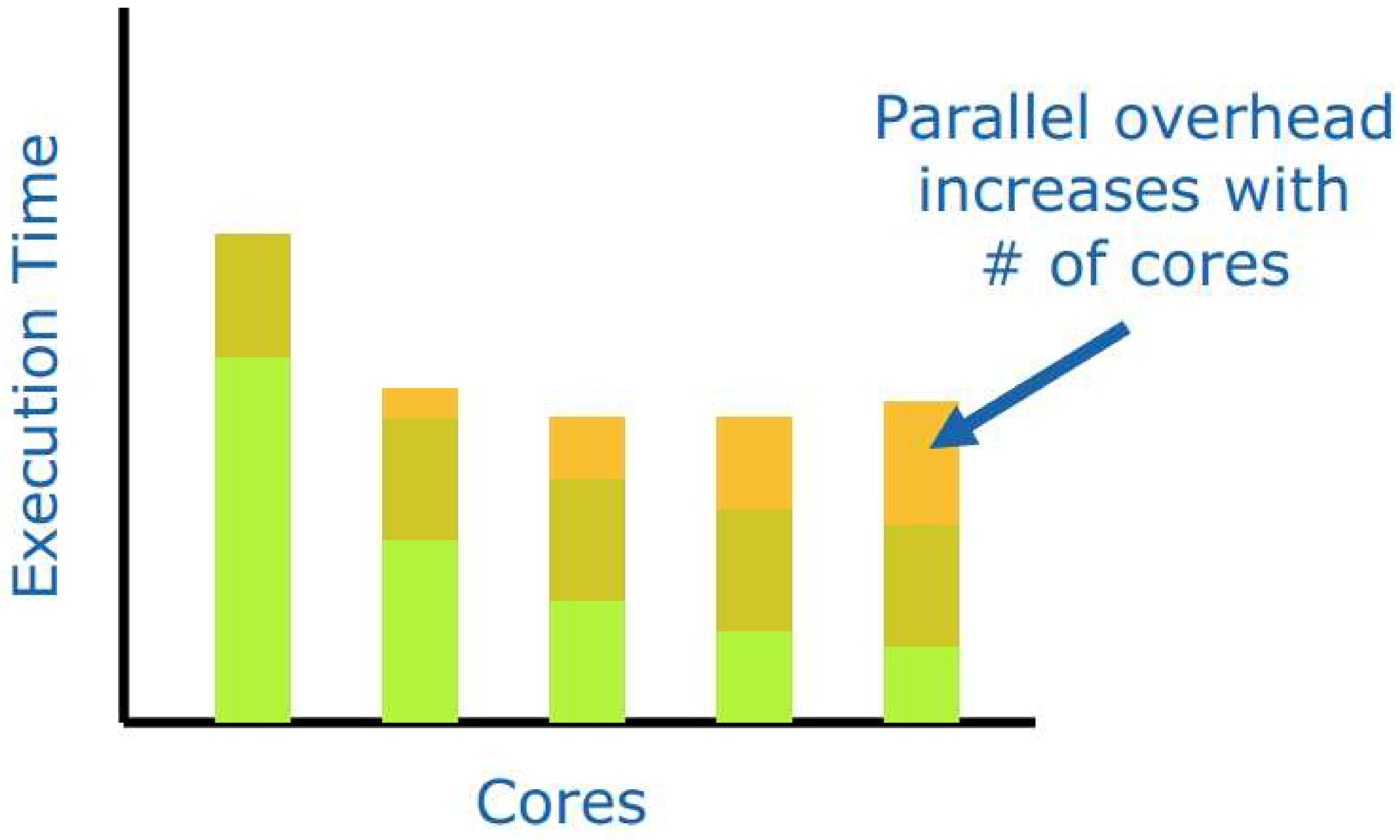


Intel, "Introduction to Parallel Programming," 2009

Amdahl's Law is Too Optimistic （乐观）

- **Amdahl's Law ignores parallel processing overhead**
- **Examples of this overhead include time spent creating and terminating threads**
- **Parallel processing overhead is usually an increasing**
- **function of the number of cores (threads)**

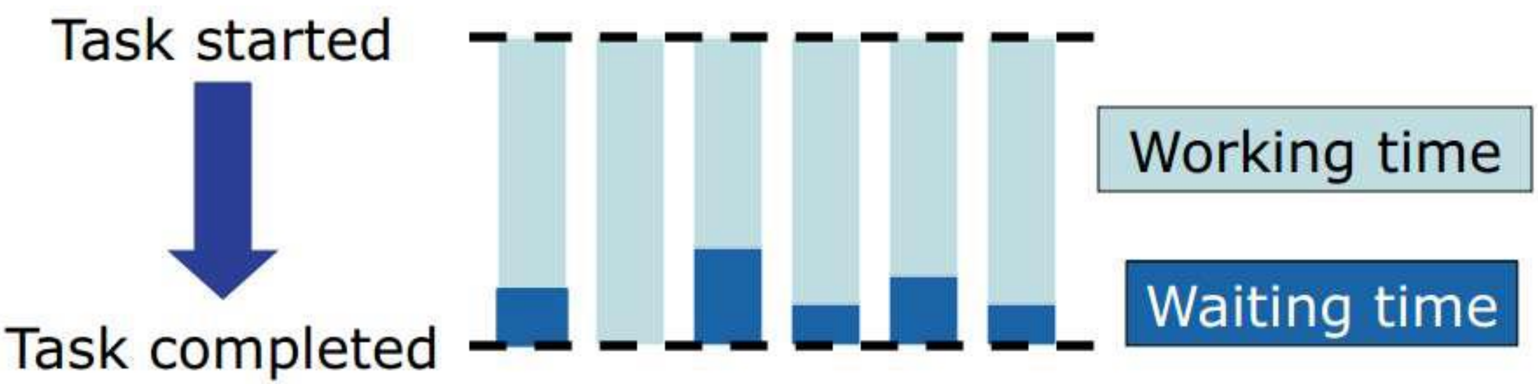
Graph with Parallel Overhead Added



Intel, “Introduction to Parallel Programming,” 2009

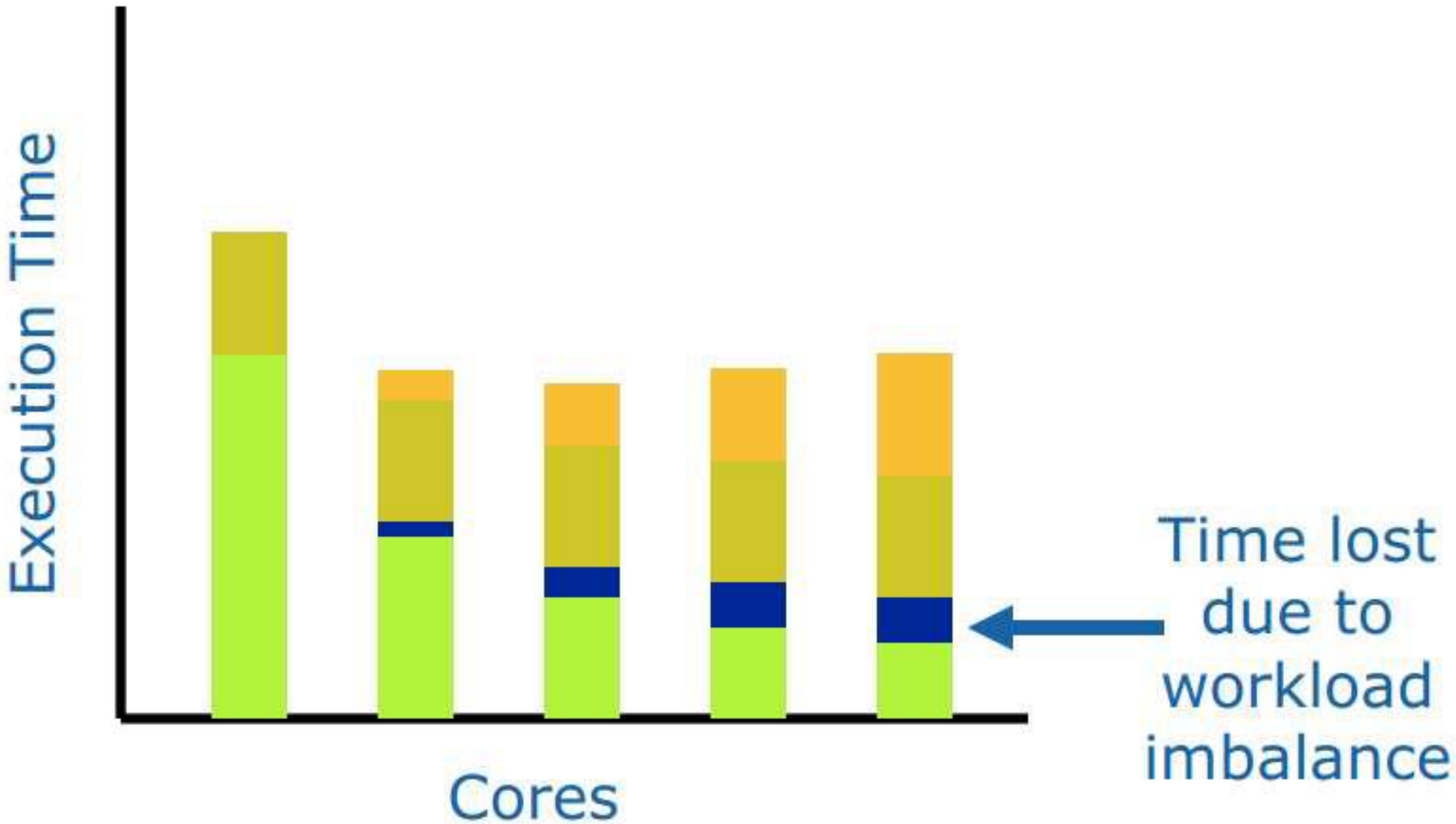
Other Optimistic Assumptions

- **Amdahl's Law assumes that the computation divides evenly among the cores**
- **In reality, the amount of work does not divide evenly among the cores**
- **Core waiting time is another form of overhead**



Intel, "Introduction to Parallel Programming," 2009

Graph with Workload Imbalance Added



Intel, "Introduction to Parallel Programming," 2009

Graph with Workload Imbalance Added

- Program executes in 5 seconds
- Profile reveals 80% of time spent in function alpha, which we can execute in parallel
- What would be maximum speedup on 2 cores?

$$\psi \le \frac{1}{0.2+(1-0.2)/2} = \frac{1}{0.6} \approx 1.67$$

- New execution time ≥ 5 sec / 1.67 = 3 seconds

Intel, "Introduction to Parallel Programming," 2009

More General Speedup Formula

n	problem size
p	number of cores
$\psi(n,p)$	Speedup for problem (of size n on p cores)
$\sigma(n)$	Time spent in sequential portion of code
$\varphi(n)$	Time spent in parallel portion of code
$\kappa(n,p)$	Parallel overhead

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

Intel, "Introduction to Parallel Programming," 2009

Amdahl's Law: Maximum Speedup

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

Assumes parallel work divides perfectly among available cores

This term is set to 0

Intel, "Introduction to Parallel Programming," 2009

The Amdahl Effect

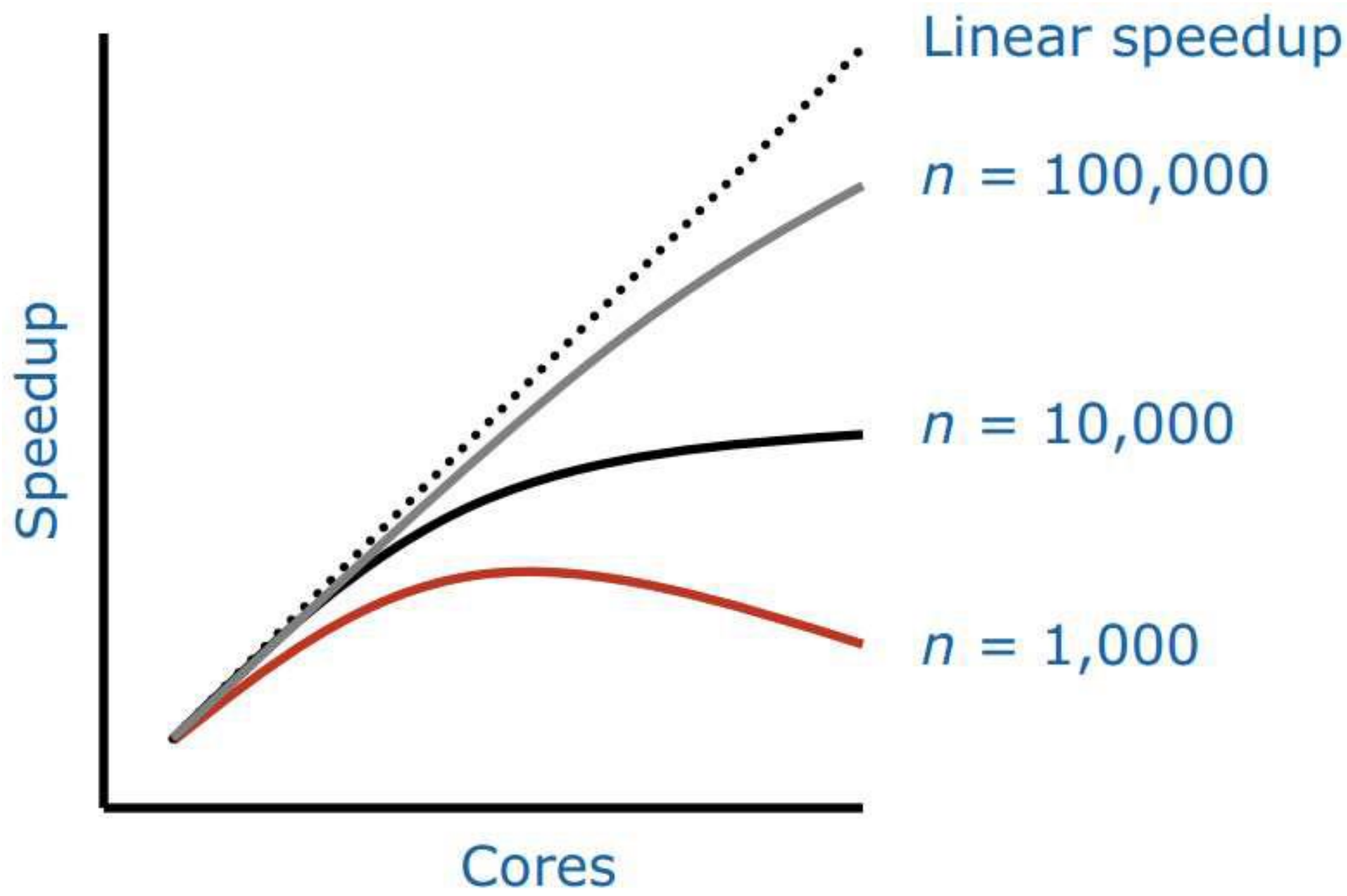
$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

As $n \to \infty$ these terms dominate

Speedup is an increasing function of problem size

Intel, "Introduction to Parallel Programming," 2009

Illustration of the Amdahl Effect



Intel, "Introduction to Parallel Programming," 2009

Start with Best Sequential Algorithm

Don't confuse "speedup" with "speed"

Speedup: ratio of program's execution time on 1 core

to its execution time on p cores

What if start with inferior sequential algorithm?

Naïve, higher complexity algorithms

- **Easier to make parallel**
- **Usually don't lead to fastest parallel algorithm**

Intel, "Introduction to Parallel Programming," 2009

Start with Best Sequential Algorithm

Suppose we want to compute `1 + 2 + 3 + ... + n`

The Parallel Way

```
 1 +  2 +  3.. + 10
11 + 12 + 13.. + 20
21 + 22 + 23.. + 30
31 + 32 + 33.. + 40
          .
          .
          .
```

And then we add up the partial sums. **O(*n*)** additions, divided by ***p*** processors.

We can use hundreds or thousands of processors, and keep them all busy.

The Serial Way

```
n*(n+1)/2
```

No matter what the value of ***n***, the serial way takes three instructions, and is **O(*1*)**. There's some parallelism here, but at minimum, it's two steps.

The serial way is
faster
lower power
less hardware
no compiler grief.

Which one would you buy?

P. Madden, "Why Parallel Computers Fails Commercially," DAC 2011.

Maximize Locality

- **Temporal locality**
  - **If a processor accesses a memory location, there is a good chance it will revisit that memory location soon**
- **Data locality**
  - **If a processor accesses a memory location, there is a good**
  - **chance it will visit a nearby location soon**
- **Programs tend to exhibit locality because they tend to have loops indexing through arrays**
- **Principle of locality makes cache memory worthwhile**

Intel, "Introduction to Parallel Programming," 2009

Parallel Processing and Locality

- **Multiple cores ⇒ multiple caches**
  - **When a core writes a value, the system must ensure no core tries to reference an obsolete value (cache coherence problem)**
  - **A write by one core can cause the invalidation of another core's copy of cache line, leading to a cache miss**
- **Rule of thumb (首要原则): Better to have different cores manipulating totally different chunks of arrays**
- **We say a parallel program has good locality if core's memory writes tend not to interfere with the work being done by other cores**

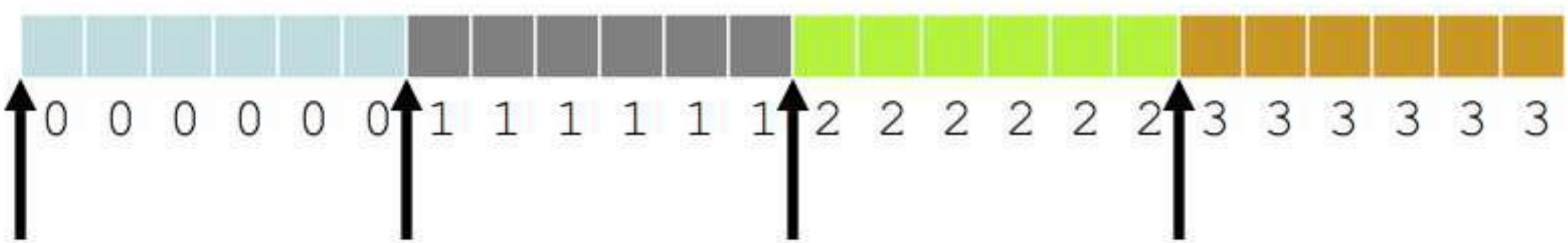
Intel, "Introduction to Parallel Programming," 2009

Example: Array Initialization

```
for (i = 0; i < N; i++) a[i] = 0;
```

Terrible allocation of work to processors

Better allocation of work to processors...

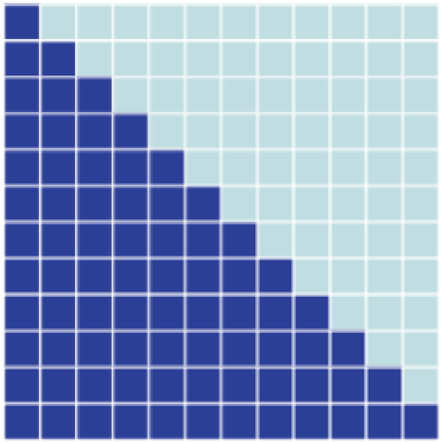


unless sub-arrays map to same cache lines!

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

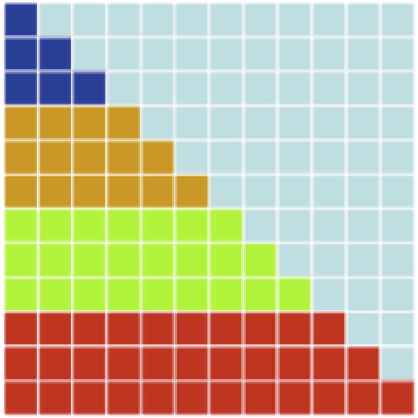
```
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

```
#pragma omp parallel for
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = ...;
```



Typically, the iterations are divided by the number of threads and assigned as chunks to a thread

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling

- **Loop schedule**
  - **How loop iterations are assigned to threads**
  - **Static schedule**
    - **Iterations assigned to threads before execution of loop**
  - **Dynamic schedule**
    - **Iterations assigned to threads during execution of loop**
- **The OpenMP** *schedule* **clause affects how loop iterations are mapped onto threads**

Intel, "Introduction to Parallel Programming," 2009

The *schedule* **Clause**

schedule (static [, chunk])

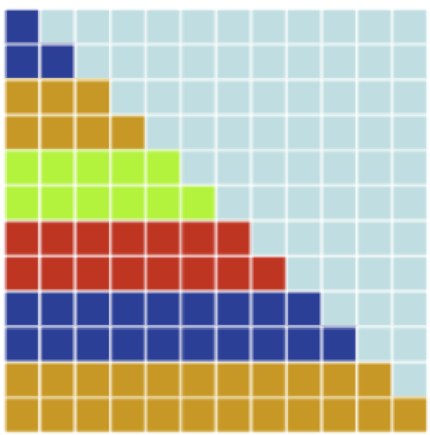
- **Blocks of iterations of size "chunk" to threads**
- **Round robin distribution**
- **Low overhead, may cause load imbalance**

Best used for predictable and similar work per iteration

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

```
#pragma omp parallel for schedule(static, 2)
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = …;
```



Intel, "Introduction to Parallel Programming," 2009

The *schedule* Clause

schedule (dynamic [, chunk])

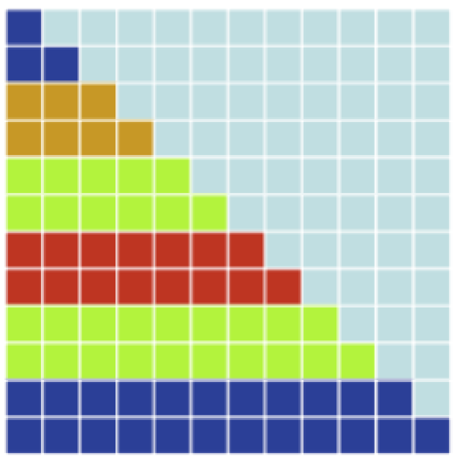
- Threads grab "chunk" iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

Best used for unpredictable or highly variable work

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

```
#pragma omp parallel for schedule(dynamic, 2)
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = ...;
```



Intel, "Introduction to Parallel Programming," 2009

The *schedule* **Clause**

schedule (guided [, chunk])

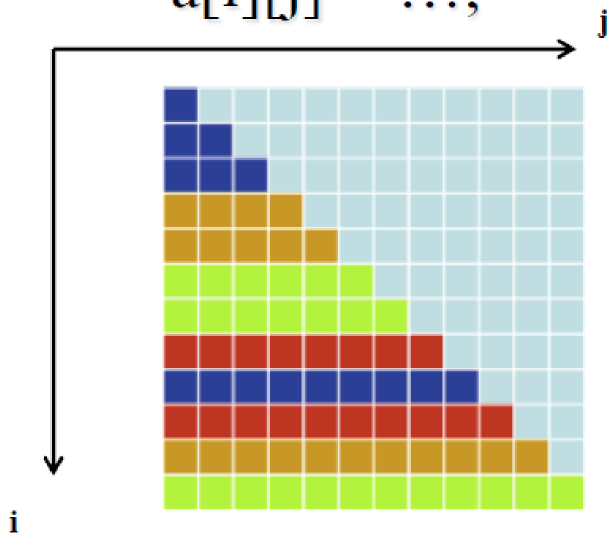
- **Dynamic schedule starting with large block**
- **Size of the blocks shrink; no smaller than "chunk"**
- **The initial block is proportional to**
  - **number_of_iterations / number_of_threads**
- **Subsequent blocks are proportional to**
  - **number_of_iterations_remaining / number_of_threads**

Best used as a special case of dynamic to reduce scheduling overhead when the computation gets progressively more time consuming

Intel, "Introduction to Parallel Programming," 2009

Loop Scheduling Example

```
#pragma omp parallel for schedule(guided)
for (int i = 0; i < 12; i++)
    for (int j = 0; j <= i; j++)
        a[i][j] = ...;
```



Intel, “Introduction to Parallel Programming,” 2009

Loop Transformations

Loop fission

Loop fusion

Loop exchange

Intel, "Introduction to Parallel Programming," 2009

Loop Fission

- **Begin with single loop having loop-carried dependence**
- **Split loop into two or more loops**
- **New loops can be executed in parallel**

Intel, "Introduction to Parallel Programming," 2009

Before Loop Fission

```
float *a, *b;
for (int i = 1; i < N; i++) {
    // perfectly parallel
    if (b[i] > 0.0) a[i] = 2.0;
    else a[i] = 2.0 * fabs(b[i]);
    // loop-carried dependence
    b[i] = a[i-1];
}
```

Intel, “Introduction to Parallel Programming,” 2009

After Loop Fission

```
float *a, *b;
#pragma omp parallel
{
#pragma omp for
    for (int i = 1; i < N; i++) {
        if (b[i] > 0.0) a[i] = 2.0;
        else a[i] = 2.0 * fabs(b[i]);
    }
#pragma omp for
    for (int i = 1; i < N; i++) b[i] = a[i-1];
}
```

Intel, “Introduction to Parallel Programming,” 2009

Loop Fusion

- The opposite of loop fission
- Combine loops increase grain size

Intel, “Introduction to Parallel Programming,” 2009

Before Loop Fusion

```
float *a, *b, x, y;
...
for (int i = 0; i < N; i++) a[i] = foo(i);
x = a[N-1] – a[0];
for (int i = 0; i < N; i++) b[i] = bar(a[i]);
y = x * b[0] / b[N-1];
```

无副作用

- **Assume functions *foo* and *bar* are side-effect free**

Intel, "Introduction to Parallel Programming," 2009

After Loop Fusion

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    a[i] = foo(i);
    b[i] = bar(a[i]);
}
x = a[N-1] – a[0];
y = x * b[0] / b[N-1];
```

- **Now one barrier instead of two**

Intel, “Introduction to Parallel Programming,” 2009

Loop Fusion with Replicated Work

- Every thread iteration has a cost
- Example: Barrier synchronization
- Sometimes it’s faster for threads to replicate work than to go through a barrier synchronization

Intel, “Introduction to Parallel Programming,” 2009

Before Work Replication

```
for (int i = 0; i < N; i++) a[i] = foo(i);
x = a[0] / a[N-1];
for (int i = 0; i < N; i++) b[i] = x * a[i];
```

适合

- **Both *for* loops are amenable to parallelization**

Intel, "Introduction to Parallel Programming," 2009

First OpenMP Attempt

```
#pragma omp parallel
{
#pragma omp for
    for (int i = 0; i < N; i++) a[i] = foo(i);
#pragma omp single
    x = a[0] / a[N-1]; // implicit barrier
#pragma omp for
    for (int i = 0; i < N; i++) b[i] = x * a[i];
}
```

- **Synchronization among threads required if x is shared and one thread performs assignment**

Intel, “Introduction to Parallel Programming,” 2009

After Work Replication

```
#pragma omp parallel private (x)
{
    x = foo(0) / foo(N-1);
#pragma omp for
    for (int i = 0; i < N; i++) {
        a[i] = foo(i);
        b[i] = x * a[i];
    }
}
```

Intel, "Introduction to Parallel Programming," 2009

Loop Fusion Example

```
#define N 23
#define M 1000
...
for (int k = 0; k < N; k++)
    for (int j = 0; j < M; j++)
        w_new[k][j] = DoSomeWork(w[k][j], k, j);
```

- **Prime number of iterations will never be perfectly load balanced**
- **Parallelize inner loop? Are there enough iterations to overcome overhead?**

Intel, “Introduction to Parallel Programming,” 2009

Loop Fusion Example

```
#define N 23
#define M 1000
...
for (int kj = 0; kj < N*M; kj++) {
    k = kj / M;
    j = kj % M;
    w_new[k][j] = DoSomeWork(w[k][j], k, j);
}
```

- **Larger number of iterations gives better opportunity for load balance and hiding overhead**
- **DIV and MOD are overhead**

Intel, "Introduction to Parallel Programming," 2009

Loop Exchange (Inversion)

- **Nested *for* loops may have data dependences that prevent parallelization**
- **Exchanging the nesting of *for* loops may**
  - **Expose a parallelizable loop**
  - **Increase grain size**
  - **Improve parallel program's locality**

Intel, "Introduction to Parallel Programming," 2009

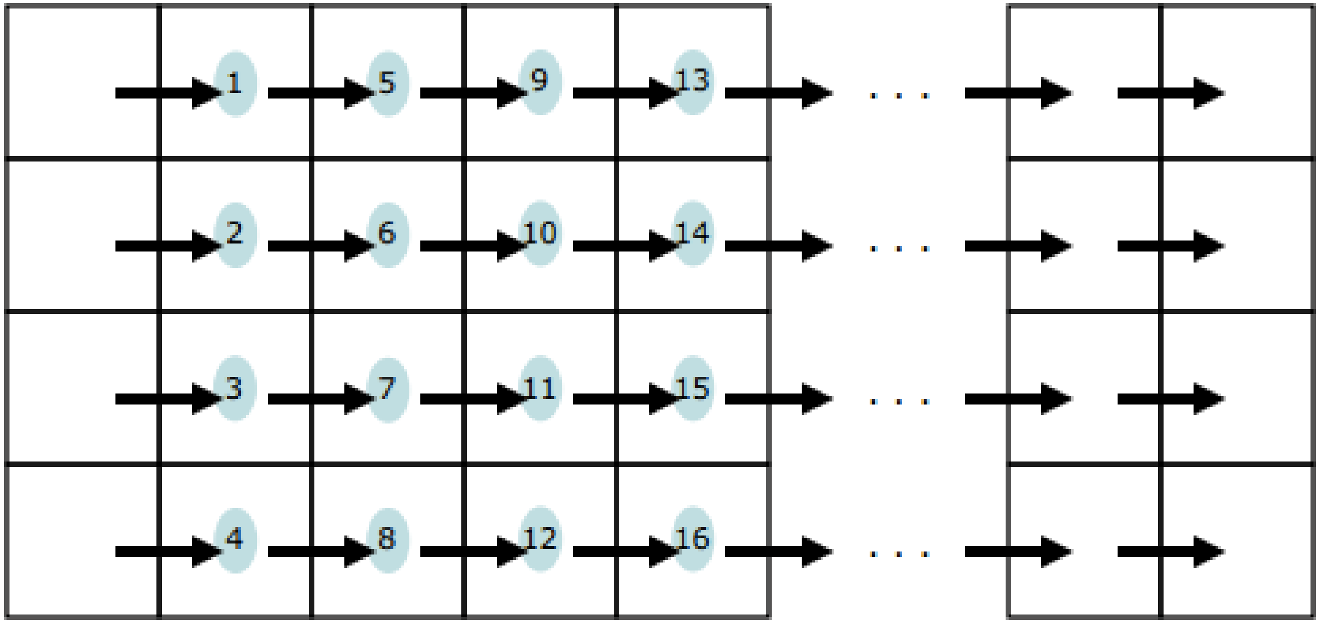
Loop Exchange (Inversion)

- **Nested *for* loops may have data dependences that prevent parallelization**
- **Exchanging the nesting of *for* loops may**
  - **Expose a parallelizable loop**
  - **Increase grain size**
  - **Improve parallel program's locality**

Intel, "Introduction to Parallel Programming," 2009

Loop Exchange Example

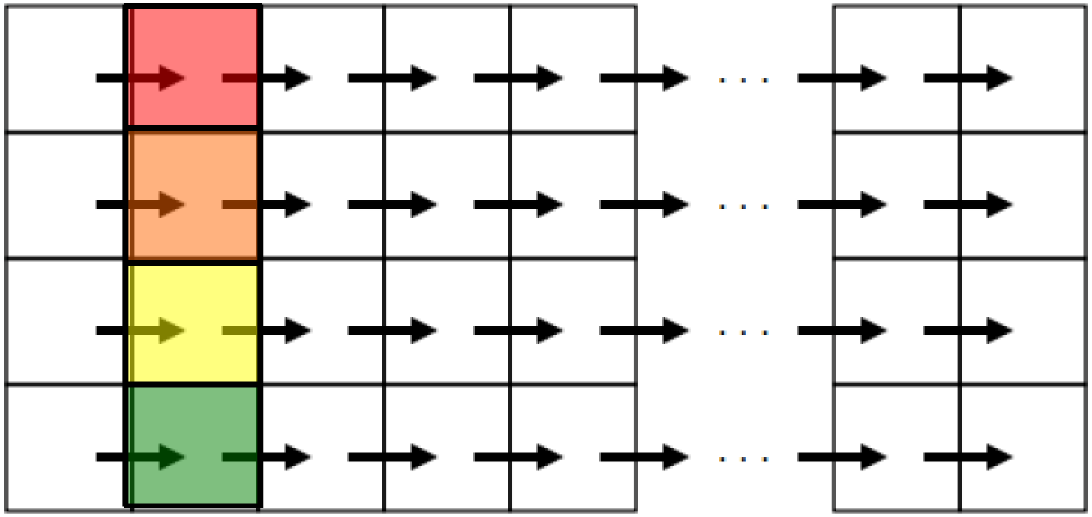
```
for (int j =1; j < n; j++)
    for (int i = 0; i < m; i++)
        a[i][j] = 2 * a[i][j-1];
```



Intel, "Introduction to Parallel Programming," 2009

Before Loop Inversion

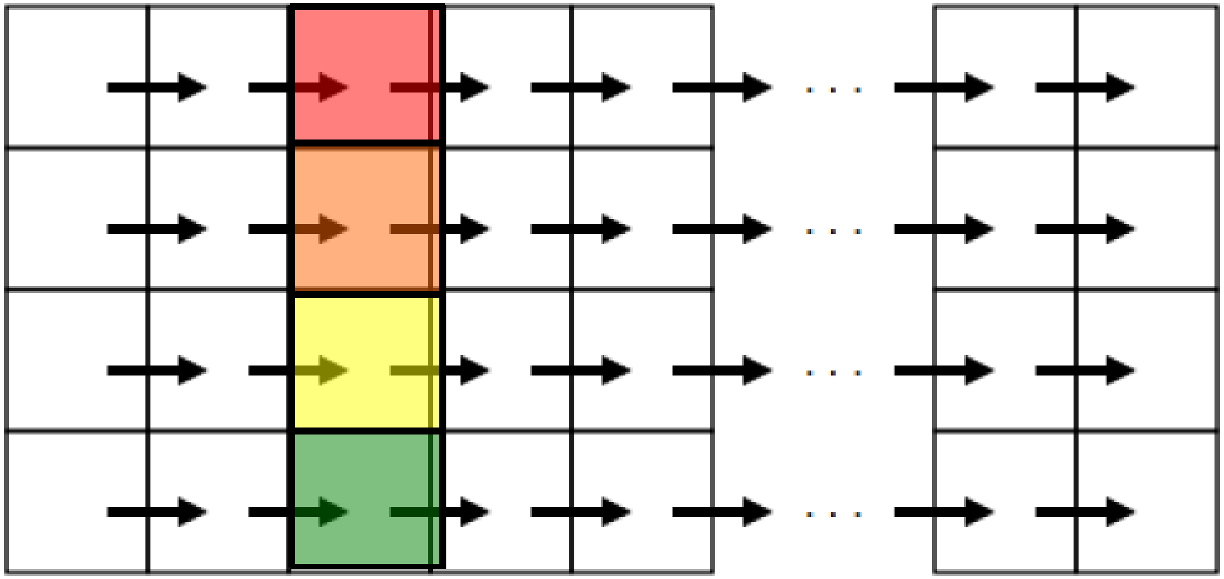
```
for (int j =1; j < n; j++)
#pragma omp parallel for
  for (int i = 0; i < m; i++)
      a[i][j] = 2 * a[i][j-1];
```



Intel, "Introduction to Parallel Programming," 2009

Before Loop Inversion

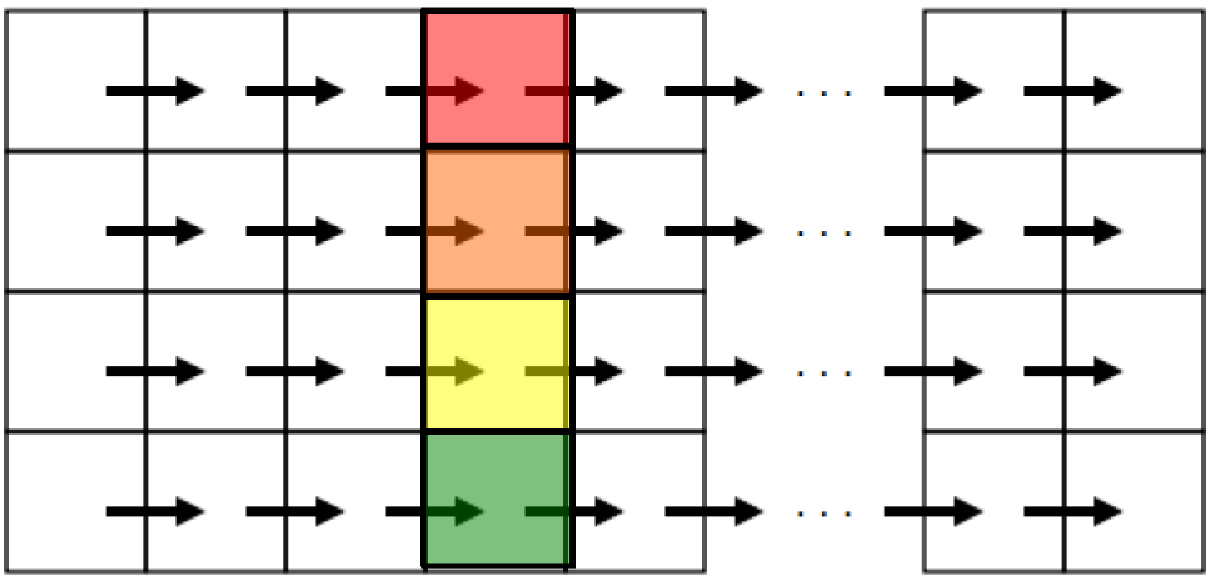
```
for (int j =1; j < n; j++)
#pragma omp parallel for
  for (int i = 0; i < m; i++)
      a[i][j] = 2 * a[i][j-1];
```



Intel, "Introduction to Parallel Programming," 2009

Before Loop Inversion

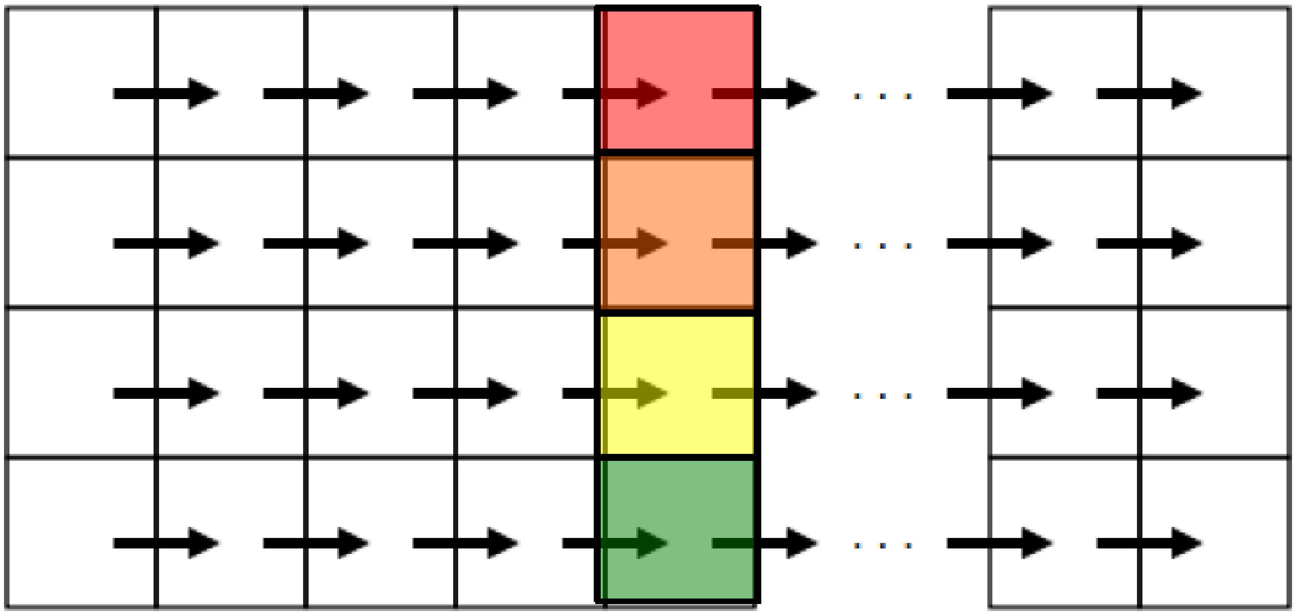
```
for (int j =1; j < n; j++)
#pragma omp parallel for
  for (int i = 0; i < m; i++)
      a[i][j] = 2 * a[i][j-1];
```



Intel, “Introduction to Parallel Programming,” 2009

Before Loop Inversion

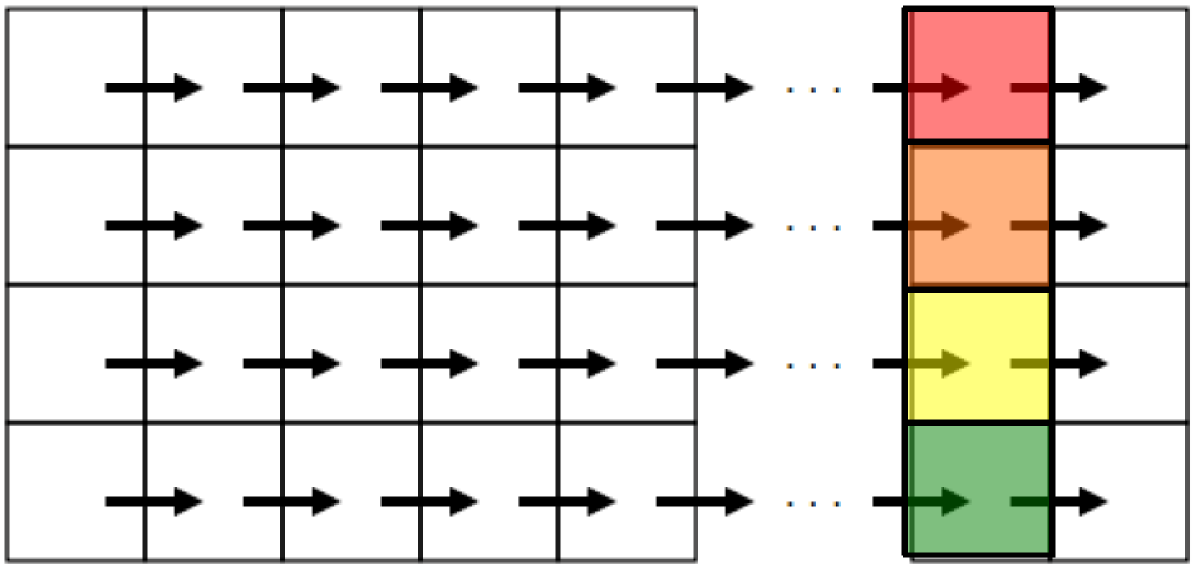
```
for (int j =1; j < n; j++)
#pragma omp parallel for
    for (int i = 0; i < m; i++)
        a[i][j] = 2 * a[i][j-1];
```



Intel, “Introduction to Parallel Programming,” 2009

Before Loop Inversion

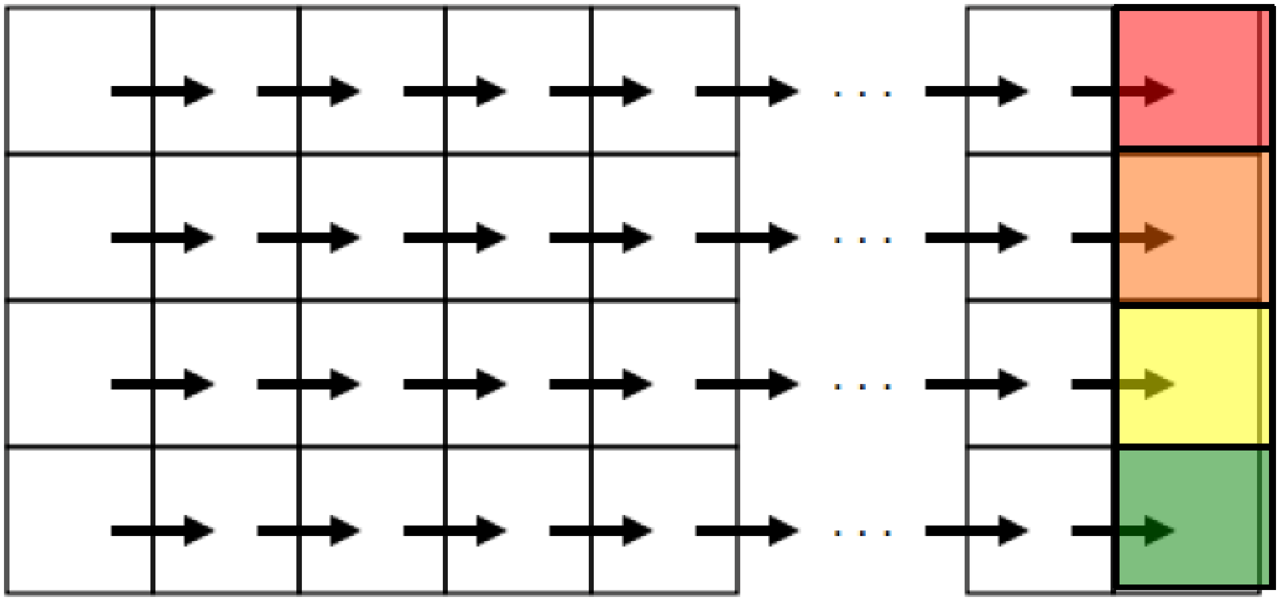
```
for (int j =1; j < n; j++)
#pragma omp parallel for
  for (int i = 0; i < m; i++)
      a[i][j] = 2 * a[i][j-1];
```



Intel, "Introduction to Parallel Programming," 2009

Before Loop Inversion

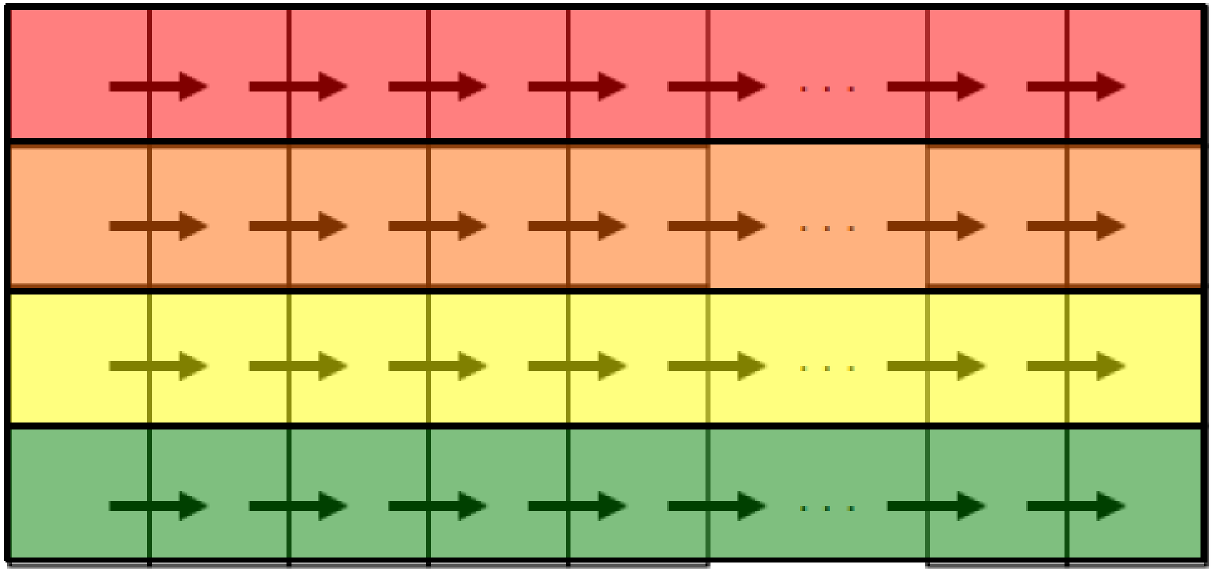
```
for (int j =1; j < n; j++)
#pragma omp parallel for
  for (int i = 0; i < m; i++)
      a[i][j] = 2 * a[i][j-1];
```



Intel, "Introduction to Parallel Programming," 2009

Before Loop Inversion

```
for (int j =1; j < n; j++)
#pragma omp parallel for
  for (int i = 0; i < m; i++)
      a[i][j] = 2 * a[i][j-1];
```



Intel, "Introduction to Parallel Programming," 2009

Summary

- **Define speedup and efficiency**
  - **Use Amdahl's Law to predict maximum speedup**
- **Techniques for Performance Optimization of Parallel Programs**
  - **Rule of thumb**
    - **Start with best sequential algorithm**
    - **Maximize locality**
  - **Scheduling**
  - **Loop transformations**
    - **Loop fission (裂变、分离)**
    - **Loop fusion （聚变，合并）**
    - **Loop exchange （交换）**

Homework

测试不同调度方法

1. Consider a simple loop that calls a function dummy containing a programmable delay. All invocations of the function are independent of the others. Partition this loop across four threads using static, dynamic, and guided scheduling. Use different parameters for static and guided scheduling. Document the result of this experiment as the delay within the dummy function becomes large.

测试不同层次的循环并行后的性能

2. Implement and test the OpenMP program for computing a matrix-matrix (50 x 50) product. Use the OMP_NUM_THREADS environment variable to control the number of threads and plot the performance with varying numbers of threads. Consider three cases in which (i) only the outermost loop is parallelized; (ii) the outer two loops are parallelized; and (iii) all three loops are parallelized. What is the observed result from these three cases?

Thank You！