Logistics

- Homework
  - Homework #1 answers posted
  - Homework #2 posted
    - Due Tuesday, May 3, 5pm

- Term projects
  - Decided to have a class project
  - Will discuss next week

- Term papers (graduate students)

- OpenACC is the programming lab this week
Contents

- What is the fork-join concept?
- What is the fork-join pattern?
- Programming Model Support for Fork-Join
- Recursive Implementation of Map
- Choosing Base Cases
- Load Balancing
- Cache Locality and Cache-Oblivious Algorithms
- Implementing Scan with Fork-Join
- Applying Fork-Join to Recurrences
Fork-Join Philosophy

*When you come to a fork in the road, take it.*

(Yogi Bera, 1925 – 2014)
Fork-Join Concept

- Fork-Join is a fundamental way (primitive) of expressing concurrency within a computation.

- **Fork** is called by a (logical) thread (*parent*) to create a new (logical) thread (*child*) of concurrency.
  - Parent continues after the *Fork* operation.
  - Child begins operation separate from the parent.
  - *Fork* creates concurrency.

- **Join** is called by both the parent and child.
  - Child calls *Join* after it finishes (implicitly on exit).
  - Parent waits until child joins (continues afterwards).
  - *Join* removes concurrency because child exits.
Fork-Join Concurrency Semantics

- Fork-Join is a *concurrency control* mechanism
  - Fork increases concurrency
  - Join decreases concurrency

- Fork-Join dependency rules
  - A parent must join with (only) its forked children
  - Forked children with the same parent can join with the parent in any order
  - A child can not join with its parent until it has joined with all of its children

- Fork-Join creates a special type of DAG
  - What do they look like?
  - What constraints do they have?
Fork Operation

- Fork creates a child thread
- What does the child do?
  - Typically, fork operates by assigning the child thread with some piece of “work”
    - Child thread performs the piece of work and then exits by calling join with the parent
    - Child work is usually specified by providing the child with a function to call on startup
- Child typically inherits parent’s state
  - In particular, it inherits the address space
  - Assumes shared memory execution
Join Operation

- Join informs the parent that the child has finished
- Child thread notifies the parent and then exits
  - Might provide some status back to the parent
- Parent thread waits for the child thread to join
  - Continues after the child thread joins
- Joining involves synchronization
- Two scenarios
  1. Child joins first, then parent joins with no waiting
  2. Parent joins first and waits, child joins and parent then continues
Fork-Join Heritage in Unix

- Fork-Join comes from basic forms of creating processes and threads in operating system
- Forking a child process from a parent process
  - Creates a new child process with `fork()`
  - *Process state* of parent is copied to child process
    - process ID of parent stored in child process state
    - process ID of child stored in parent process state
  - Parent process continues to next PC on `fork()` return
  - Child process starts execution at next PC
    - process ID is automatically set to child process
    - child can call `exec()` to overlay another program, including getting a new address space
Fork-Join Heritage in Unix (2)

- Joining a child process with a parent process
  - Child process exits and parent process is notified
    - if parent is blocked waiting, it unblocks
    - if parent is not waiting, some indication is made that the child process exited, so that the parent will see this
    - child process effectively joins
  - Parent process calls `waitpid()` (effectively join) for a particular child process
    - if the child process has called `join()`, parent continues
    - if the child process has not called `join()`, parent blocks

- Fork-Join also implemented for threads
#include <sys/types.h> /* pid_t */
#include <sys/wait.h> /* waitpid */
#include <stdio.h> /* printf, perror */
#include <stdlib.h> /* exit */
#include <unistd.h> /* _exit, fork */

int main(void)
{
    pid_t pid;
    pid = fork();

    if (pid == -1) {
        /* When fork() returns -1, an error happened. */
        perror("fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        /* When fork() returns 0, we are in the child process. */
        printf("Hello from the child process!\n");
        _exit(EXIT_SUCCESS); /* exit() is unreliable here, so _exit must be used */
    } else {
        /* When fork() returns a positive number, we are in the parent process
         * and the return value is the PID of the newly created child process. */
        int status;
        (void)waitpid(pid, &status, 0);
    }
    return EXIT_SUCCESS;
}
Fork-Join in POSIX Thread Programming

- POSIX standard multi-threading interface
  - For general multi-threaded concurrent programming
  - Largely independent across implementations
  - Broadly supported on different platforms
  - Common target for parallel library and language implementations
    - still can program parallel applications with Pthreads directly if you want, but it is low-level and challenging

- Provides primitives for:
  - Thread creation and management
  - Synchronization
Thread Creation

```c
#include <pthread.h>
int pthread_create(
    pthread_t *thread_id,
    const pthread_attr_t *attribute,
    void *(*thread_function)(void *),
    void *arg
);
```

- `thread_id`
  - thread’s unique identifier
- `attribute`
  - contain details on scheduling policy, priority, stack, ...
- `thread_function`
  - function to be run in parallel (entry point)
- `arg`
  - arguments for function `func`
Example of Thread Creation

```c
void *func(void *arg) {
    ...
}

void main() {
    int X;
    pthread_t id;
    ...
    pthread_create(&id, NULL, func, &X);
    ...
}
```
void pthread_exit(void *status)

- Terminates the currently running thread
- Implicitly called when function called in
  pthread_create returns
Thread Joining

```c
int pthread_join(
    pthread_t thread_id,
    void **status);
```

- Waits for thread `thread_id` to terminate
  - Either implicitly by returning
  - Or by calling `pthread_exit()`
- Status receives the return value or the value given as argument to `pthread_exit()`
Thread Joining Example

```c
void *func(void *arg)
{
    ...
    pthread_exit();
}

void main()
{
    int X;
    pthread_t id;
    ...
    pthread_create(&id, NULL, func, &X);
    ...
    pthread_join(id, NULL);
    ...
}
```
General Program Structure

- Encapsulate parallel parts in functions
- Use function arguments to parameterize thread behavior (functionality)
- Call `pthread_create()` with the function
- Call `pthread_join()` for each thread created
- Need to take care to make program “thread safe”
  - Avoid using shared global variables for thread state
Pthread Process Management

- `pthread_create()`
  - Creates a parallel thread executing a given function
  - Passes function arguments
  - Returns thread identifier

- `pthread_exit()`
  - Terminates thread

- `pthread_join()`
  - Waits for particular thread to terminate
Pthreads Synchronization

- Create/exit/join
  - Provide some coarse form of synchronization
  - “Fork-join” parallelism
  - Requires thread creation/destruction

- Need for finer-grain synchronization
  - Mutex locks
  - Condition variables
Pthreads “Hello World”

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS 5

void *TaskCode(void *argument)
{
    int tid;
    tid = *((int *) argument);
    printf("Hello World! It's me, thread \%d\n", tid);
    /* optionally: insert more useful stuff here */
    return NULL;
}
```
int main(void)
{
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int rc, i;

    /* create all threads */
    for (i=0; i<NUM_THREADS; ++i) {
        thread_args[i] = i;
        printf("In main: creating thread %d\n", i);
        rc = pthread_create(&threads[i], NULL, TaskCode, (void *)&thread_args[i]);
        assert(0 == rc);
    }

    /* wait for all threads to complete */
    for (i=0; i<NUM_THREADS; ++i) {
        rc = pthread_join(threads[i], NULL);
        assert(0 == rc);
    }

    exit(EXIT_SUCCESS);
}
Fork-Join Pattern

- Control flow divides (forks) into multiple flows, then combines (joins) later
- During a fork, one flow of control becomes two
- Separate flows are “independent”
  - Does “independent” mean “not dependent”?
    - No, it just means that the 2 flows of control “are not constrained in their computation” (i.e., they are assumed to execute independently of each other)
- During a join, two flows become one, and only this one flow continues
Fork-Join Pattern

- Fork-Join directed graph:

Is it possible for B() and C() to have dependencies between them? Why?
Fork-Join Pattern

- Typical **divide-and-conquer** algorithm implemented with fork-join:

```c
void DivideAndConquer( Problem P ) {
    if( P is base case ) {
        Solve P;
    } else {
        Divide P into K subproblems;
        Fork to conquer each subproblem in parallel;
        Join;
        Combine subsolutions into final solution;
    }
}
```
Fork-Join Pattern for Divide-Conquer
Fork-Join Pattern for Divide-Conquer (2)

\[ K = 2 \]
(2-way fork-join)

\[ N = 3 \]
(3 levels of join)
Fork-Join Pattern for Divide-Conquer

$2^3 = 8$-way parallelism
Fork-Join Pattern

- Selecting the base case size is critical
- Recursion must go deep enough for plenty of parallelism
- Too deep, and the granularity of sub-problems will be dominated by scheduling overhead
- With $K$-way fork-join and $N$ levels of fork-join, can have up to $K^N$-way parallelism
Fibonacci Example

- Recursive Fibonacci is simple and inefficient

```c
long fib ( int n ) {
    if (n < 2) return 1;
    else {
        long x = fib(n-1);
        long y = fib(n-2);
        return x + y;
    }
}
```
Fibonacci Example

- Recursive Fibonacci is simple and inefficient
- Are there dependencies between the sub-calls?
- Can we parallelize it?
Fibonacci in Parallel Example

```c
long fib ( int n ) {
  if (n < 2) return 1;
  else {
    long x = fork fib (n-1);
    long y = fib(n-2);
    join;
    return x + y;
  }
}
```
Programming Model Support for Fork-Join

- Cilk Plus:
  
  ```cilk
  cilk_spawn B();  // Fork
  C();
  cilk_sync;      // Join
  ```

- B() executes in the child thread
- C() executes in the parent thread
Programming Model Support for Fork-Join

- Cilk Plus:

```cilk
    cilk_spawn B();
    C();
    cilk_sync;
```

```cilk
    cilk_spawn A();
    cilk_spawn B();
    cilk_spawn C();
    D(); // Not spawned, executed in spawning task
    cilk_sync; // Join
```

```cilk
    for ( int i=0; i<n; ++i )
        if ( a[i]!=0 )
            cilk_spawn f(a[i]);
    cilk_sync;
```

Good form!
Cilk Plus:

cilk_spawn B();
cilk_spawn C();
/* nil */
cilk_sync;

Bad form! Why?
Programming Model Support for Fork-Join

- **TBB**
  - `parallel_invoke()`
    - For 2 to 10 way fork
    - Joins all tasks before returning
  - `Tbb::task_group`
    - For more complicated cases
    - Provides explicit join

```c++
task_group g;
for (int i=0; i<n; ++i)
  if (a[i] != 0)
    g.run([=, &a]{f(a[i]);}); // Spawn f(a[i]) as child task
  g.wait(); // Wait for all tasks spawned from g
```
Programming Model Support for Fork-Join

- OpenMP:

```plaintext
#pragma omp task
B();  # Forked task
C();  # Performed by spawning task
#pragma omp taskwait
```


OpenMP:

```c
#pragma omp task
B();
C();
#pragma omp taskwait
```

Forked task can also be a compound statement:

```c
{B(); C(); D();}
```
Programming Model Support for Fork-Join

- OpenMP:

```c
#pragma omp task
B();
C();
#pragma omp taskwait
```

Must be enclosed in an OpenMP parallel construct … Why?
More on the 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**
  
  ```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++**
  
  ```c
  #pragma omp parallel for [clause [clause ...]]
  for (index = first; text_expr; increment_expr) {
      body of the loop
  }
  ```
Example: PARALLEL DO

- 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
    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 $a*x + y$ (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
   y(i) = a * x(i+1) + y(i+1)
enddo
return
end

What happens here?
How Many Threads?

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

- Use `omp_set_num_threads()` function

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

```
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

```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
```

CIS 410/510: Parallel Computing, University of Oregon, Spring 2016
PARALLEL Directive

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

- **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

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

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

Is this ok?
Parallel versus Parallel Do

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

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

!$omp parallel do
output: 10 Hello world messages
```

```
!$omp parallel
output: 10*T Hello world messages
```

PARALLEL DO is a work sharing directive

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 1 + 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])
}
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

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

!$omp parallel do
do I = 1,10
   print *, 'Hello world', I
enddo
!$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
Recursive Implementation of Map

- Map is a simple, useful pattern that fork-join can implement
- Good to know how to implement map with fork-join if you ever need to write your own map with novel features (fusing map with other patterns)
- Cilk Plus and TBB implement their map constructs with a similar divide-and-conquer algorithm
Recursive Implementation of Map (2)

cilk_for( unsigned i=lower; i<upper; ++i )
f(i);

cilk_for can be implemented with a divide-and-conquer routine...

if( lower<upper ) recursive_map(lower,upper,grainsize,f)
Recursive Implementation of Map (3)

```cpp
template<typename Func>
void recursive_map( unsigned lower, unsigned upper, unsigned grainsize, Func f ) {
  if( upper-lower <= grainsize )
    // Parallel base case
    for( unsigned i=lower; i<upper; ++i )
      f(i);
  else {
    // Divide and conquer
    unsigned middle = lower+(upper-lower)/2u;
    cilk_spawn recursive_map( lower, middle, grainsize, f );
    recursive_map( middle, upper, grainsize, f );
  }
  // Implicit cilk_sync when function returns
}
```
Recursive Implementation of Map (4)

```python
recursive_map(0, 9, 2, f)
```
Choosing Base Cases

- For parallel divide-and-conquer, two base cases:
  - Stopping parallel recursion
  - Stopping serial recursion

- For a machine with $P$ hardware threads, we might think to have $P$ leaves in the spawned functions tree

- This often leads to poor performance
  - Scheduler has no flexibility to balance load
Choosing Base Cases (2)

- Given leaves from spawned function tree with equal work, and equivalent processors, system effects can effect load balance:
  - Page faults
  - Cache misses
  - Interrupts
  - I/O

- Best to **over-decompose** a problem

- This creates **parallel slack**
Over Decomposition and Parallel Slack

- **Over-decompose**: parallel programming style where more tasks are specified than there are physical workers. Beneficial in load balancing.

- **Parallel slack**: Amount of extra parallelism available above the minimum necessary to use the parallel hardware resources.
Load Balancing

- Sometimes, threads will finish their work at different rates.
- When this happens, some threads may have nothing to do while others may have a lot of work to do.
- This is known as a **load balancing** issue.
Load Balancing

- Cilk Plus and TBB use work stealing to automatically balance fork-join work
- In a work-stealing scheduler, each thread is a worker
- Each worker maintains a stack of tasks
- When a worker’s stack is empty, it grabs from the bottom of another random worker
  - Tasks at the bottom of a stack are from the beginning of the call tree – tend to be a bigger piece of work
  - Stolen work will be distant from stack’s owner, minimizing cache conflicts
Load Balancing

- Cilk Plus and TBB work-stealing differences:

Cilk Plus

TBB
Performance of Fork/Join

Let $A\parallel B$ be interpreted as “fork $A$, do $B$, and join”

Work: $T(A\parallel B)_1 = T(A)_1 + T(B)_1$

Span: $T(A\parallel B)_\infty = \max(T(A)_\infty, T(B)_\infty)$
Cache Locality / Cache-Oblivious Algorithms

- Work/Span analysis ignores memory bandwidth constraints that often limit speedup
- Cache reuse is important when memory bandwidth is critical resource
- Tailoring algorithms to optimize cache reuse is difficult to achieve across machines
- Cache-oblivious programming is a solution for this
- Code is written to work well regardless of cache structure
Cache-Oblivious Programming

- Cache-oblivious programming strategy:
  - Recursive divide-and-conquer – good data locality at multiple scales
  - When a problem is subdivided enough, it can fit into the largest cache level
  - Continue subdividing to fit data into smaller and faster cache

- Example problem: matrix multiplication
  - Typical, non-recursive, algorithm uses three nested loops to implement
  - Large matrices will not fit in cache with this approach
Implementing Scan with Fork-Join

- We saw that the map pattern can be implemented with the fork-join pattern.
- Now we will examine how to implement the scan operation with fork-join.
- Input: initial value (\textit{initial}) and sequence \((r_0, r_1, \ldots, r_n)\)
- Output: exclusive scan sequence \((s_0, s_1, \ldots, s_n)\)
- Upsweep computes a set of partial reductions on tiles of data.
- Downsweep computes final scan by combining partial reductions.
Implementing Scan with Fork-Join (2)
Implementing Scan with Fork-Join (3)
During the upsweep, each node computes a partial reduction of the form:

\[ r_{i:m} = r_{j:k} + r_{i+k:m-k} \]
During the downsweep, each node computes a partial scan of the form:

\[ s_i = \text{initial} + r_{0:i} \quad \text{and} \]

\[ s_{i+k} = s_i + r_{i:k} \]