System Software Support for Parallel Programming

I-Ting Angelina Lee

CSE 591, Fall 2018
What Is Parallel Programming?

- Divide up your computation into multiple components that can be worked on in parallel …
- So that you can simultaneously use multiple compute resources to solve the computational problem.
Different Types of Parallel Programming

Supercomputer: multiple computing nodes connected with high-bandwidth network

Program it using MPI (Message Passing Interface)
Different Types of Parallel Programming

Within a single node: multiple cores with shared memory

Traditional paradigm: persistent threads
Why Parallel Programming?

- Performance!

Problem: Parallel programming is hard
Challenges in Programming a Multicore Machine

- Scheduling: What executes when and on which core
- Synchronization: How to coordinate accesses to shared resources
- Locality: How to effectively use caches and proximity of cores
Challenges in Programming a Multicore Machine

Scheduling

Synchronization

Locality

Traditional paradigm (pthreads) does not address these challenges well.
My Research Goal

Make parallel programming on commodity multicore hardware accessible for everyone.
Challenges in Programming a Multicore Machine

Traditional paradigm (pthreads) does not address these challenges well.
Example Application: Dedup*

Dedup compresses a stream of data by compressing unique elements and removing duplicates.

```c
int fd_out = open_output_file();
bool done = false;
while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        chunk->is_dup = deduplicate(chunk);
        if(!chunk->is_dup) compress(chunk);
        write_to_file(fd_out, chunk);
    }
}
```

*Extrapolated from the PARSEC benchmark [BKS08]

**Stage 0:** While there is more data, read the next chunk from the stream.

**Stage 1:** Check for duplicates.

**Stage 2:** Compress first-seen chunk.

**Stage 3:** Write to output file.
Pipeline Parallelism in Dedup

while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        chunk->is_dup = deduplicate(chunk);
        if(!chunk->is_dup) compress(chunk);
        write_to_file(fd_out, chunk);
    }
}

Let’s model Dedup’s execution as a pipeline dag.

- A node denotes the execution of a stage in an iteration.
- Edges denote dependencies between nodes.

Dedup exhibits pipeline parallelism.
Parallelizing Dedup with Pthreads

while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }  
    else {
        chunk->is_dup = deduplicate(chunk);
        if(!chunk->is_dup) compress(chunk);
        write_to_file(fd_out, chunk);
    }
}

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
(The programmer writes the scheduling code.)
Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.

![Diagram](image_url)

- Get_Input
- Deduplicate
- Compress
- Hashtable
- Write_Output

stage  thread  concurrent queue
Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.

To load balance better (scheduling), add multiple threads to heavy stages.

Need a concurrent hashtable (synchronization).
Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.

Chunks are processed out of order.
The cross-edge dependencies are violated (synchronization).
Parallelizing Dedup with Pthreads

1. Assign threads to stages.
2. Threads communicate via concurrent queues.
3. Execute.

Chunks are processed out of order.
The cross-edge dependencies are violated (synchronization).
Sort the output before we write it out.
Parallelizing Dedup with Pthreads

1. Assign threads to stages.

2. Threads communicate via concurrent queues.

3. Execute.

Threads contend on queues (synchronization).

Add more queues.
Parallelizing Dedup with Pthreads

1. Assign threads to stages.

2. Threads communicate via concurrent queues.

3. Execute.

Deadlock!

Threads in the compress stage are not getting enough cycles (scheduling).

Limit queue size.
Parallelizing Dedup with Pthreads*

1. Assign threads to stages.

2. Threads communicate via concurrent queues.

3. Execute.

Threads in the compress stage are not getting enough cycles (scheduling).

Limit queue size.

* Based on the parallel implementation in PARSEC [BKS08].
The programmer must manually handle scheduling and synchronization.

The setup code for parallel execution using pthreads.
The programmer must manually manage scheduling and synchronization.

- Scheduling logic intermixed with program logic $\Rightarrow$ spaghetti code.
- Threads interact via shared memory $\Rightarrow$ no well-defined ordering of events.
- The scheduling logic interacts with the need for synchronization.
Structured Parallel Programming

A programming model that allows the programmer to express the logical parallelism of the computation to using control constructs.

- separates the scheduling logic from program logic;
- automates scheduling and synchronization; and
- provides a clean mental model for the programmer to reason about parallelism.
Traditional Computing Stack

- tools
- user application
- compiler
- operating system
- hardware

Provides the pthread abstraction as surrogates for cores
State of Art: Concurrency Platform

A concurrency platform should provide:

- an interface for specifying the *logical parallelism* of the computation;
- a runtime layer to automate scheduling and synchronization; and
- guarantees of performance and resource utilization competitive with hand-tuned code.
My Research

- Design language abstractions for structured parallel programming
- Develop efficient system support for these language abstractions
- Design tool support for debugging and performance engineering programs written in these high-level language abstractions
Cilk-P’s Linguistic Support for Pipeline Parallelism

An instance of structured parallel programming
Encode Parallelism of Dedup

while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; } else {
        chunk->is_dup = deduplicate(chunk);
        if(!chunk->is_dup) compress(chunk);
        write_to_file(fd_out, chunk);
    }
}

1. Pipeline the loop.

   Stage 0

   Stage 1

   Stage 2

   Stage 3

   Cross Edge
Encode Parallelism of Dedup

```c
pipe_while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        chunk->is_dup = deduplicate(chunk);
        if(!chunk->is_dup) compress(chunk);
        write_to_file(fd_out, chunk);
    }
}
```

1. Pipeline the loop.
2. Denote stages.

: Cross Edge
Encode Parallelism of Dedup

```c
pipe_while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        pipe_stage;
        chunk->is_dup = deduplicate(chunk);
        pipe_stage;
        if(!chunk->is_dup) compress(chunk);
        pipe_stage;
        write_to_file(fd_out, chunk);
    }
}
```

Stage 0

Stage 1

Stage 2

Stage 3

Iterations

1. Pipeline the loop.
2. Denote stages.
3. Enforce cross-edge dependencies

→ Cross Edge
Encode Parallelism of Dedup

```c
pipe_while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        pipe_stage_wait();
        chunk->is_dup = deduplicate(chunk);
        pipe_stage();
        if(!chunk->is_dup) compress(chunk);
        pipe_stage_wait();
        write_to_file(fd_out, chunk);
    }
}
```

1. Pipeline the loop.
2. Denote stages.
3. Enforce cross-edge dependencies
The Pipeline Linguistics in Cilk-P

```c
int fd_out = open_output_file();
bool done = false;
pipe_while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        pipe_stage_wait(1);
        chunk->is_dup = deduplicate(chunk);
        pipe_stage(2);
        if(!chunk->is_dup) { compress(chunk); }
        pipe_stage_wait(3);
        write_to_file(fd_out, chunk);
    }
}
```

Loop iterations may execute in parallel in a pipelined fashion, where stage 0 executes serially.

End the current stage, advance to stage 1, and wait for the previous iteration to finish stage 1.

End the current stage and advance to stage 2.
The Pipeline Linguistics in Cilk-P

```c
int fd_out = open_output_file();
bool done = false;
pipe_while(!done) {
    chunk_t *chunk = get_next_chunk();
    if(chunk == NULL) { done = true; }
    else {
        pipe_stage_wait(1);
        chunk->is_dup = deduplicate(chunk);
        pipe_stage(2);
        if(!chunk->is_dup) compress(chunk);
        pipe_stage_wait(3);
        write_to_file(fd_out, chunk);
    }
}

These keywords have serial semantics [FLR98].
```
The Pipeline Linguistics in Cilk-P

pipe_while(!done) {  
    chunk_t *chunk = get_next_chunk();  
    if(chunk == NULL) { done = true; }  
    else {  
        pipe_stage_wait(1);  
        chunk->is_dup = deduplicate(chunk);  
        pipe_stage(2);  
        if(!chunk->is_dup) compress(chunk);  
        pipe_stage_wait(3);  
        write_to_file(fd_out, chunk);  
    }  
}

These keywords allow the user to express the logical parallelism.
On-the-Fly Pipelining of X264

Cilk-P supports on-the-fly pipeline parallelism, where the pipeline is constructed dynamically as the program executes.

By enclosing `pipe_stage` and `pipe_stage_wait` statements within other control constructs, one can:

- skip stages;
- make cross edges data dependent; and
- vary the number of stages across iterations.
Piper: Cilk-P’s Provably-Efficient Scheduler

Elegant linguistic interface is only half the battle.
PIPER: A Work-Stealing Scheduler

A worker (surrogate for a processor) by default follows the *serial execution order*.

Each iteration is a "task."

: done  : not done
A worker (surrogate for a processor) by default follows the \textit{serial execution order}. 

Each iteration is a "task."
**PIPER: A Work-Stealing Scheduler**

A **worker** (surrogate for a processor) by default follows the *serial execution order*.

Each iteration is a "task."

- Serial semantics; and
- Don't need queues to pass elements between stages;
- Potentially better locality.

A worker (surrogate for a processor) by default follows the *serial execution order*. 
PIPER: A Work-Stealing Scheduler

A worker *steals* work from a randomly selected victim when it runs out of work to do.

Iteration is enabled by the blue worker.

Represented as:
- \( i_0 \): done
- \( i_1 \): not done

Steal!
PIPER: A Work-Stealing Scheduler

A worker **steals** work from a randomly selected victim when it runs out of work to do.

- **P**: worker
- **P**: work

: done  : not done
Performance Measures [CLRS09]

Let $T_p$ be the time it takes to execute this dag on $P$ processors.

**Work $T_1$** : The sum of the weights of the nodes in the dag. $T_1 = 733$

**Span $T_\infty$** : The length of a longest path in the dag. $T_\infty = 112$

**Parallelism $T_1 / T_\infty$** : The maximum possible speedup. $T_1 / T_\infty = 6.54$

**Work Law** : $T_p \geq T_1 / P$  
**Span Law** : $T_p \geq T_\infty$
PIPER's Guarantees

**Definition.** $T_P$ — execution time on $P$ processors
$T_1$ — work  $T_\infty$ — span  $T_1/T_\infty$ — parallelism

$S_P$ — stack space on $P$ processors
$S_1$ — stack space of a serial execution
$K$ — throttling limit  $f$ — maximum frame size
$D$ — depth of nested pipelines

- **Time bound:** $T_P \leq T_1/P + O(T_\infty + \lg P)$ expected time

  $\Rightarrow$ **linear speedup** when $P \ll T_1/T_\infty$ and $T_\infty > \lg P$

- **Space bound:** $S_P \leq P(S_1 + fDK)$
The Check-Next Overhead

![Diagram showing the check-next overhead with iterations and states labeled as done or not done.](image-url)
The Check-Next Overhead
The Check-Next Overhead

Iteration \(i_2\) gets suspended.
The Check-Next Overhead

The blue worker re-enables iteration $i_2$. 

$\begin{array}{c}
\text{Iterations} \\
\hline
i_0 & 1 & 5 & 9 & 13 & 17 & 21 & \ldots \\
1 & 2 & 6 & 10 & 14 & 18 & 22 & \ldots \\
3 & 7 & 11 & 15 & 19 & 23 & \ldots \\
4 & 8 & 12 & 16 & 20 & 24 & \ldots \\
\end{array}$

$P$ : done $P$ : not done
The Check-Next Overhead

The purple worker steals iteration $i_2$. 

The diagram shows iterations $i_0$ to $i_5$ with numbers 1 to 24. The purple worker is marked by the purple circles, and the check-mark symbol indicates whether an iteration is done or not.
The Check-Next Overhead

Iteration $i_2$ gets suspended again.

: done  : not done
The Check-Next Overhead

The blue worker re-enables iteration $i_2$ again.

Iteration $i_2$ can get suspended and re-enabled repeatedly.

⇒ The blue worker must check next to re-enable $i_2$ after every stage!
Optimization: Lazy Enabling

Idea:
Be *really really* lazy about the *check-next* operation.

- **Iterations**
  - $i_0$
  - $i_1$
  - $i_2$
  - $i_3$
  - $i_4$
  - $i_5$

- **Nodes**
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - 9
  - 10
  - 11
  - 12
  - 13
  - 14
  - 15
  - 16
  - 17
  - 18
  - 19
  - 20
  - 21
  - 22
  - 23
  - 24

- **States**
  - : done
  - : not done

- **Actions**
  - check $i_2$?
  - Steal!
Optimization: Lazy Enabling

**Idea:**
Be *really really lazy* about the check-next operation.

Punt the responsibility of checking next onto a thief stealing or until the worker reaches the end of its iteration.

*With ample parallelism, this cost does not effect the performance!*
Implementation and Evaluation

Goal: Be competitive with highly-tuned code
Dedup Performance Comparison

![Graph showing performance comparison between different thread libraries (Pthreads (Best), TBB, Cilk-P, Pthreads (Intuitive)) over varying number of processors (P)].
Dedup Performance Comparison

Measured parallelism for Cilk-P (and TBB)’s pipeline is merely 7.4.
The pthreaded implementation has more parallelism due to unordered stages.
Dedup Performance Using Pthreads

Different configuration (threads per stage) leads to different results.

You don't need to do any of this with Cilk-P!
Ferret Performance Comparison

Cilk-P matches the best hand-tuned pthreaded code, and incurs no performance penalty for using the more general on-the-fly pipeline instead of a construct-and-run pipeline.
X264 Performance Comparison

Cilk-P matches the performance of hand-tuned pthreaded code, and the application programmer does not need to use any locks and conditional variables.
Pipeline Parallelism in Cilk-P

An instance of structured parallel programming

Cilk-P features:

- expressive linguistics for pipeline parallelism that separates the scheduling logic from program logic;
- effectively automates scheduling and synchronization; and
- provides a clean mental model for the programmer to reason about parallelism.
Cilk-P Inherited Fork-Join Parallelism from Cilk

Cilk's fork-join parallelism [FLR98]:

```c
int cilk fib(int n) {
    if(n < 2) { return n; }
    int x = spawn fib(n-1);
    int y = spawn fib(n-2);
    sync;
    return (x + y);
}
```
Cilk-P: A Unified Model
My Research

- Design language abstractions for structured parallel programming
- Develop efficient system support for these language abstractions
- Design tool support for debugging and performance engineering programs written in these high-level language abstractions
Determinacy Race

A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

**Example**

```c
int x = 0;
parallel_for(int i=0, i<2, ++i) {
    x++;  // B
}
assert(x == 2);  // D
```

![Dependency graph](image)

```
int x = 0;
x++;  // B
assert(x == 2);
x++;  // C
```

dependency graph
Determinacy Race

A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

```c
int x = 0;
parallel_for(int i=0, i<2, ++i) {
    x++;
} assert(x == 2);
```

*x* can be either 1 or 2.
Why Determinacy Race?

In the absence of a determinacy race, a program executes in a deterministic fashion.

1. Nondeterminism makes reasoning about parallel programs challenging!¹

2. Parallel programming must be deterministic by default!²

3. Deterministic parallel algorithms can be fast!³

2. Parallel programming must be deterministic by default! HotPar, 2009.
3. Shared-memory parallelism can be simple, fast, and scalable, CMU 2015 (winner of the ACM Doctoral Dissertation Award).
Determinacy Race

- Two *logically parallel* strands access the same memory location, with at least one being a write.
- In the absence of a determinacy race, a dynamic multithreaded computation behaves deterministically.
On-the-fly Determinacy Race Detection

The tool detects races as the program executes.

Goals:
• Allow the program to execute in parallel
• Detect races efficiently (asymptotically optimal)
• Provide strong correctness guarantees: report a race if and only if a race exists for the given input
Components of On-the-fly Determinacy Race Detection

• Design data structures to maintain **series parallel relationships** that tell us if two nodes are logically in parallel.
• Maintain **access histories** that tell us which nodes accessed the memory location previously.
• **Challenge:** Have low overheads and should scale.
P-Racer

- Provably efficient and correct parallel on-the-fly race detector for both fork-join and pipeline parallelism

Open problem:
- Reduce overheads of access history and instrumentation.
- Generalize to programs with more complex structural properties.
- Generalize to programs with locks.
Issues with Locks

- Lock operations generate complex dependences, making it difficult to track SP-relationships efficiently.
- Races or not depending on the schedule of lock acquire / release.
PORRidge

- Provably efficient and scalable deterministic record and replayer for fork-join parallel programs that employ locks
  - encapsulate all nondeterminism in the runtime system!
  - Can record and replay on different number of threads

- Open problem:
  - Currently the tool only captures nondeterminism due to lock operations
  - Reduce overhead for logging (both space and time)
Questions?

Ask me anything!