Compiler-level Concurrency Support: OpenMP, Cilk

Chris Rossbach
Outline for Today

• Questions?
• Administrivia
  • Go go go!
• Agenda
  • Compiler supported parallelism/concurrency
  • OpenMP
  • Cilk
Faux Quiz Questions

• What is a loop-carried dependence?
• List some tradeoffs between manual and auto parallelization
• List some challenges that make auto-parallelization of C/C++ hard; do any of them go away with managed language support?
• How does spawn differ from spawn_next in Cilk? Why does the language need both?
• How does OpenMP deal with partitioning work across threads? Compare and constrain this with Cilk.
**Message Passing: Motivation**

- Threads have a *lot* of downsides:
  - Tuning parallelism for different environments
  - Load balancing/assignment brittle
  - Shared state requires locks →
    - Priority inversion
    - Deadlock
    - Incorrect synchronization
  - ...

- Message passing:
  - *Threads aren’t the problem, shared memory is*
  - *Restructure programming model to avoid communication through shared memory (and therefore locks)*
Compiler Parallelization: Motivation

• Threads have a *lot* of downsides:
  • Tuning parallelism for different environments
  • Load balancing/assignment brittle
  • Shared state requires locks →
    • Priority inversion
    • Deadlock
    • Incorrect synchronization
  • …

• Compiler Parallelization:  
  • *Threads and shared memory aren’t the problem, the PROGRAMMER is*  
  • *restructure programming model to get the compiler to write the tricky code*
A simple program

```c
int main() {
    int * data = malloc(10000 * sizeof(int));
    for(int i = 0; i < 10000; i++) {
        data[i] = data[i] * data[i];
    }
}
```

How can we parallelize this?

Could a compiler parallelize this? If so, how? If not, why not?
How can we parallelize this one?

```c
int main() {
    int * data = ...
    for(int i = 1; i < 10000; i++) {
        data[i] = data[i] * data[i-1];
    }
}
```

Could a compiler tell the difference?
Another simple program

```c
int main() {
    int * data = ...
    int * temp = ...
    int * result = ...
    for(int i = 0; i < 10000; i++) {
        temp[i] = pipeline_stage1(data[i]);
    }
    for(int i = 0; i < 10000; i++) {
        result[i] = pipeline_stage2(temp[i]);
    }
}
```

Multiple forms of parallelism—both very simple and compiler-accessible
What about this one?

```c
int fib(int n) {
    if(n == 0 || n == 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main() {
    return fib(1000000);
}
```

Hopeless?
Auto- and Guided Compiler parallelization

• Totally do-able, *sometimes*

• Wide range of approaches:
  • partial/guided
  • Restricted programming model
  • Fully automatic
  • We’re going to see a lot of variants later in the semester: *today guided*

• Core: compiler looks for parallel idioms
  • Runs static analyses to decide safety
  • Not always guaranteed to be correct/performant

• Challenges same as for human
  • Decomposition/partitioning
  • Synchronization/Communication
  • Identifying *Dependences*
Data Dependence

• Three types of data dependence:

1. Flow (True) dependence: read-after-write
   ```c
   int a, b, c;
   a = c * 10;
   b = 2 * a + c;
   ```

2. Anti Dependency: write-after-read
   ```c
   int a, b, c;
   a = b* 4+ c;
   c = b + 40;
   ```

3. Output Dependence: write-after-write
   ```c
   int a, b, c;
   a = b *c ;
   a = b + c + 10;
   ```

Loop dependency analysis
• Compiler detects loops that can be safely and efficiently executed in parallel
• To know whether usages of an array access the same memory location, compiler performs dependency tests: dataflow analysis
Dependency in Loops

Two main types of dependency in loops

Loop Independent: Dependence in same iteration

```c
for (i = 2; i <= 4; i++)
    a[i] = b[i] + c[i];
    d[i] = a[i];
```

Loop Carried: Dependence over the iteration

```c
for (i = 2; i <= 4; i++)
    a[i] = b[i] + c[i];
    d[i] = a[i-1];
```

Loop dependency analysis

- Compiler detects loops that can be safely and efficiently executed in parallel
- To know whether usages of an array access the same memory location, compiler performs dependency tests: dataflow analysis
How about this one?

```c
int main() {
    int * data, temp, out = ... 
    for(int i = 0; i < 100; i++) {
        for(int j = 0; j < 100; j++) {
            int idx = i * 100 + j;
            temp[idx] = data[idx] + data[i];
        }
    }
    for(int i = 0; i < 100; i++) {
        for(int j = 0; j < 100; j++) {
            int idx = i * 100 + j;
            out[idx] = data[idx] + data[i];
        }
    }
}
```

Super parallel. Has data parallelism, nested parallelism, pipeline...
How to partition?

In general, compiler can’t do this arbitrarily without *hints*
OpenMP

- Standard for shared memory programming
  - Target: scientific applications.
- Specific support for scientific application needs
  - unlike Pthreads
- API is a set of compiler directives
  - Programmer inserts in the source program
  - Plus a few library functions
- Ideally, compiler directives do not affect sequential code.
  - `pragma`'s in C / C++
  - (special) comments in Fortran code.
  - If the compiler ignores them → correct single-threaded program
OpenMP API Example

Sequential code:
   statement1;
estatement2;
estatement3;

We want to execute:
• statement 2 in parallel
• statement 1 and 3 sequentially.
OpenMP API Example

OpenMP parallel code:

```c
statement 1;
#pragma <specific OpenMP directive>
statement2;
statement3;
```

Statement 2 (may be) executed in parallel.
Statement 1 and 3 are executed sequentially.

- By giving a parallel directive, the user asserts that the program will remain correct if the statement is executed in parallel.
- OpenMP compiler does not check correctness.
API Semantics

• Master thread executes sequential code.
• Master and slaves execute parallel code.
• Note: very similar to fork-join:
• But allows nesting!
OpenMP Compiler

• Sequential switch →
  • comments and pragmas are ignored.
• Parallel switch →
  • translation into parallel program.
• One source for sequential and parallel!
OpenMP Directives

• **Parallelization directives:**
  • parallel region
  • parallel for

• **Data environment directives:**
  • shared, private, threadprivate, reduction, etc.

• **Synchronization directives:**
  • barrier, critical

• Always apply to the next statement
  • must be a structured block.

• Examples
  • `#pragma omp ...
   statement`
  • `#pragma omp ...
   { statement1; statement2; statement3; }`
OpenMP Parallel Region

#pragma omp parallel

• A number of threads are spawned at entry.
• Each thread executes the same code.
• Each thread waits at the end.
• Similar to a number of create/join’s in Pthreads.

• How to get threads to do different things?
  • Through explicit thread identification (as in Pthreads).
  • …and work-sharing directives.
Thread Identification

int omp_get_thread_num()
int omp_get_num_threads()

• Library function (not annotation)
• Gets the thread id.
• Gets the total number of threads.

#pragma omp parallel
{
    if( !omp_get_thread_num() )
        master();
    else
        slave();
}
Work Sharing Directives

- Always occur within a parallel region directive.
- Two principal ones are
  - parallel for
  - parallel section
OpenMP Parallel For

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

- Each thread executes a subset of the iterations.
- All threads wait at the end of the parallel for.

```c
#pragma omp for
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ ) {
        c[i][j] = 0.0;
        for( k=0; k<n; k++ )
            c[i][j] += a[i][k]*b[k][j];
    }
```

Fork
Join

0 1 2 3 4 5 6 7
Multiple Work Sharing Directives

• May occur within a single parallel region
  
  ```
  #pragma omp parallel
  {
    #pragma omp for
    for( ; ; ) { ... }
    #pragma omp for
    for( ; ; ) { ... }
  }
  ```

• All threads wait at the end of the first for.
Conditional Parallelism

• Parallelism only useful for large problem size
• For smaller sizes, overhead exceeds benefit.

```c
#pragma omp parallel if( expression )
#pragma omp for if( expression )
#pragma omp parallel for if( expression )
```

• Execute in parallel if expression

```c
for( i=0; i<n; i++ )
    #pragma omp parallel for if( n-i > 100 )
    for( j=i+1; j<n; j++ )
        for( k=i+1; k<n; k++ )
            a[j][k] = a[j][k] - a[i][k]*a[i][j] / a[j][j]
```
Scheduling of Iterations

• Scheduling: assigning iterations to a thread.
• Default is block scheduling.
• OpenMP allows other scheduling strategies:
  • Cyclic, block, gss (guided self-scheduling), dynamic…

```bash
#pragma omp parallel for schedule(<sched>)
```

• `<sched>` can be one of
  • block (default)
  • cyclic
  • Gss
  • Etc.
Example

```c
#define THREADS 16
#define N 100000000

int main () {
  int i;

  printf("Running %d iterations on %d threads guided.\n", N, THREADS);
  #pragma omp parallel for schedule(guided) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */
  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.
Data Environment Directives

• All variables are by default shared.
• One exception: the loop variable of a parallel for is private.
• Data directives:
  • Private
  • Threadprivate
  • Reduction

```c
#pragma omp parallel for
for( i=0; i<n; i++ )
  for( j=0; j<n; j++ ) {
    c[i][j] = 0.0;
    for( k=0; k<n; k++ )
      c[i][j] += a[i][k]*b[k][j];
  }
```

• a, b, c are shared
• i, j, k are private
Private Variables

#pragma omp parallel for private( list )

- Private copy for each thread for each variable in the list.

```
for( i=0; i<n; i++ ) {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
}
```

- Swaps the values in a and b.
- Loop-carried dependence on tmp.
- Easily fixed by privatizing tmp.

```
#pragma omp parallel for private( tmp )
for( i=0; i<n; i++ ) {
    tmp = a[i];
    a[i] = b[i];
    b[i] = tmp;
}
```

- Removes dependence
Reduction Variables

```
#pragma omp parallel for reduction( op:list )
```

- op is one of +, *, -, &, ^, |, &&, or ||
- The variables in list must be used with this operator in the loop.
- The variables are automatically initialized to sensible values.

```
#pragma omp parallel for reduction( +:sum )
for( i=0; i<n; i++ )
    sum += a[i];
```

- Sum is automatically initialized to zero.
OpenMP synchronization

Implicit Barrier
- beginning and end of parallel constructs
- end of all other control constructs
- implicit synchronization can be removed with `nowait` clause

• Explicit
critical
OpenMP critical directive

Enclosed code
– executed by all threads, but
– restricted to only one thread at a time
• C/C++:
#pragma omp critical [( name ) ]new-line
structured-block

• A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name. All unnamed critical directives map to the same unspecified name.
C / C++: cnt = 0;
f=7;
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<20; i++) {
        if (b[i] == 0) {
#pragma omp critical
            cnt ++;
        }
    /* endif */
    a[i] = b[i] + f * (i+1);
} /* end for */
} /*omp end parallel */
int main()
{
    int nthreads, tid;
    int n = 8;
    #pragma omp parallel num_threads(4) private(tid)
    {
        #pragma omp single
        {
            tid = omp_get_thread_num();
            printf("Hello world from (%d)\n", tid);
            printf("fib(%d) = %d by %d\n", n, fib(n), tid);
        }
    } // all threads join master thread and terminates
}

Static int fib(int n){
    int i, j, id;
    if(n < 2)
        return n;
    #pragma omp task shared (i) private (id)
    {
        i = fib(n-1); 
    }
    #pragma omp task shared (j) private (id)
    {
        j = fib(n-2); 
    }
    return (i+j);
}
OpenMP Summary

• Programmer gives the compiler hints
• Compiler auto-parallizes based on those hints
• Seems to require a lot of hints, no?
• What do you think?
Cilk

• Goal:
  To implement dynamic, asynchronous, concurrent programs.

• Cilk programmer optimizes:
  • total work
  • critical path

• A Cilk computation:
  • dynamic, directed acyclic graph (dag)
Cilk Terms

- Cilk *program* is a set of *procedures*

- A *procedure* is a *sequence* of *threads*

- Cilk *threads* are:
  - represented by nodes in the dag
  - **Non-blocking**: run to completion: **no** waiting or suspension: **atomic** units of execution
Programming Model

• Threads can *spawn* child threads
  • downward edges connect a parent to its children

• A child & parent can run **concurrently**.
  • Non-blocking threads ➔ a child **cannot** return a value to its parent.
  • The parent spawns a *successor* that receives values from its children
Programming Model

- Thread & successor: parts of the same Cilk procedure.
  - connected by horizontal arcs
- Children’s returned values:
  - received before their successor begins
  - They constitute data dependencies.
  - Connected by curved arcs
Execution Time & Scheduling

- Execution time of a Cilk program using P cores depends on:
  - **Work** ($T_1$): time for Cilk program with 1 processor to complete.
  - **Critical path** ($T_\infty$): the time to execute the longest directed path in the dag.
  - $T_P \geq T_1 / P$ (not true for some searches)
  - $T_P \geq T_\infty$

- Cilk uses run time scheduling: work stealing.

- For “fully strict” programs
  - asymptotic optimality for:
    - space, time, & communication
Cilk Language

- Cilk is an extension of C

- Cilk programs are:
  - preprocessed to C
  - linked with a runtime library

- Declaring a thread:

  ```c
  thread T ( <args> ) { <stmts> }
  ```

  - T is preprocessed
    - C function of 1 argument
    - return type `void`

  - The 1 argument is a pointer to a `closure`
Environment: Closures and Continuations

- A **closure** is a data structure that has:
  - a pointer to the C function for \( T \)
  - a slot for each argument (inputs & continuations)
  - a **join counter**: count of the missing argument values

- A closure is **ready** when join counter == 0.
- A closure is **waiting** otherwise.
- They are allocated from a runtime heap

- **Continuation** is a data type,
  ```
  cont int x;
  ```
- Global reference to an empty slot of a closure.
- It is implemented as 2 items:
  - a pointer to the closure; (what thread)
  - an int value: the slot number. (what input)
Creating Parallel Work

• To *spawn* a child, a thread creates its closure:

  ```
  spawn T (<args>)
  ```

  • creates child’s closure
  • sets available arguments
  • sets join counter

• To specify a missing argument, prefix with a “?”

  ```
  spawn T (k, ?x);
  ```

• A *successor* thread spawned the same way except the keyword `spawn_next` is used:

  ```
  spawn_next T(k, ?x)
  ```

• Children typically have no missing arguments; successors do.
Explicit continuation passing

• Nonblocking threads ➔ a parent cannot block on children’s results.
• It spawns a successor thread.
• Paradigm called explicit continuation passing.
• Cilk provides a primitive to send a value from one closure to another.

send_argument( k, value )

send value to the argument slot of a waiting closure specified by continuation k.
Cilk Procedure for computing a Fibonacci number

```c
thread int fib ( cont int k, int n ) {
    if ( n < 2 ) send_argument( k, n );
    else {
        cont int x, y;
        spawn_next sum ( k, ?x, ?y );
        spawn fib ( x, n - 1 );
        spawn fib ( y, n - 2 );
    }
}

thread sum ( cont int k, int x, int y ) {
    send_argument ( k, x + y );
}
```
Nonblocking Threads: Pros, Cons

• *Shallow call stack.* (for us: fault tolerance)

• *Simplify* runtime system:
  
  Completed threads leave C runtime stack empty.

• *Portable* runtime implementation

Con: Burdens programmer with explicit continuation passing.
Stealing Work: The Ready Deque

- Work-stealing:
  - Process with no work selects a victim
  - Gets shallowest thread in victim’s spawn tree.
- Thieves choose victims randomly.
- Each closure has a level:
  - level( child ) = level( parent ) + 1
  - level( successor ) = level( parent )
- Each processor keeps a ready deque:
  - Contains ready closures
  - The $L^{th}$ element contains the list of all ready closures whose level is $L$. 
if (!readyDeque.isEmpty())
    take deepest thread
else
    steal shallowest thread from readyDeque of randomly selected victim
Why Steal Shallowest closure?

• Shallow threads *probably* produce more work, therefore, reduce communication.

• Shallow threads *more likely to be* on critical path.