

Parallelism

Weng Kai

Why Parallelism

- No more speed, but more cores
- Sequential program runs on one core that doesn't benefit from multi-cores

Why not Automatically?

- Run multiple instances on multiple cores?
- Special hardware puts one program onto multiple cores?
- Special software translates one program into a multi-core version?

并行计算

- 通过多个处理器共同解决同一个计算问题，每一个处理器单独承担整个计算任务中的一部分
- 计算任务的分解是并行计算中的首要关键问题
 - 时间：流水线
 - 空间：分解-计算-同步

并行计算的条件

- 并行机
 - 单核内的SIMD
 - 多核/多机
 - 异构：GPU/FPGA
- 问题具有并行度，即可以分解
- 并行编程

求值累加

```
for ( int i=lo; i<hi; i++ ) {  
    sum += 1/a[i];  
}
```

- 可以分成几段，分别计算累加，得到n个结果，然后把n个结果加起来
- 不断地两两相加直到得到结果
- 并行会带来新的算法，这很难由程序从串行算法中发现

算法复杂度

- 算法复杂度由四则运算次数和存储空间大小两部分组成

x^n 的计算

以 $a[i]$ 表示 n 的二进制的每一位， k 为二进制位数

$y=1$

for $i=0$ to $k-1$ do

 If $a[i] = 1$ do

$y = y * x$

 else

$x = x * x$

$y = y * x$

$O(\log n)$

该算法不能并行

矩阵乘以向量

- $y = Ax$, 其中 A 是 $m \times n$ 的, y 与 x 分别是 m 和 n 维向量

$y = 0$

for $i = 0$ to $n - 1$ do

$y = y + A[i] * x[i]$

- $O(n^2)$

并发、并行、分布式

- 并发：一个程序的多个任务同时执行
- 并行：一个任务分解为多个子任务同时执行，协作完成一个问题
- 分布式：并行的计算在不同的计算机上进行
- 本课不研究并行如何计算，而是如何在编程语言层面表达并行

Parallel Execution

- Sequential:
 - S1, S2, S3, S4, S5, S6
- Which of these steps can be paralleled.

example

- sum of an array of numbers

- $\text{sum1} = \text{sum of lower half}$

- $\text{sum2} = \text{sum of upper half}$

- $\text{sum} = \text{sum1} + \text{sum2}$

async

finish

Seq. to Para.

- S1, S2, S3, S4, S5, S6

S1

finish {

 async {

 S2,S3

 }

}

S4, S5, S6

- S1, S2, S3, S4, S5, S6

S1

finish {

 async S2

 S3

 }

}

S4, S5, S6

Java Folk/Join Framework

```
public class Sum {  
    private static final int SIZE=100000;  
  
    private int[] a;  
    private int lo,hi;  
    private int sum;
```

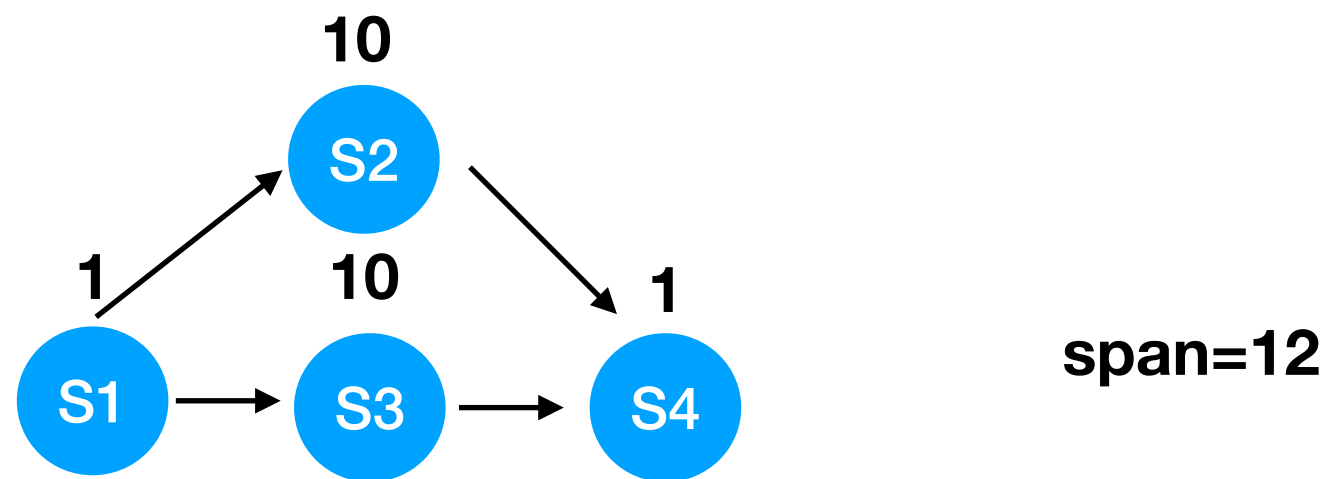
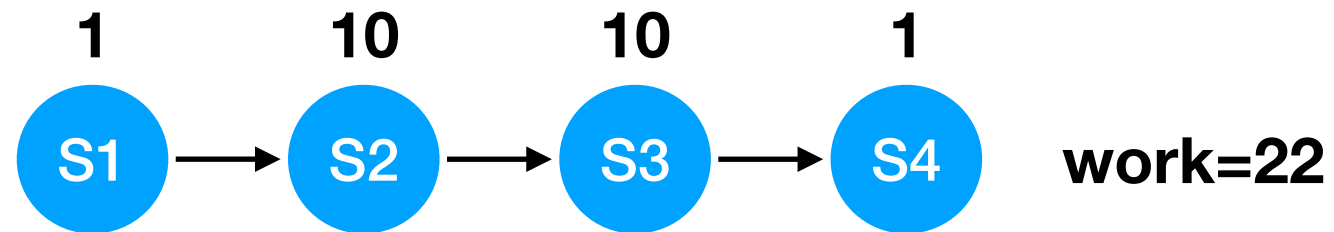
```
l.compute();  
r.compute();  
sum = l.sum+r.sum;
```

← async

← finish

```
public void compute() {  
    if ( lo == hi ) {  
        sum = a[lo];  
    } else if ( lo > hi ) {  
        sum = 0;  
    } else {  
        int mid = (lo+hi)/2;  
        Sum l = new Sum(a, lo, mid);  
        Sum r = new Sum(a, mid+1, hi);  
        l.compute();  
        r.compute();  
        sum = l.sum+r.sum;  
    }  
}
```

Computation Graphs



- $WORK(G)$ = sum of execution times of all nodes in G
- $SPAN(G)$ = length of the longest path in G
- ideal parallelism = $WORK(G)/SPAN(G)$

并行计算的度量

- 粒度：两次通信之间每个处理器计算工作量的大小
- 并行度：某一时刻多个处理器上可以同时执行的子任务个数
- 可扩展性：处理器数目增加时并行程序的表现
- 计算通信比：计算花费的时间/处理器间消息传递花费的时间

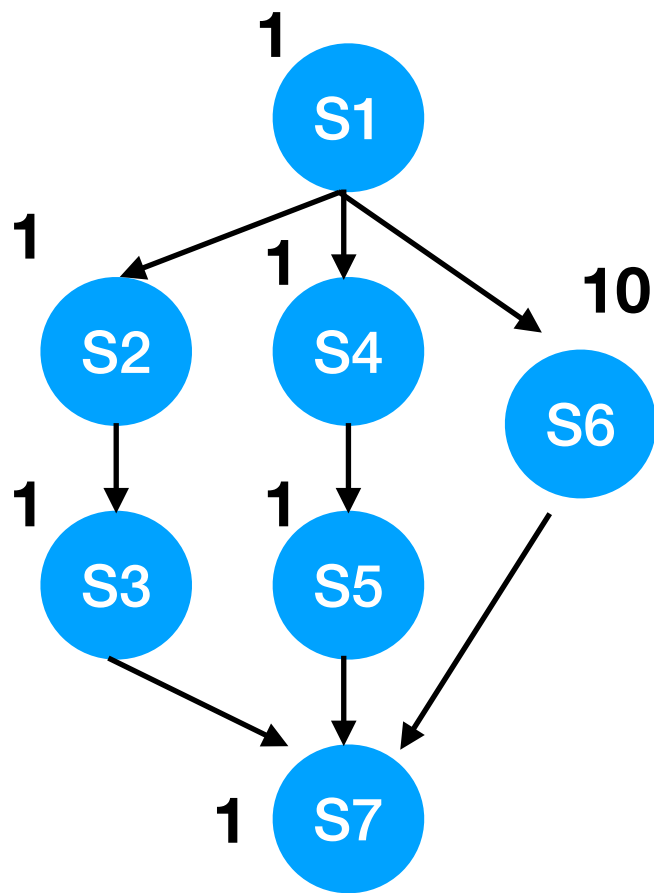
加速比

- 在p个核上的程序的加速比 $S = T_{\text{串行}} / T_{\text{并行}}$
- 理想 $S = p$

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
S/p	1.0	0.95	0.90	0.81	0.68

with multiple-cores

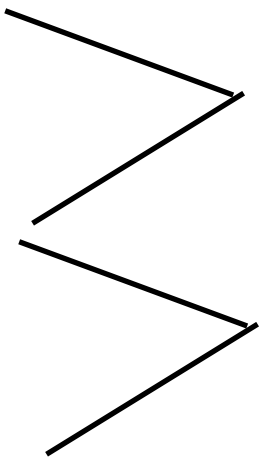
- It has different time consumption under different arrangement



Amdahl's Law

- $\text{speedup} \leq \text{work}/\text{span}$
- $q = \text{fraction of sequential work}$
 - $\text{speedup} \leq 1/q$

Functional Parallelism

- $A = F(B)$
 - $C = G(A)$
 - $D = H(A)$
- 
- wait
- para

- $FA = \text{future}\{F(B)\}$
- $FC = \text{future}\{G(FA.get())\}$
- $FD = \text{future}\{H(FA.get())\}$



object

computation graphs?

Future Tasks

- extends RecursiveTask (not RecursiveAction)
- compute() returns a result
- join() waits and gets the value
 - return left.compute()+right.join()

Memoization

- $Y1 = G(X1)$
- $Y2 = G(X2)$
- $Y3 = G(X1)$
- $Y1 = G(X1)$
- $\text{insert}(G, X1, Y1)$
- $Y2 = G(X2)$
- $Y3 = \text{lookup}(g, x1)$
- $FY1 = \text{future}\{G(X1)\}$
- $\text{insert}(G, X1, FY1)$
- $Y2 = G(X2)$
- $Y3 = \text{lookup}(G, X1).get()$

to compute Fib

Data Race

- `async { sum1 = sum of lower half }`
- `sum2 = sum of upper half`
- `sum = sum1 + sum2`
- could be read-write or write-write

Determinism

- functional determinism
 - same input \rightarrow same output
- structure determinism
 - same input \rightarrow same computation graph

Parallel Loops

```
for (...) {
```

```
    compute(p);
```

```
}
```

```
finish {
```

```
    forall
```

```
    for (...) {
```

```
        async {compute(p)};
```

```
    }
```

```
}
```

Matrix Multiply

- $A \times B \rightarrow C$
 - $C[i][j] = \text{sum}(0 \text{ to } n-1: a[i][k] * b[k][j])$
- ```
for ([i,j] : [0:n-1,0:n-1]) {
 c[i][j] = 0;
 for (k: [0,n-1])
 c[i][j] += a[i][k]*b[k][j]
}
```

# Barriers in P. Loops

```
forall (i: [0:n-1]) {
```

```
 s = lookup(i);
```

```
 print("hello"+s);
```

```
 barrier
```

```
 print("bye"+s);
```

```
}
```

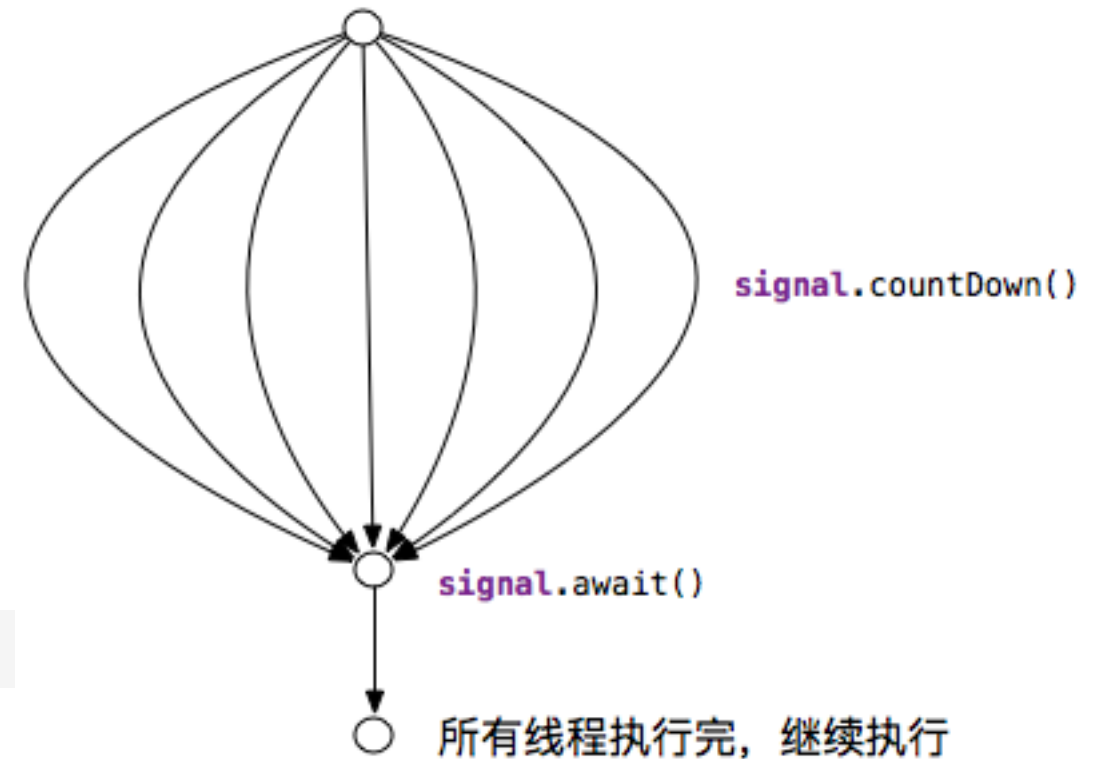
- executed in any order

# CountDownLatch

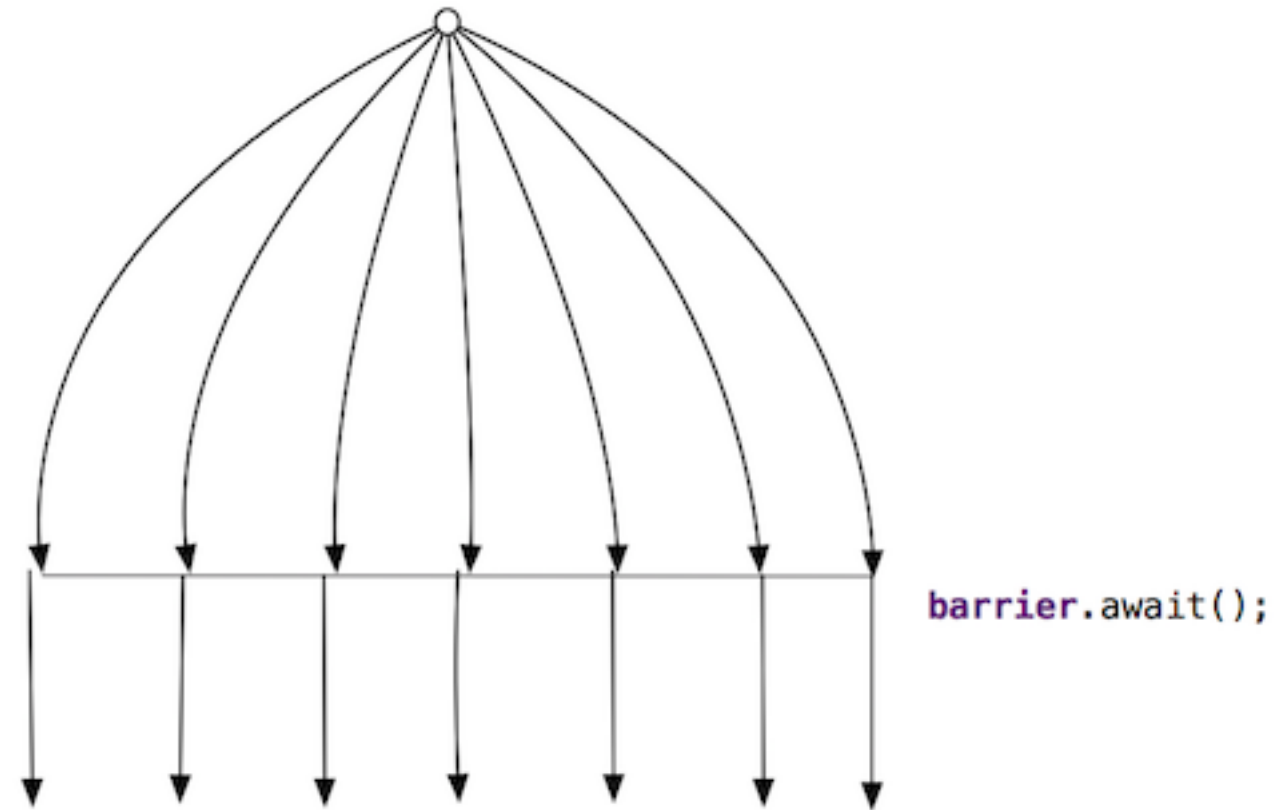
```
CountDownLatch startSignal = new
CountDownLatch(1);
CountDownLatch doneSignal = new
CountDownLatch(N);
```

```
public void run() {
 try {
 startSignal.await();
 doWork();
 doneSignal.countDown();
 } catch (InterruptedException
ex) {} // return;
}
```

CountDownLatch signal = new CountDownLatch(7)

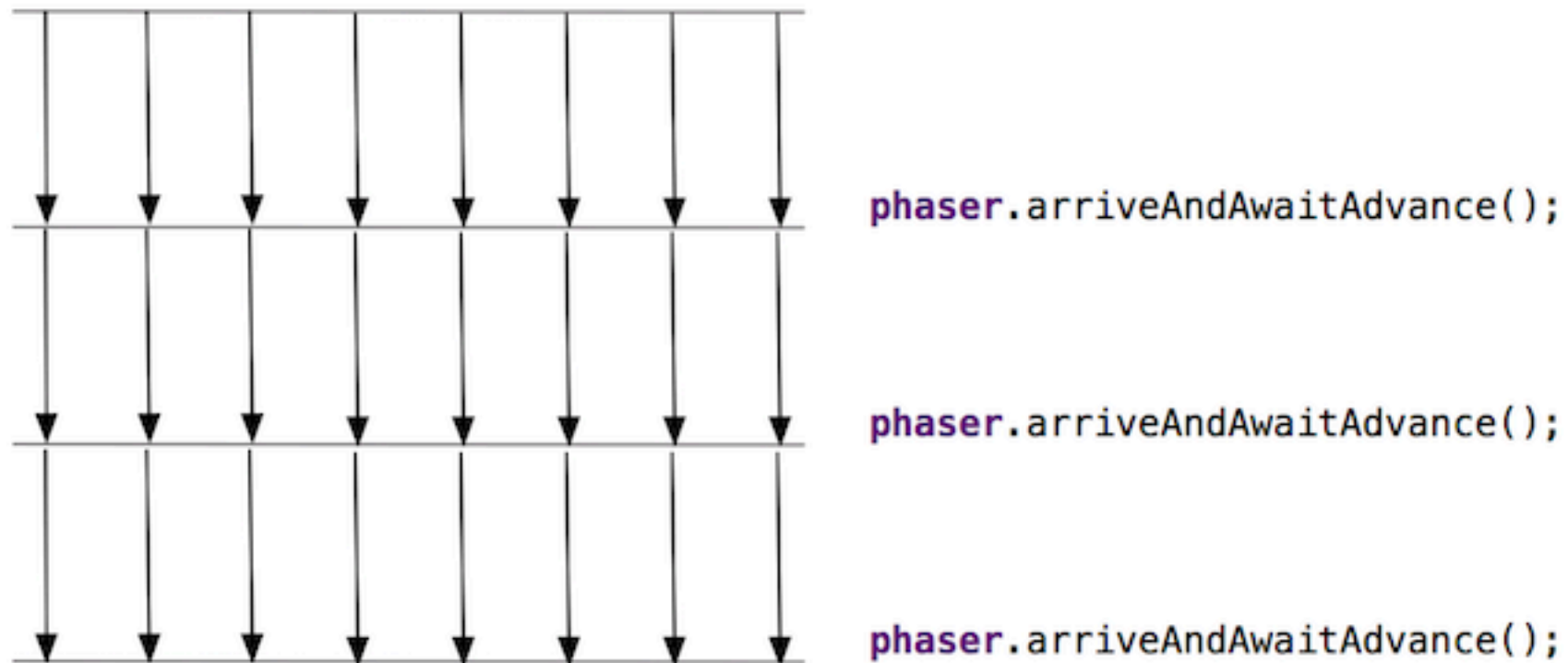


# CyclicBarrier



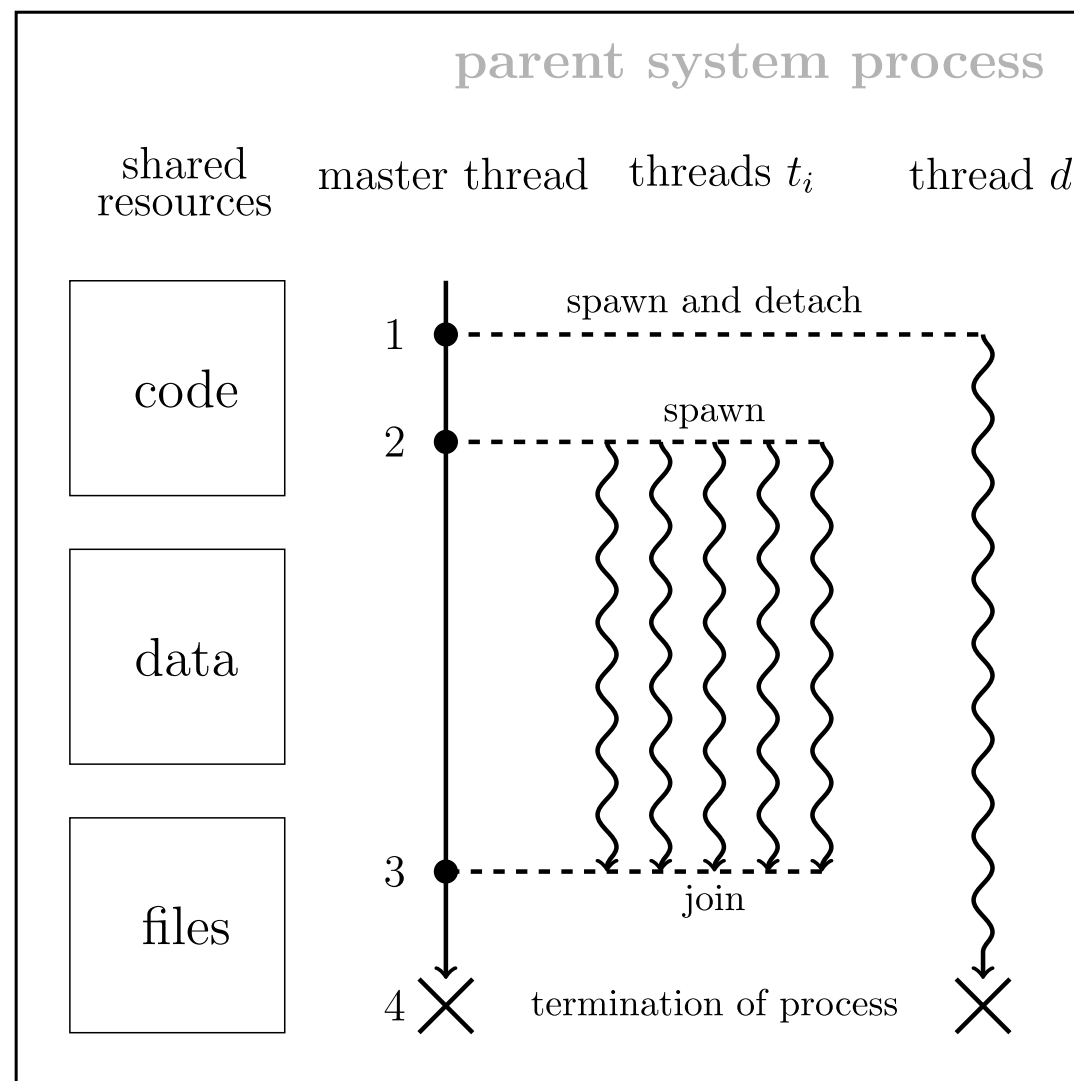
# Phaser

- A reusable synchronization barrier



# 并行编程

- 多线程
- OpenMP
- CUDA



# MULTITHREADING vs MULTIPROCESSING

- Multiprocessing parallelizes a program over multiple compute units (e.g. CPU cores) in order to exploit redundant resources such as registers and arithmetic logic units (ALUs) of different CPU cores to speed up computation.
- Multithreading shares the hardware resources such as caches and RAM of a single core or multiple cores in order to avoid idling of unused resources.



```

#include <iostream>
#include <cstdint>
#include <vector>
#include <thread>

// this function will be called by the threads (should be void)
void say_hello(uint64_t id) {
 std::cout << "Hello from thread: " << id << std::endl;
}

// this runs in the master thread
int main(int argc, char * argv[]) {
 const uint64_t num_threads = 4;
 std::vector<std::thread> threads;

 // for all threads
 for (uint64_t id = 0; id < num_threads; id++)
 // enqueue the thread object in vector threads
 // using argument forwarding, this avoids unnecessary
 // move operations to the vector after thread creation
 // threads.push_back(std::thread(say_hello, id));
 threads.emplace_back(
 // call say_hello with argument id
 say_hello, id
);
 // join each thread at the end
 for (auto& thread: threads)
 thread.join();
}

```

- g++ 4.1.cpp -std=c++11 -pthread

# FIBONACCI SEQUENCE

```
template <
 typename value_t,
 typename index_t>
void fibo(
 value_t n,
 value_t * result) {

 value_t a_0 = 0;
 value_t a_1 = 1;

 for (index_t index = 0; index < n; index++) {
 const value_t tmp = a_0; a_0 = a_1; a_1 += tmp;
 }

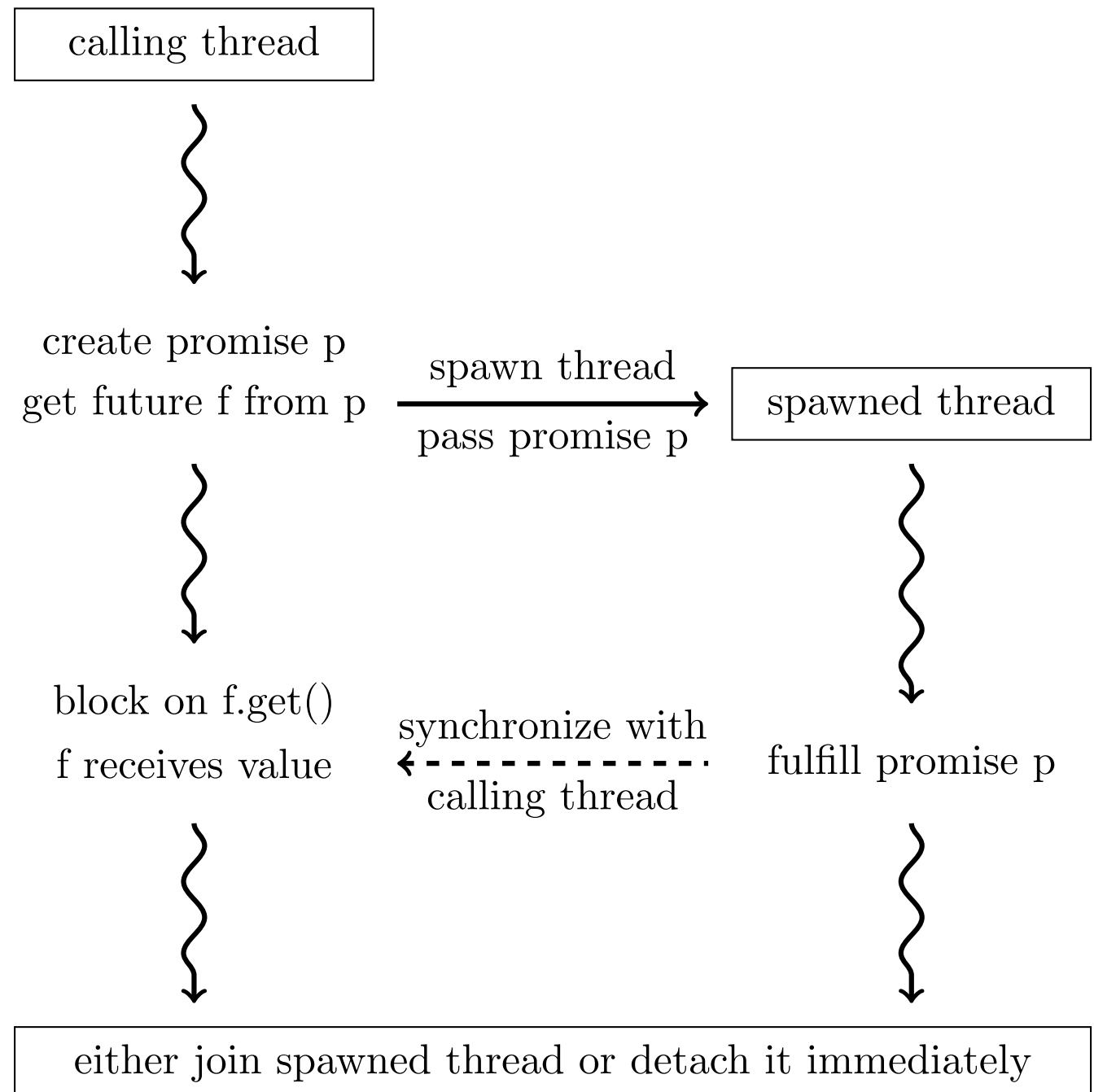
 *result = a_0;
}
```

**return\_values/traditional.cpp**

# PROMISES AND FUTURES

- promises that are fulfilled in the future
- a pair of tied objects  $s = (p, f)$  where  $p$  is a writable view of the state  $s$ , the promise, which can be set to a specific value

**promise\_future.cpp**



# ASYNCHRONOUS WAY

- The command `std::async` executes a task asynchronously using either dedicated threads or the calling master thread. Task creation and assignment of the corresponding future is as simple as

```
auto future = std::async(fibo, id);
```

- Unfortunately, the command `std::async` has to be taken with a grain of salt since its behavior can be vastly unintuitive despite its simple syntax. Let us briefly enumerate severe pitfalls related to its use:
  1. A naive call to `std::async` does not necessarily imply that a novel thread is spawned. The execution might be performed by the calling thread.
  2. The execution of the task could be delayed forever if we do not access the corresponding future via `future.get()`.
  3. The execution of distinct tasks could be serialized if we do not pay attention to the scopes of the corresponding futures.

# Lazy

- `std::launch::async` spawns a thread and immediately executes the task.  
`auto future = std::async(std::launch::async, fibo, id);`
- `std::launch::deferred` executes the task in a lazy evaluation fashion on the calling (same) thread at the first time the future is accessed with `get()`.  
`auto future = std::async(std::launch::deferred, fibo, id);`

# OpenMP

- OpenMP is an application programming interface (API) for platform-independent shared-memory parallel programming in C, C++, and Fortran. It pursues a semi-automatic parallelization approach which allows for the augmentation of sequential code using simple compiler directives in contrast to the explicit programming model of C++11 multithreading which involves the manual spawning, joining, and synchronization of threads.
- Complex schedules, task parallelism, the convenient parallelization of loops and reductions can be realized with just a few additional lines of code. Despite its simplicity, the parallelization efficiency of OpenMP-augmented CPU code is often as good as handcrafted solutions using C++11 threads.

# Brief History

- The OpenMP API specifications are produced and published by the OpenMP Architecture Review Board (ARB), which consists of a group of major computer hardware and software vendors, as well as major parallel computing user facilities. Historically, the first specification released was OpenMP for Fortran 1.0 in October 1997, followed by OpenMP for C and C++ 1.0 in October 1998. Significant milestones for the C and C++ version are version 2.0 in March 2002, version 2.5 in May 2005, version 3.0 in May 2008, version 3.1 in July 2011, version 4.0 in July 2013, and version 4.5 from November 2015 [2,3]. Specifications of corresponding releases are publicly available on the respective website of the OpenMP ARB: <http://www.openmp.org/specifications>.

# Basic of OMP

- The basic philosophy of OpenMP is to augment sequential code by using special comment-like compiler directives – so-called pragmas – to give hints to the compiler on how the code can be parallelized
- Whether the compiler should include support for the OpenMP API is specified by the compiler flag `-fopenmp`. If we compile the code without this flag, we end up with sequential code. This behavior allows for the convenient deployment of sequential code that can be augmented with optional support for parallel execution.



# Hello

```
#include <iostream>

int main() {
 // run the statement after the pragma in the current team
 #pragma omp parallel
 std::cout << "Hello world!" << std::endl;
}
```

- `g++ -O2 -std=c++14 -fopenmp hello_world.cpp -o hello_world`
- You may set the environment variable `OMP_NUM_THREADS` before running the program in order to control the number of threads used

# parallel for

```
#pragma omp parallel for
for (...) {
 ...
}
```

```
#pragma omp parallel
{
 #pragma omp for
 for (...) {
 ...
 }
}
```

- to execute a for-loop without data dependencies in parallel
- The directive `#pragma omp for` acts as splitting primitive that partitions the set of  $n$  indices into chunks of approximate size  $n/p$  where  $p$  is the number of utilized threads
- As a result, we are not allowed to manipulate the iterator index or its bounds in the body of the for-loop

# VECTOR ADDITION

```
#pragma omp parallel for
for (uint64_t i = 0; i < num_entries; i++) {
 x[i] = i;
 y[i] = num_entries-i;
}
```

- vector\_add.cpp

# Implicit Synchronization

```
#pragma omp parallel
{ // <- spawning of threads

 #pragma omp for
 for (uint64_t i = 0; i < num_entries; i++) {
 x[i] = i;
 y[i] = num_entries - i;
 }

 // <- implicit barrier

 #pragma omp for
 for (uint64_t i = 0; i < num_entries; i++)
 z[i] = x[i] + y[i];

 // <- another implicit barrier

 #pragma omp for
 for (uint64_t i = 0; i < num_entries; i++)
 if (z[i] - num_entries)
 std::cout << "error at position "
 << i << std::endl;
} // <- joining of threads

// <- final barrier of the parallel scope
```

```
#pragma omp parallel
{
 #pragma omp for nowait
 for (...) { ... }

 // <- here is no barrier

 #pragma omp for
 for (...) { ... }
}
```

# VARIABLE SHARING AND PRIVATIZATION

```
int main () {

 int i, j; // must be declared before the loops

 // #pragma omp parallel for produces incorrect
 // results due to race condition on variable j
 for (i = 0; i < 4; i++)
 for (j = 0; j < 4; j++)
 printf("%d %d\n", i, j);
}
```

```
int main () {

 int i = 1;

 // each thread declares its own i
 // but leaves it uninitialized
 #pragma omp parallel private(i)
 {
 // WARNING: i is not initialized here!
 printf("%d\n", i); // could be anything
 }
}
```

```
int main () {

 int i = 1;

 // each thread declares its own i
 // and sets it to i = 1
 #pragma omp parallel firstprivate(i)
 {
 printf("%d\n", i); // i == 1
 }
}
```

# Capturing Private Variables

```
int main () {

 // maximum number of threads and auxiliary memory
 const int num = omp_get_max_threads();
 int * aux = new int[num];

 int i = 1; // we pass this via copy by value

 #pragma omp parallel firstprivate(i) num_threads(num)
 {
 // get the thread identifier j
 const int j = omp_get_thread_num();
 i += j; // any arbitrary function f(i, j)
 aux[j] = i; // write i back to global scope
 }

 delete [] aux; // aux stores the values [1, 2, 3, ...]
```

