Programming Paradigms

Concurrency (Part 2)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Summer 2020
Overview

- Introduction
- Concurrent Programming Fundamentals
- Implementing Synchronization
- Language-level Constructs
Processes, Threads, Tasks

- **Process**: Operating system construct that may execute threads
- **Thread**: Active entity that the programmer thinks of as running concurrently with other threads
- **Task**: Unit of work that must be performed by some thread
Processes, Threads, Tasks

- **Process**: Operating system construct that may execute threads
- **Thread**: Active entity that the programmer thinks of as running concurrently with other threads
- **Task**: Unit of work that must be performed by some thread
Processes, Threads, Tasks

- **Process**: Operating system construct that may execute threads
- **Thread**: Active entity that the programmer thinks of as running concurrently with other threads
- **Task**: Unit of work that must be performed by some thread
- Terminology differs across PLs and systems
- More general than, e.g., Java’s “threads”
Communication

- **Constructs to pass information between threads**
  - **Shared memory**: Some variables accessible by multiple threads
  - **Message passing**: No shared state, but threads send messages to each other
  - Some PLs provide both
Synchronization

- Mechanisms to control relative order of operations in different threads
  - Explicit in shared-memory model
    - Must synchronize to ensure that variable read sees newest value stored in the variable
  - Implicit in message-passing model
    - Sender receives message after it has been sent
Spinning vs. Blocking

- Two forms of synchronization
- **Spinning** (also: busy-waiting)
  - Thread re-evaluates some condition until it becomes true (because of some other thread)
- **Blocking**
  - Waiting threads stops computation until some condition becomes true
  - Scheduler reactives the thread
# Examples

<table>
<thead>
<tr>
<th></th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Distributed computing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td>Java, C#, C/C++</td>
<td>Go</td>
<td>Erlang</td>
</tr>
<tr>
<td><strong>Extension</strong></td>
<td>OpenMP</td>
<td></td>
<td>Remote procedure call</td>
</tr>
<tr>
<td><strong>Library</strong></td>
<td>pthreads, Windows threads</td>
<td>MPI</td>
<td>Internet libraries</td>
</tr>
</tbody>
</table>
Thread Creation Syntax

- How to create a thread of execution?
- Five answers in popular PLs
  - Co-begin
  - Parallel loops
  - Launch-at-elaboration
  - Fork (with optional join)
  - Implicit receipt
Co-begin

- Compound statement where all statements are executed concurrently
- Example (pseudo-code):

```plaintext
co-begin
    stmt_1
    stmt_2
    ...
    stmt_n
end
```
Example: C with OpenMP

```c
#pragma omp sections
{
  #pragma omp section
  { printf("thread 1 here\n"); }

  #pragma omp section
  { printf("thread 2 here\n"); }
}
```
#pragma omp sections
{
    # pragma omp section
    {
        printf("thread 1 here\n");
    }
    # pragma omp section
    {
        printf("thread 2 here\n");
    }
}
Parallel Loops

- Loop whose iterations execute concurrently instead of sequentially

- **Ex. 1**: C with OpenMP
  
  ```c
  #pragma omp parallel for
  for (int i = 0; i < 3; i++) {
    printf("thread %d here\n", i);
  }
  ```

- **Ex. 2**: C# with Task Parallel Library
  
  ```csharp
  Parallel.For(0, 3, i => {
    Console.WriteLine("Thread " + i + " here");
  });
  ```
Synchronization in Parallel Loops

- What about data races in parallel loops?
- Most PLs: Developer’s responsibility
- Some PLs: Implicit synchronization
  - E.g., forall loops in Fortran 95:
    Synchronization on every assignment
    - All reads on right-hand side are before writes on the left-hand side
Example: Fortran 95

forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall
Example: Fortran 95

forall (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end forall

Reads and writes of array elements
Example: Fortran 95

Assignments: Implicit synchronization points

forall (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end forall

Reads and writes of array elements
Example: Fortran 95

```
forall (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end forall
```

At first, all threads read from \texttt{B} and \texttt{C}
Example: Fortran 95

forall (i=1:n-1)
A(i) = B(i) + C(i)
A(i+1) = A(i) + A(i+1)
end forall

At first, all threads read from B and C
Then, all writes to A(i) happen
Example: Fortran 95

\[
\text{forall (i=1:n-1) }
\begin{align*}
    A(i) &= B(i) + C(i) \\
    A(i+1) &= A(i) + A(i+1)
\end{align*}
\text{ end forall}
\]

At first, all threads read from \( B \) and \( C \)

Then, all writes to \( A(i) \) happen

Next, all threads read the just written values from \( A \)
Example: Fortran 95

forall (i=1:n-1)
A(i) = B(i) + C(i)
A(i+1) = A(i) + A(i+1)
end forall

At first, all threads read from B and C.

Then, all writes to A(i) happen.

Next, all threads read the just written values from A.

Finally, the threads write updated values to A(i+1).
Quiz: Parallel Loops

forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall

What is the value of A after executing the loop with these initial values:

- A is 0, 0, 0
- B is 1, 2, 3
- C is 3, 2, 3
- n is 3

(Note: Arrays indices starts at 1 in Fortran)

Please vote via Ilias.
\[
\begin{align*}
T_1 & \quad i := 1 \\
1 + 3 &= 4 \\
A(1) &= 4 \\
4 + 4 &= 8 \\
A(2) &= 8 \\
\end{align*}
\]

\[
\begin{align*}
T_2 & \quad i := 2 \\
2 + 2 &= 4 \\
A(2) &= 4 \\
4 + 0 &= 4 \\
A(3) &= 4 \\
\end{align*}
\]

\[A = \left[ 4, 8, 4 \right]\]
Data Sharing in Parallel Loops

- Some PLs: Can specify which variables are shared among threads
- E.g., OpenMP
  - Shared data: All threads access same data
  - Private data: Each thread has its own copy
  - Reduction: Reduce a private variable across all threads at end of loop
Example: C with OpenMP

double A[N];
double sum = 0;

#pragma omp parallel for 
    default(shared) reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += A[i];
}
printf("sum: %f\n", sum);
Example: C with OpenMP

do double A[N];
do double sum = 0;

#pragma omp parallel for 
   default(shared) reduction (+: sum)
for (int i = 0; i < N; i++) {
   sum += A[i];
}
printf("sum: \%f\n", sum);

All variables (except for i) are shared by default
Example: C with OpenMP

double A[N];
double sum = 0;

#pragma omp parallel for 
   default(shared) reduction(+: sum)
for (int i = 0; i < N; i++) {
   sum += A[i];
}
printf("sum: %f\n", sum);

All variables (except for i) are shared by default

Exception from default:

- Each thread has private copy of sum initialized before entering loop
- At end of loop, combine all copies with +
Launch-at-Elaboration

- Associate a thread with a specific subroutine
- **Start thread when subroutine gets called**
- At end of subroutine, wait for thread to complete
- Thread *shares local variables with the subroutine*
Example: Ada

```ada
procedure P is
    task T is
        Put_Line ("In task T");
    end T;
begin
    Put_Line ("In default task of P");
end P;
```
Example: Ada

"Task" is Ada's terminology for "thread"

```ada
procedure P is
  task T is
    Put_Line ("In task T");
  end T;
begin
  Put_Line ("In default task of P");
end P;
```

"Task" is Ada's terminology for "thread"

Runs concurrently with (implicit) task of P
Fork/Join

- **Fork**: Explicit creation of a thread
- **Join**: Wait for a previously forked thread to terminate
Example: Java

class ImageRenderer extends Thread {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by the thread
    }
}

// ...

ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
Example: Java

Threads: Subclasses of Thread

class ImageRenderer extends Thread {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by the thread
    }
}

// ...

ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();
Example: Java

class ImageRenderer extends Thread {
    ImageRenderer(someArg) {
    }
    public void run() {
        // code run by the thread
    }
}

// ...

ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();
Example: Java

```java
class ImageRenderer extends Thread {
    ImageRenderer(someArg) {
    public void run() {
        // code run by the thread
    }
}
// ...
ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();
```

Threads: Subclasses of Thread

Share values with thread via arguments

Lifetime of thread
Example: C#

class ImageRenderer {
    public void Render() {
        // code to be run by the thread
    }
}

// ...

ImageRenderer rendObj = new ImageRenderer();
Thread rend = new Thread(
    new ThreadStart(rendObj.Render));
rend.Start();
// ...
rend.Join();
Example: C#

class ImageRenderer {  
    public void Render() {  
        // code to be run by the thread
    }
}

// ...

ImageRenderer rendObj = new ImageRenderer();
Thread rend = new Thread(
    new ThreadStart(rendObj.Render));
rend.Start();
// ...
rend.Join();

Just a normal method ...

... turned into a thread
Thread Pools

- **Goal**: Separate *tasks to execute* from how they are executed in threads

- **Thread pool**: Set of (idle) threads that can execute tasks
  - Reduces cost of creating and starting threads by reusing them
  - Let pool implementation decide how exactly to schedule tasks for execution
Example: Java

class ImageRenderer implements Runnable {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by this task
    }
}

// ...

Executor pool = Executors.newFixedThreadPool(4);
pool.execute(new ImageRenderer(...));
Example: Java

class ImageRenderer implements Runnable {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by this task
    }
}
// ...
Executor pool = Executors.newFixedThreadPool(4);
pool.execute(new ImageRenderer(...));

Not a Thread anymore, but only a Runnable
Example: Java

class ImageRenderer implements Runnable {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by this task
    }
}

// ...
Executor pool = Executors.newFixedThreadPool(4);
pool.execute(new ImageRenderer(...));

Not a Thread anymore, but only a Runnable

Pool with 4 threads used to execute any number of threads
Spawn and Sync in Cilk

- Extension of C language
- Programmer expresses tasks and their dependencies
  - `spawn` calls a function to be executed as a logically concurrent task
  - `sync` joins all tasks spawned by the calling task
- Scheduler assigns tasks to processor cores through work stealing
Example: Fibonacci

Sequential implementation:

```c
int fib (int n) {
    if (n < 2) return 1;
    else {
        int res = 0;
        res += fib (n - 1);
        res += fib (n - 2);
        return res;
    }
}
```
Example: Fibonacci

Parallel implementation with Cilk:

cilk int fib (int n) {
    if (n < 2) return 1;
    else {
        int res = 0;
        res += spawn fib (n - 1);
        res += spawn fib (n - 2);
        sync;
        return res;
    }
}

Example: Fibonacci

Parallel implementation with Cilk:

```cilk
int fib (int n) {
  if (n < 2) return 1;
  else {
    int res = 0;
    res += spawn fib (n - 1);
    res += spawn fib (n - 2);
    sync;
    return res;
  }
}
```

Execute in parallel with parent

Wait until children have returned
Implicit Receipt

- In remote procedure call (RPC)-based systems
- Create **thread in response to an incoming request** from some other address space
  - E.g., from another machine
Quiz: Concurrency

Which of the following sentences are true?

- Concurrency means different machines perform computations at the same time.
- In OpenMP’s parallel loops, all data is private to the respective thread.
- A thread in a thread pool may execute any number of tasks.
- The scheduler re-actives a busy-waiting thread.

Please vote via Ilias.
Quiz: Concurrency

Which of the following sentences are true?

■ Concurrency means different machines perform computations at the same time.

■ In OpenMP’s parallel loops, all data is private to the respective thread.

■ A thread in a thread pool may execute any number of tasks.

■ The scheduler re-actives a busy-waiting thread.

Please vote via Ilias.
Overview

- Introduction
- Concurrent Programming Fundamentals
- Implementing Synchronization
- Language-level Constructs